

*i*ntegrating
F_ortran
And
XML 

Participant documentation

Toby White and Andrew Walker
National Institute for Environmental e-Science
University of Cambridge

8th - 10th January 2007



Contents

Timetable and course outline	3
Welcome to iFaX	5
First, set up your environment	6
Practical 1: Writing xml with wxml	7
Practical 2: XML output from a real program	14
Practical 3: Reading XML using XPath	24
Practical 4: XML input to Fortran	31
Sources of further information	37

Timetable and course outline

Monday 8th January

9:00 - 10:00	Registration and coffee in Earth Sciences Common Room	
10:15 - 10:30	Introduction to NIEeS	Martin Dove
10:30 - 10:45	System setup and compiler install	
10:45 - 11:30	Introduction to the course and overview of XML	Toby White
11:30 - 12:00	Writing XML, introducing the FoX library	Toby White
12:00 - 13:00	Practical 1: Using FoX to write HTML and MathML	Andrew Walker
13:00 - 14:00	Lunch at Downing College	
14:00 - 15:00	XML languages	Toby White
15:00 - 15:30	Introduction to KML, Google Earth and practical 2.	Andrew Walker
15:30 - 16:00	Coffee in the Earth Sciences Common Room	
16:00 - 17:15	Practical 2: Adding XML output to real Fortran code.	Toby White
17:15 - 17:30	Review	Toby White

Drinks and dinner at Downing College from 19:00

Tuesday 9th January

9:00	Meet in the Balfour room and deal with questions	
9:30 - 10:30	The XML landscape	Toby White
10:30 - 11:00	Coffee in the Earth Sciences Common Room	
11:00 - 11:45	Introduction to scripting with XPath and an introduction to Practical 3	Andrew Walker
11:45 - 13:00	Practical 3: Using XPath to extract data from XML documents	Andrew Walker

13:00 - 14:00	Lunch at Downing College	
14:00 - 15:00	Semantics and AgentX	Phil Couch / Rik Tyer
15:00 - 15:30	Introduction to SAX	Toby White
15:30 - 16:00	Coffee in the Earth Sciences Common Room	
16:00 - 17:00	Practical 4: Use of the FoX SAX interface	Toby White
17:00 - 17:30	Course review	Toby White

Drinks and dinner at Downing College from 19:00

Wednesday 10th January

Optional day where participants can incorporate XML into their own code.

9:00	Meet in the Balfour room
10:30 - 11:00	Coffee in the Earth Sciences Common Room
13:00 - 14:00	Lunch at Downing College
15:30 - 16:00	Coffee in the Earth Sciences Common Room

Welcome to iFaX

iFaX - integrating Fortran and XML

In recent years, a number of changes have come about in the way that data is used, fuelled by a combination of increasing computer speed, and increasing network speed.

There is a growing problem of information management - computers are so fast now that we can run useful calculations, producing megabytes of output, even on our own desktop computers. In fact, we are capable of creating and collating such enormous amounts of data that we are in danger of losing our ability to sensibly manage it - it is becoming increasingly difficult to browse, search, and analyse our own data.

There is also a growing desire to be able to share data more effectively - the speed of data transfer across the internet has made it much easier to send raw data to collaborators, or even publish it to the world at large. However, it is rarely easy to immediately use somebody else's data, with no knowledge of provenance, or context, or even common data formats.

Furthermore, driven both by the increasing speed of desktop computers (and thus their ability to display high-quality graphics) and the increasing speed of the network (enabling vast amounts of data to be transferred around, thus allowing interactive, remotely hosted, applications), there is a flourishing growth in exciting and innovative visualization technologies. SVG and Google Earth are cases in point.

What all of these issues have in common is XML. XML is a large part of the answer to any data management solution, enabling storage of metadata, and enabling you to encode data in commonly understood, well-documented data formats. In addition, XML has become the language of choice for data transfer to visualization applications like Google Earth.

However, in the computational scientific world, most of our data is produced by programs written in Fortran. Fortran and XML do not naturally fit together, and there are a number of issues to be overcome in bringing a Fortran workflow into an XML environment.

In this workshop, we have drawn on the combined expertise of a number of scientists, who over the past few years, under the auspices of the UK eScience programme, have successfully dealt with XML in a scientific context, and bridging the Fortran-XML gap.

We will teaching you about XML, its background, and its uses in a scientific context.

We will introduce you to a Fortran library which will let you write and read XML directly from a Fortran program, without any intermediaries.

We will teach you how to write analysis script to quickly and easily extract data from XML, much like you would write grep or Perl scripts to do so from a text file.

And throughout the course, we will be working with real Fortran scientific software, to show you exactly how you bring existing Fortran codes into an XML-aware environment.

In short, we will bring you up to speed with the skills necessary for operating as a computational scientist in the world of XML.

First, set up your environment

The first steps are to set up your programming environment so you can complete the remainder of the exercises easily. There are three stages, (i) set up the desktop so that you can quickly find the programs you will need to use, (ii) install a Fortran 95 compiler and (iii) download and unpack the files associated with the exercises.

The desktop

You may not have used a Mac before, so if the interface is unfamiliar, please feel free to ask for help.

All the applications you need should be available in the Dock; the row of icons at the bottom of the desktop. If you hover your mouse pointer over each icon, it will tell you what they are. You should have (amongst other things):

- Terminal - this is an XTerm emulator, you will spend most of your time using this.
- Firefox - the web browser - you'll need this for the first practical.
- Google Earth - you'll need this for the second practical.

You'll need an editor to program with. "vi" and "emacs" are installed and available as on any other Unix system. If you prefer Xemacs, you should find it also on the dock, along with BBedit, a user-friendly Mac editor.

If you cannot find any of these on the Dock, don't panic, we'll show you how to put them there.

Double clicking on any of the icons will start the program.

Install G95

We will be using the free G95 compiler for this course. This compiler sticks fairly strictly to the Fortran standards. Installation of a pre-compiled binary is easy on almost any flavour of operating system.

1. To download the compressed binary open Firefox (click its blue and orange icon in the dock) and go to www.g95.org. Follow the link to "Download binaries, source and manual" at the top left of the page and scroll down to the section marked "Stable Version 0.9, August 2006". Click on the link marked "HTTP" next to "x86 OSX" and select "Save to Disk" and click "OK" when prompted. A icon labeled "g95-x86-osx.tgz" should appear on your desktop after a couple of seconds.
2. In a terminal window, type the following commands to unpack the executable and place it into your path:

```
mkdir bin
cd bin
mv ~/Desktop/g95-x86-osx.tgz .
tar -xzvf g95-x86-osx.tgz
ln -s g95-install/bin/i386-apple-darwin6.8-g95 g95
export PATH=~/.bin:$PATH
cd
```

Now add the PATH export to your `~/.bashrc` - ask if you need help.

3. Check that g95 is installed by typing “g95 -v” at the prompt, you should see some information about the setting used to build the compiler.

Download the exercises

Finally you need to download and unpack the files used for in practical 1. Download the file by pointing Firefox at http://archive.niees.ac.uk/talks/ifax/Practical_1.tgz. Click “Save to disk” and “Ok” in the dialogue and the compressed file should arrive on your desktop. Unpacking this tar archive on the desktop to give the easiest access to files. In the terminal emulator type:

```
cd ~/Desktop
tar xzf Practical_1.tgz
cd Practical_1
```

and you are ready to start practical 1. Files for the other practical sessions can be downloaded in the same way, as can worked solutions. The solutions for practical 1 can be found at http://archive.niees.ac.uk/talks/ifax/Solutions_1.tgz.

Practical 1: Writing xml with wxml

The aims of this morning’s exercises are to familiarize you with the process of compiling the FoX library and using its wxml API to produce simple xml documents. The tasks revolve around producing a simple HTML document, then modifying a program that evaluates a simple mathematical function to output HTML before finally building an HTML report containing mathematical notation.

Exercise 1.1

In the directory “~/Desktop/Practical_1/exercise_1” you should find a file named simple.f90. As the name implies this simple fortran program is designed to produce an equally simple XML document. Without compiling the code sketch out what you expect the XML document produced by this code to look like.

Space for your outline of the XML document produced by simple.f90:

Quick guide to (x)html

HTML is the markup language used to encode the structure and to a lesser extent the meaning of pages on the web. Web browsers load, parse and render a page for display based on content of the document. There are several versions of HTML and one of these (XHTML) is strictly an XML language. XHTML documents have <html> as the document root, one <head> element and one <body> element. The document head contains metadata about the document and the body contains the data that will actually be displayed by the browser. A (very) simple example looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Andrew's home page</title>
  </head>
  <body>
    <h1>Andrew's home page</h1>
    <p>I'm a postdoctoral research associate involved
      in the <i>e</i>Minerals project. This is my web site.</p>
  </body>
</html>
```

Note that this is a well-formed XML document. Other tags to note are <p>, which delimits a paragraph, <i> which marks text to be italicized and , marking bold text. In order to make sure that Firefox correctly understands the document is XHTML, files should have a '.xhtml' extension and the xmlns attribute should be included as shown above.

Reminder - compiling and linking FoX

1) Download the FoX source code from:

<http://www.uszla.me.uk/software/source/FoX/>

2) Uncompress to your Desktop, and build the libraries:

```
cd ~Desktop
tar xzf FoX-2.1.0.tgz
cd FoX-2.1.0
./config/configure
make
```

3) Return to the Practicals directory, and compile source code with relevant include flags:

```
cd ../Practical_1
g95 -c `../FoX-2.1.0/FoX-config --fcflags --wxml` simple.f90
```

4) Link source code with library and library search flags:

```
g95 simple.o `../FoX-2.1.0/FoX-config --libs --wxml` -o simple
```

5) For a single file, the compiler command line including FoX is:

```
g95 -o simple simple.f90 `../FoX-2.1.0/FoX-config`
```

Before moving on and compiling the code there are some aspects you should note (you may have already spotted some or all of these, but they are listed below anyway).

1. There are two important stylistic points to note. First most of the arguments passed to subroutines in the FoX library make use of keywords. This is useful as it makes the code more readable, is less likely to break if you upgrade to a new version of the FoX library and is the only way to deal with the many optional arguments in many of the subroutines. Second, it is worth noting the indentation scheme used. Whenever a new element is opened the level of indentation is increased, as if the call was opening an if or do block. This also helps prevent the creation of code that leads to the production of unbalanced or overlapping tags (things that are forbidden in XML)
2. There are two calls to `xml_AddAttribute`, one for each “MyXMLElement” element and they are passing different data types as the value argument of the call. To anybody versed in Fortran 77 it may come as a surprise that this can be done in Fortran, but it is permitted for many attributes of the subroutines in FoX. If you are interested in knowing how this works you could take a look at “Module Procedures” in a Fortran 90 reference book.
3. Take a close look at the string passed to the second `xml_AddAttribute` call. Some of these characters are not permitted within an XML document (at least not permitted outside of CDATA) as they have special meanings. The problematic characters are “<” and “&” - both of these are correctly escaped by FoX so they can be passed in as part of the string.

Exercise 1.2

Now it is time to compile `simple.f90`, link it against the FoX library and see if it produces the results you expect. To do this you will need to download, configure and make FoX (see the reminder below). We suggest that this is done in the “Practical_1” directory so it can be accessed easily for the whole practical.

(see the next page for compilation instructions)

Once you have compiled the code you can able to run it by typing “./simple” at the command prompt. A new file should be produced called `simple.xml`. Look at this file - does it look like the outline you sketched in exercise 1? There are a couple of things you should note related to the calls to `xml_AddAttribute`. The floating point number “2.0” is expressed in scientific notation, FoX does this for all floating point numbers it encodes, and the problematic characters in the second call have been encoded.

Exercise 1.3

Before moving on to write your own fortran program it’s worth looking to see what FoX does when a program attempts to output XML that is not well formed. There are many ways that an XML document can be badly formed and the aim of this exercise is to write a fortran code that can be compiled, but that produces badly formed XML. Modify `simple.f90`, recompile the code and run it to see what happens in the following cases:

1. Closing an unopened tag. What happens if you attempt to close an element that has not been opened. It may be particularly instructive to change the case of the string representing the final element name from “MyXmlRoot” to “MyXMLRoot”?

Answer: _____

2. Unbalanced tags. What happens when an element is closed before an element that it encloses is closed (change the order of the last two `xml_EndElement` calls, for example).

Answer: _____

3. Unclosed tags. What happens if a program terminates correctly without closing all the tags (remove the last two `xml_EndElement` calls, for example).

Answer: _____

4. Adding elements to an unopened file. What happens when you attempt to create a new element before calling `xml_OpenFile`.

Answer: _____

Exercise 1.4

Write a fortran program to produce a simple home page in html. The code should be simple (a single program block without flow control) and use FoX’s wxml interface to output XHTML. You could use `simple.f90` as a starting point. There should be an `<html>` root element with two children, `<head>` and `<body>`, the `<head>` element should have at least a `<title>` child while the `<body>` element should contain one or more `<p>` and `<h1>` elements. Compile the code and link it to the FoX libraries.

Confirm that the code writes a valid html file and check that you see the expected rendering when it is loaded into a web browser (use the “File → Open File” menu in Firefox). The example solution produces XHTML like that shown in the “quick introduction to HTML”, above.

You should also check that your file declares the correct namespace - you should see `xmlns='http://www.w3.org/1999/xhtml'` within the root `<html>` tag. If not you will need to add a namespace declaration with a call to the `xml_DeclareNamespace` subroutine immediately before your first `xml_AddElement` call. The meaning and usefulness of XML namespaces will be explained later in the course.

Exercise 1.5

The files for exercise 1.5 and 1.6 can be found in “~/Desktop/Practical_1/exercise_5”. The Fortran program contained in the file erf.f90 calculates the error function:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

using the Taylor series expansion:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)n!} = \frac{2}{\sqrt{\pi}} \left(x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{x^7}{42} + \frac{x^9}{216} - \dots \right)$$

for input values of the argument (x) and the truncation of the expansion (n). The series itself is implemented in function `erf_loop`. (Note: I have no idea if the program actually produces the correct results, or if this is a numerically stable or an efficient way to evaluate the function - the numerical and recursive nature of the program serves a purpose, which should become clear in the next exercise.)

Currently the program only writes output to the standard output stream (you should see it in the terminal when you compile and run the program). Your task for this part of the exercise is to have the program output an XHTML file containing the same data as is currently output to the terminal in addition to the current output. You should be able to view the result in a web browser.

Exercise 1.6

The aim of this exercise is to write an (XML encoded) record of the actual calculation used to produce the result of the error function. Ideally we would like to write this record in a format that can be ‘understood’ by a computer in a way that simply writing the appropriate symbols into an HTML file is not. For example, we would like a computer to be able to parse the XML document and extract the expression then produce some executable code to enable the expression to be evaluated. In short, the aim is to encode the semantics of the summation into the XML document. Do this by modifying the `erf_loop` functions to write content MathML to the same file as the HTML is being written to. Content MathML is an XML standard for expressing the meaning of a mathematical expression - and Firefox is capable of parsing MathML and producing human readable output.

A short guide to MathML is provided over the page. I’m not going to write very much more in the way of explicit instructions for this exercise - the information over the page should help get you started and you should feel free to ask one of us if you are having trouble.

A short guide to content MathML

Perhaps the best way to explain how content MathML works is by starting with an example of the encoding for $(a+b)$. The MathML to express this is

```
<apply>
  <plus/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

We use reverse polish notation and embed operations within `<apply>` elements. The fragment says "apply the addition operator (plus) to the variables (ci) a and b ". Expressions can be embedded within other expressions - so x^{2n+1} can be expressed:

```
<apply>
  <power/>
  <ci> x </ci>
  <ci>
    <apply>
      <plus/>
      <ci>
        <apply>
          <times/>
          <cn> 2 </cn>
          <ci> n </ci>
        </apply>
      </ci>
      <cn> 1 </cn>
    </apply>
  </ci>
</apply>
```

Note that `<ci>` represents a variable and `<cn>` represents a number. You can find a list of the other elements you will need to use are over the page. In order to put mathematical expressions within an XHTML document, you must embed them as shown below, and, just before you start adding MathML elements, do:

```
call xml_DeclareNamespace(xf, "http://www.w3.org/1998/Math/MathML")
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>MathML in an HTML document</title>
  </head>
  <body>
    <p> Expressing  $(a+b)$  is easy:</p>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <plus/>
        <ci> a </ci>
        <ci> b </ci>
      </apply>
    </math>
  </body>
</html>
```

continued over...

The mathML elements you will need to use are:

<math>: This must be the root element of any MathML fragment.

<root>: the nth root of a is given by:

```
<apply>
  <root/>
  <degree><ci type='integer'> n </ci></degree>
  <ci> a </ci>
</apply>
```

<power>: The power element is a generic exponentiation operator. That is, when applied to arguments a and b, it returns the value of a^b .

```
<apply>
  <power/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were evaluated at $a=5$ and $b=3$ it would yield 125.

<factorial>: $n!$ would be represented by:

```
<apply>
  <factorial/>
  <ci> n </ci>
</apply>
```

If this were evaluated at $n=5$ it would evaluate to 120.

<minus>: This can be used as a unary arithmetic operator (e.g. to represent “-x”) or a binary arithmetic operator (e.g. “x-y”). These examples would be:

```
<apply>
  <minus/>
  <ci> x </ci>
</apply>
```

and:

```
<apply>
  <minus/>
  <ci> x </ci>
  <ci> y </ci>
</apply>
```

respectively.

<pi/>: represents π .

A much longer list of operators can be found in the MathML specification at: <http://www.w3.org/TR/2003/REC-MathML2-20031021/chapter4.html>

Practical 2: XML output from a real program

In the first practical session you learnt about the rules of XML, and were introduced to the use of FoX, a Fortran library for writing well-formed XML documents. You were introduced to the FoX API, and used it to write some simple Fortran programs which created simple HTML pages.

In this afternoon's session, you will learn how you might incorporate XML output into a large existing Fortran program, of the sort which you probably work with.

You will also be introduced to the practical use of multiple XML vocabularies, and be exposed to some of the questions that arise when choosing XML languages.

For this exercise, we will be working with a seismology code - but no special knowledge is needed.

The Fortran code in question is called *hypoDD*¹, and is designed to calculate the positions of the hypocentres of a series of earthquakes. You don't need to know about the details of the calculation, but you do need to know that it starts with an input file, containing initial guesses for the hypocentre location, then runs through a series of steps, outputting its best guess at the hypocentres after each step, and refining its estimate as it proceeds. To start off, you will unpack and compile the code, and run a sample job.

(Note: if you are having difficulty, then solutions to all of these exercises are available in the file `Practical_2.tgz`)

Exercise 2.1: Using *HypoDD*

The source code for *HypoDD* may be retrieved from the same folder as all of the other programs. The file is called `hypoDD.tar.gz` - download it, and unpack it in your home directory.

```
> tar xzf hypoDD.tar.gz
> cd hypoDD
> ls
hypoDD_input    hypoDD_src
```

You will see two directories; `hypoDD_input` contains some example input files, while `hypoDD_src`, as you might expect, contains the source files. The `Makefile` has been set up correctly for your environment already, so go into the source directory and compile the program:

1 <http://www.ldeo.columbia.edu/~felixw/hypoDD.html>

```
> cd hypoDD_src
> make
...
```

You should now have an executable, `hypoDD`.

Now use the executable with one of the example input files. Go into the Example directory you saw previously, and run the example input:

```
> cd ../hypoDD_input
> ../hypoDD_src/hypoDD hypoDD.inp
...
```

Output will be produced on the screen, most of which you can ignore. However, if you look at the contents of the directory after the job has completed, you will notice that several new files have been created. We are only interested in the two files, `hypoDD.loc` (which contains the initial guess at the earthquake locations, and `hypoDD.reloc`, (which contains the final, refined, estimate of the earthquake locations). This is the primary input/output of the program, and a seismologist would be interested in visualizing and exploring these numbers - so it is the data in these files we will be interested in exporting to XML in this practical.

Exercise 2.2: Compile *hypoDD* with FoX

We are now going to include XML output into *hypoDD*. As mentioned, you might want to produce multiple sorts of information from a program. In this practical we are going to record two sorts of information. Firstly, we will record metadata about the run, and secondly, we will record some of the output data in an XML format as well, for visualization and post-processing purposes.

However, before we do that, you need to know how to include the FoX library into an existing program; how to set up a program directory with source files and a Makefile so it is all correctly compiled.

Firstly, copy the FoX distribution into the source directory, unpack it & rename it.

```
> cd ../hypoDD_src
> tar xzf FoX-2.1.0.tar.gz
> mv FoX-2.1.0 FoX
```

Now we need to edit the Makefile so that `make` will configure and compile FoX, and link the `hypoDD` executable to the library.

To configure, add a line like the following:

(NB, remember that Makefile needs a `TAB`, not a `SPACE`, on the second line!!)

```
FoX/.config:
  (cd FoX;config/configure FC=$(FC))
```

This tells make how to configure FoX, making sure that it uses the same compiler as the main program. Then, for compiling FoX, add:

```
FoX/.FoX: FoX/.config
  (cd FoX;make)
```

This tells make how to compile FoX, and that it must configure it first. But we need to ensure that this compilation takes place before *hypoDD* is compiled, so we add a further dependency on all the *hypoDD* files - find the target line beginning '%o', and add a dependency on the FoX compilation process, like so:

```
%.o: %.f FoX/.FoX
  $(FC) $(FFLAGS) -c $(@F:.o=.f) -
```

The FoX library needs to be visible to the compiler when it compiles the object files - look again at this line - it provides the compilation command for all of the *hypoDD* files. You'll see that it includes a variable `FFLAGS`, which is designed to hold compiler flags. This is how we will tell the compiler about FoX. Find where `FFLAGS` is set, and add the FoX-compile-command:

```
# Flags for g95 compiler
FFLAGS = -ffixed-line-length-132 -O -I$(INCLDIR) \
`FoX/FoX-config --fflags --wxml`
```

Finally, to ensure that linking happens correctly, find the target in the Makefile which causes the linking step (hint: it's the one which uses `LDFLAGS`), and do two things:

- Firstly, add the FoX-link-command to `LDFLAGS`.
- Secondly, add a dependency on `FoX.make` (so that FoX gets compiled)

```
LDFLAGS = `FoX/FoX-config --libs --wxml`
$(CMD): $(OBJS) FoX.make
  $(FC) $(OBJS) $(LIBS) -o $@ $(LDFLAGS)
```

Now type make again, and see if it works!

Exercise 2.3: Create output metadata

We now have *hypoDD* compiling with FoX - but we haven't yet told it to produce any output. The first output we are going to create is some simple metadata; a short output file

Dublin Core Metadata Initiative (DCMI)

Namespace URI: “<http://purl.org/dc/elements/1.1/>”.

The DCMI (<http://dublincore.org>) is a standard for cross-domain information resource description, using 15 defined elements to describe the metadata of a document. Its namespace URI is “<http://purl.org/dc/elements/1.1/>”.

A few of them are shown here:

<title>	Title of the document.
<creator>	An entity (person, organization, service, program) primarily responsible for creating the document.
<subject>	Keywords describing the subject of the document.
<description>	A free-text description of the document.
<contributor>	An entity (person, organization, service, program) who contributed towards the creation of the document.
<date>	A date associated with the document (eg date of creation)
<relation>	A reference to a related resource.

describing the data produced, for posterity. We will use the Dublin Core² metadata standard. Dublin Core is a widely understood, very simple set of metadata terms, used across a wide range of domains.

For a real scientific program you would probably use a different, more specialized, metadata vocabulary, but here we are aiming for simplicity. Figure 2.1 summarizes a few useful elements from the Dublin Core vocabulary. Using the WXML calls you learnt in the previous practical, teach *hypoDD* to construct a short document, ‘`metadata.xml`’, describing the metadata associated with the output of the run. Use as many of the DC metadata tags listed in Figure 2.1 as you like.

Hint: the main program is in the file `hypoDD.f`. Also note that the variable `dattim` is a string containing the current date and time.

Exercise 2.4: Create output data for visualization.

So far we are producing one metadata output document. That’s not very exciting, though. An important use of XML is to encode data in a standard form, so you can visualize it through standard tools, without having to write special-purpose visualizers for every tool. *hypoDD* comes bundled with a set of Matlab scripts for plotting its output on a grid, but that makes it very difficult to compare with other data, or, indeed, even with a simple map.

The standard visualization tool for most geographical data these days is Google Earth (which you will find already installed on your computer), and it accepts input as an XML language, KML³. KML is a large and complex language, but we only need a fraction of it here. Figure 2.2 describes the aspects of KML you need to know for this practical.

The output of *hypoDD* is a step-by-step series of latitude/longitude coordinates, with associated seismological data. The first part of this exercise will show how to write these points out in XML so that Google Earth will understand them.

2 <http://dublincore.org/>

3 <http://earth.google.com/kml/>

KML - Keyhole Markup Language

KML is an XML language for feeding geographical data to Google Earth. A few of the most useful elements are explained below.

<code><kml></code>	The root element of the document. Must carry the Namespace URI.
<code><Document></code>	A wrapper for a set of Folders
<code><Folder></code>	A collection of Placemarks
<code><name></code>	The Name of a Folder or Document - GE will use this to display additional information
<code><Placemark></code>	Instructs Google Earth to place a placemaker over:
<code><Point></code>	Defines a geographic point, and contains:
<code><coordinates></code>	The coordinates of the point; its the longitude and latitude, separated by a comma.

Thus the simplest useful document looks something like:

```
<kml xmlns="http://earth.google.com/kml/2.0">
  <Placemark>
    <Point>
      <coordinates> 52.203267, 0.119788 </coordinates>
    </Point>
  </Placemark>
</kml>
```

Try saving that as a file 'here.kml' and loading it into Google Earth.

When you have more than one Placemark, you should put them in a Folder - and when you have different kinds of Placemarks, you should group them into different Folders.

```
<kml xmlns="http://earth.google.com/kml/2.0">
  <Document>
    <Folder>
      <name>First Folder</name>
      <Placemark>
        <Point>
          <coordinates> 52.203267, 0.119788 </coordinates>
        </Point>
      </Placemark>
    </Folder>
    <Folder>
      <name>Second Folder</name>
      <Placemark>
        <Point>
          <coordinates> 52.203267, 0.119788 </coordinates>
        </Point>
      </Placemark>
    </Folder>
  </Document>
</kml>
```

The initial points are written out to the *.loc file at about line 500, and final points to the *.reloc file at about line 1000 of hypoDD.f, and the latitude and longitude are held in the arrays src_lat and src_lon. Use WXML calls to write out a KML document, containing the data in both these arrays.

Hint - note that Google Earth will not understand exponential notation! Refer to the FoX documentation on how to control numerical formatting through FoX.

To check your output is correct, load it into Google Earth - there are instructions in your notes. You should see a series of points representing earthquakes near San Jose, California.

Hint - if you group your KML Points into named Folders, and add a name tag to the Document, then Google Earth will be able to better display your points.

Exercise 2.5: Elaborating the output data.

The KML you have output so far only shows the locations of the earthquakes. hypoDD has calculated the depth of the earthquake, and also outputs magnitude, amongst other things. It would be nice to display those too in some way.

Here, though, we encounter a problem. Google Earth knows nothing of seismology, so there is no way to unambiguously store data of earthquake magnitude within KML. Furthermore, GE also does not understand depth *beneath* the earth's surface. So we are faced with a problem - how should we store this portion of the data?

We will explore several ways to do this. The easiest way is to simply attach a description to each point (see Figure 2.3). Adapt your KML output to include a description tag on each earthquake point, with the magnitude data in it. (Magnitude is held in the array ev_mag). Load the output into Google Earth, and click on the placemarkers to view the description.

This still doesn't let you easily see at a glance the trends in earthquake magnitude. It is possible to use more complicated aspects of KML styling to put different icons over the point, with varying sizes and colours, to represent the variation of various quantities. However, it is not our intention here to teach KML, so a version of hypoDD has been prepared which already does this transformation. The source code and executable for this can be

KML - Keyhole Markup Language

You can add information using the <description> tag:

```
<kml xmlns="http://earth.google.com/kml/2.0">
  <Document>
    <Placemark>
      <description>Magnitude: 3.56</description>
      <Point>
        <coordinates> 0.119788, 52.203267 </coordinates>
      </Point>
    </Placemark>
  </Document>
</kml>
```

found in the file `hypoDD_markup.tgz`. Run it and view the output. In this case, all the earthquakes are represented by coloured circles whose diameters are proportional to the magnitude of the earthquake.

Notice, though, that transforming the original data in this fashion into KML involves throwing away all the original information about the magnitude, in favour of opaque information on colours. Though a great improvement for visualization, it rather defeats much of the point of using XML, which is to have strongly marked-up, well-described, and easily processable data. We will explore other avenues later on in this practical.

Exercise 2.6: Abstracting XML writing

Before we continue with trying to encode our seismology data, we will take a short detour. You will notice that (unless you've been very careful) `hypoDD.f` is now getting quite cluttered with FoX calls. This is a problem for a number of reasons:

- It makes the source code look ugly. Fortran looks ugly enough as it is, there's no need to make it worse!
- It makes it hard to understand the flow of control in the program - the XML writing interrupts your reading of the program's logic.
- Because of this, it will annoy other people working on the same program - if they don't understand XML, they just see a lot of clutter getting in the way of a perfectly good program.
- And, if they don't understand XML, they might well accidentally edit the FoX calls in a way that damages the XML output, by for example misspelling a name, or altering the order of FoX calls to produce an invalid document.

It's been our experience that XML-izing a program can only be successfully, politically as well as technically, if you abstract away the complexity as much as possible.

You will see that the code for writing out a latitude/longitude coordinate involves quite a lot of XML/FoX verbiage to essentially transfer only two numbers into the KML document.

Write a Fortran module which isolates all of the FoX calls and variables behind subroutines, leaving in the main program only calls like

```
call kml_AddPoint(lat, lon, magnitude)
```

Since you are using a module, you will need to tell make about the dependency. Append a line like the following to the `Makefile` (assuming your module is in a file called `xml.f`)

```
hypoDD.f: xml.o
```

The model, marked-up version of *hypoDD* mentioned above has references to a full Fortran library of such KML calls.

Exercise 2.7: Multiple namespace documents

In exercise 2.3, you created a short Dublin Core metadata document describing the *hypoDD* output. You've now also created a KML document containing geographical data. While it is useful that both documents exist, and the metadata document describes the source of the KML document, it would be even better if the KML document carried its own description with it.

This will be possible by using an important feature of XML - namespaces. Namespaces are a means of distinguishing different XML languages within the same document. They have the great advantage that while all the data is stored in the same document, an XML-reading tool is able to extract only the data that it understands and cares about, and ignore the rest.

In this case, we are going to embed our metadata document within the KML document. Although Google Earth does not understand Dublin Core metadata, the presence of DC elements inside KML will cause no problems, due to namespacing. Similarly, any tools which understand DC metadata can read the KML document and understand the metadata even if it knows nothing of KML.

If you don't remember how namespaces work, refer back to the notes from the previous talk, and the FoX API.

Take your current version of *hypoDD*, and alter it so that it only outputs one XML document, but one which has namespaced DC metadata incorporated in the KML. Inspect the output to ensure that both DC and KML data is present, then load the result into GE to see that the DC metadata is ignored.

Exercise 2.8: Additional private data

We could now use this ability to incorporate multiple namespaces to try and solve the problem we faced in exercise 2.5 - how should we incorporate our seismological data?

Well, it turns out that there is an XML language dedicated to seismological data - QuakeML⁴. Google Earth does not understand it, but there are tools which do, and it is a small and easy language to understand. So we will encode our data in QuakeML frag-

QuakeML - Earthquake Markup Language

Information on earthquake events is held within a <location> tag like so:

```
<location xmlns="http://quakeml.ethz.ch/ns/eventlist/location"
  unique_id="6883774">
  <origin-time>
    <year>1997</year>
    <month>4</month>
    <day>12</day>
    <hour>23</hour>
    <minute>0</minute>
    <second>1</second>
  </origin-time>
  <latitude unit="degree">47.51</latitude>
  <longitude unit="degree">9.44</longitude>
  <depth unit="km">10</depth>
  <magnitude unit="M">3.2</magnitude>
</location>
```

4 <http://www.quakeml.ethz.ch/quakeml/>

ments, embedded within a KML document. A brief overview of the relevant parts of QuakeML is given in Figure 2.4.

Write a program that, for each KML point it outputs, embeds a properly-namespaced QuakeML `<location>` tag within each KML `<Point>`. Initially, just output the latitude, longitude, depth, and magnitude; then output the time data if you can.

Once this is complete, you can now produce, with every run of the Fortran code, a single XML document which contains

- metadata on the program run
- result data in a recognized, domain-specific XML vocabulary
- marked up with a display XML vocabulary such that it is immediately visualizable alongside other geographical data.

Google Earth will read just the KML, DC-aware metadata tools will read just the metadata, and seismological tools will read the QuakeML.

This concludes this practical session. Tomorrow morning's session will involve using these joint KML/QuakeML documents in some realistic XML analysis scenarios.

Conclusion

This practical session should have taught several things. You should have learnt:

- How to alter the compilation process of an existing code to incorporate FoX.
- Sensible ways to incorporate FoX calls into an existing code.

In addition, the following general principles should have been at least partially illustrated:

- XML output code should be well-abstracted away from the main flow of the program.
- The choice of XML language depends upon the use that will be made of it - different choices are suitable for metadata handling, visualization, data storage etc.
- It is not necessary to use only a single language, if documents may have multiple purposes. Namespacing allows incorporation of multiple vocabularies.

And there is a broad view you should take away from this, when adapting your own code to produce XML. If you are very lucky, then your data (or metadata) is of a sort for which someone has already written an extensive vocabulary and set of tools.

However, it is quite probable that you are not so lucky. So, what to do? Where possible, you should choose an existing vocabulary for your XML data (you can only use web browsers if your text is in HTML, or Google Earth if your coordinates are in KML).

However, you may find that no existing XML vocabulary completely covers all your needs. In this case, several choices are available to you. Above we explored two of them;

- in exercise 2.5, we put the data into a general-purpose KML tag, which made it almost visualizable, but not in a useful way, and the data was not meaningfully marked up.

- in exercise 2.8, we embedded the data in a suitable, existing XML language, which kept the data, but without the ability to visualize it directly.

In the next practical session we will explore some other methods for approaching the same problem, involving transforming between XML languages.

Practical 3: Reading XML using XPath

Yesterday you looked at ways of producing XML output from Fortran codes. In today's exercises you will look using the XML in ways beyond reading it into a web browser or Google Earth. The aim of the first session is to give you an understanding of XPath, a rather high level interface to data from an XML document used by many XML technologies. It's worth designing your documents so that they can be easily used with XPath expressions. There is no Fortran XPath interface, so this exercise will be done using Python.

Don't worry, though, you don't need to know Python, and XPath is a language-agnostic interface; for this exercise you will only be doing XPath, not Python. But this also reflects practicalities, in that even for data produced by Fortran codes, much analysis and post-processing is done using scripting languages such as Perl or Python.

The files needed for this practical can be downloaded as a compressed archive from: http://archive.niees.ac.uk/talks/ifax/Practical_3.tgz.

Exercise 3.1: Installing an XPath library

By default the computers we are working with do not come with a pre-installed XPath library for python. However, it's very easy to download and set up such a library for your own use:

1. First download the file lxml-1.1.2.tgz from <http://codespeak.net/lxml/lxml-1.1.2.tgz> and extract the files from the archive. For example, using curl you would type the following at the command line:

```
cd ~
curl http://codespeak.net/lxml/lxml-1.1.2.tgz > lxml-1.1.2.tgz
tar xzf lxml-1.1.2.tgz
```

2. Now build the package (this involves compiling some C code that is made available to python. Just type (the second step may take a couple of minutes):

```
cd lxml-1.1.2
python setup.py build
```

3. That's it - you have now created libXML bindings to your python installation! If you wanted to install the library for all users of the system (and you had root access) you would now type "setup.py install", but you don't so instead we will just set up things so that you have access to the library. Type:

```
mv build/lib.macosx-10.4-i386-2.5/lxml $HOME
export PYTHONPATH=$HOME
```

You should now be ready to start using XPath. Note: if you change terminal windows you will have to type the final "export" line again. Add the export command to your .bashrc, like you did for g95. (Ask for help if necessary)

Exercise 3.2: Exploring XPath

Files for this exercise can be found in the exercise_2 directory. The file xpath.py is a simple python script designed to read an XML document, run an XPath query and print the

result. The XML document `monty.xml` contains some quotations from the film “Monty Python and the Holy Grail”. Take a look at the XML file and the `xpather.py` script and try to deduce what will be output when the script is run. Now run the script by typing:

```
python xpather.py
```

You should see “[‘Monty Python and the Holy Grail’]” - which is python’s way of printing a one element list (array) with a single string element.

For the rest of this part of the exercise you should change the XPath query to extract other data. To change the `xpather.py` script you just need to edit and save it- there is no need to compile a python script. The only line you will need to change is:

```
answer = docRoot.xpath("/film/@name")
```

Modify the XPath query to extract the following information:

1. The date of the film encoded in the “<data date='1975'>” element. *Hint: you will need to add an additional location step in the XPath query, and modify the attribute name.*

XPath query: _____

2. The names of all five of the listed Pythons. *Hint: The fact that this requires a list of names to be returned does not add to the complexity of the XPath query needed, again you just need to change a location step and modify the attribute name.*

XPath query: _____

3. The quotations of all the characters. *Hint: you won't need any attributes, just three location steps.*

XPath query: _____

4. You can also complete part 3 with a single location step. Can you construct such an XPath expression?

XPath query: _____

5. Modify your solution to part 2 so that only the name of the Python who played Sir Boris is reported. *Hint: You will need to use a predicate (in square brackets).*

XPath query: _____

6. Write a query to return the quotation of Eric Idle.

XPath query: _____

7. Write a query to return the quotation of the character Sir Bedevere.

XPath query: _____

XPath syntax

In building XPath expressions think of the XML document to be parsed as a tree-like data structure made up of nodes in a way that is very similar to locating files in a directory structure. These nodes represent all aspects of the document including all elements, attributes, namespaces and text. There is also a special root node element represented by a slash (/) at the start of an XPath query.

Queries are run by passing a query string, which is executed from a given element node. Normally this will be from the root element node, but sometimes it is useful to execute XPath queries on subsections of documents, in which case the node that is queried is called the *context node*.

Query strings are built out of a series of location steps delimited by slashes. If the query starts with a slash it represents an *absolute path* (starting from the root), if not, it is a *relative path*, starting from the current *context node*. Thus simple queries examining element nodes look like directory paths (but never end with a slash):

```
"/a/simple/absolute/query" or "a/simple/relative/query"
```

To extract the contents of a text node add a "text()" location step to the end of the query:

```
"/a/query/to/get/a/text/node/text()"
```

Attributes are indicated with an "@" character: to retrieve the value of the attribute "name" an attribute, use a query like:

```
"/a/person/@name"
```

Namespaces are indicated with a colon, so if the previous query was within a namespaced document, it would look like:

```
"/ns:a/ns:person/@name"
```

(NB when you are using namespaces, namespace prefixes must be included on every element - yes, this is annoying. Sorry!).

You can write short tests (or *predicates*) within an XPath expression using square brackets, and test attribute values against strings (included in single quotes). So a query to return "age" attributes only from `person` elements which also have "name=Andrew" look like:

```
"/ns:a/ns:person[@name='Andrew']/@age"
```

Finally, the double slash means "look for this path fragment anywhere within the document" - not just from the root. The following will return a list of the names of all the people in the document, wherever they are located:

```
"//ns:person/@age"
```

XPath Functions look rather like functions in many other languages, with a function name followed by arguments in parentheses. Functions are probably more useful when embedding XPath in an XML document than when it is used within a script (as the scripting language is likely to include its own useful functions). Some potentially useful functions include:

“count (/some/query)”
returns the number of nodes matched by the query.

“substring(string, n, m)”
returns the nth to the mth characters from the string. TOHW

Exercise 3.3: XPath with Namespaces.

In the exercise_3 directory you should also find a file called `monty_ns.xml`. As the name suggests, this is a version of `monty.xml` with XML namespaces added. We saw in Practical 2 that documents with multiple namespaces are a useful way to combine different XML vocabularies in the same file. In this exercise we you will examine ways to use XPath on such a document. In `monty_ns.xml` I have imagined that two XML vocabularies exist, one to describe films with the namespace `http://www.example.com/films`, and one to describe quotations with the namespace `http://www.example.com/quotes`. We will use this mixed vocabulary for this exercise. TOHW

First try some of the solutions to exercise 3.2 with the file `monty_ns.xml`. A copy of `xpather.py` is included in the directory. Note that the file name `monty.xml` has been changed to `monty_ns.xml` on the line to load the XML: TOHW

```
docRoot = lxml.etree.parse(source="monty_ns.xml") TOHW
```

Do any of the previous expressions work?

You should find that all the expressions return an empty list (`[]`). The XPath expressions have not matched any nodes. This is because we have not specified the namespace and so the XPath library is only looking for nodes with no defined namespace. All nodes in `monty_ns.xml` have a namespace defined, either directly with an `xmlns=` declaration, or by inheritance. TOHW

The file `xpather_ns.py` is a python script set up to do the same work as `xpather.py`, but for namespaced documents. Take a look at the `xpather_ns.py` file. You should note two changes. First the line: TOHW

```
namespaces = {'q': 'http://www.example.com/quotes',  
              'f': 'http://www.example.com/films' }
```

has been added. This declares a python dictionary of namespaces and related local short names (f and q) to enable their use in XPath expressions with much less typing. Secondly, the call to the XPath library has been modified:

```
answer = docRoot.xpath("/f:film/@name", namespaces)
```

to tell the library to use the dictionary of namespaces. One way to think of this is that the dictionary lists all namespaces this script is designed to know about. XPath expressions will simply ignore namespaces that the script does not understand.

Run `xpather_ns.py` - does it work? You should now repeat Exercise 3.2 with this namespace aware version.

1. The date of the film encoded in the “`<data date='1975' />`” element. Do you need to give a namespace to each data element in the search path, or is the namespace inherited through the query?

XPath query: _____

2. The names of all five of the listed Pythons.

XPath query: _____

3. The quotations of all the characters. *Hint: remember that `<quote>` elements are in a different namespace to `<film>`, `<data>` and `<comic>`.*

XPath query: _____

4. You can also complete part 3 with a single location step. Can you construct such an XPath expression?

XPath query: _____

5. Modify your solution to part 2 so that only the name of the Python who played Sir Boris is reported.

XPath query: _____

6. Write a query to return the quotation of Eric Idle.

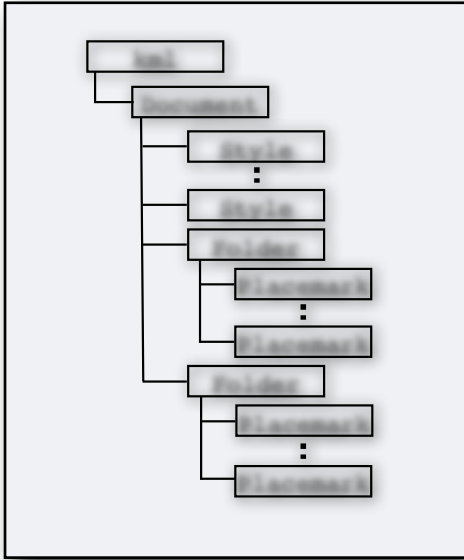
XPath query: _____

7. Write a query to return the quotation of the character Sir Bedevere.

XPath query: _____

Exercise 3.4: Matching Nodes

The remainder of this morning's practical involves using XPath to extract data from the mixed namespace KML document produced by yesterday's final exercise. In case you did not finish the final part of that exercise, a suitable XML document named `hypoDD.kml` is provided in the `exercise_4` directory along with all the python scripts needed for the the remainder of this practical.



The basic structure a typical KML document is given in the figure on the left in terms of a tree-like representation (not all elements are shown). The document root is represented by a `<kml>` element, this (may) have one or more `<Document>` child elements called, each of which contains some number of `<Style>`s and `<Folder>`s. The python script "document_info.py" is designed to print some information about the KML file, but the XPath expressions are missing. Fill them in to make the script work.

The first XPath expression is intended to return the name(s) of any `<Document>`s which are children of the root element. The second two expressions are supposed to return numbers, specifically the number of

`<Style>`s and `<Folder>`s belonging to the `<Document>`s.

First XPath query: _____

Second XPath query: _____

Third XPath query: _____

Hint: Absolute XPath expressions (starting with a / including all the elements needed to find the information) can be used for this exercise. There is an XPath function that can be used to find the number of nodes returned by an XPath query.

Exercise 3.5: Nodsets and loops

The python script `folder_info.py` is designed to extract some simple information regarding all the folders within a KML document. Because folders can be nested inside other folders to an arbitrary depth it is not possible to write a general expression to find all folders in a document with an absolute query. Instead this script is intended to perform data extraction in two stages. The first query should extract a list of all folders and stores the result in the variable 'folders' (which will be a list). This XPath query should match all `<Folder>` elements in the document wherever they are located. The second XPath query is located within a loop which extracts each node in series from the list in the 'folders' variable and places it in the object 'node' which is passed to the subsequent XPath expressions. These two expressions need to contain relative XPath queries (not starting with a '/'). The first should return the name of the folder represented by 'node' and the second should count the number of Placemarks within the folder.

First XPath query: _____

Second XPath query: _____

Third XPath query: _____

Exercise 3.6: Analysis

The python script `calculate_moves.py` performs some more involved analysis of the output data using the quakeML data embedded within the KML document. Specifically, the script is designed to work out how far each of the earthquake locations have been moved during the *hypoDD* run represented by the document. Six XPath expressions are needed. In order, these need to:

1. Extract a list of placemarks within the folder with the name 'Initial positions'.

XPath query: _____

2. Extract the `unique_id` from the quakeML location element embedded in each placemark in turn. This expression involves a change of namespace.

XPath query: _____

3. Extract the latitude and longitude of each event.

XPath query: _____

XPath query: _____

4. Find the latitude and longitude of the event with a matching `unique_id` from the 'Final positions' folder.

XPath query: _____

The script should then print the distance that *hypoDD* has moved each earthquake during its refinement process.

Practical 4: XML input to Fortran

The first two practical sessions dealt with how to produce XML from Fortran code. The last session showed you how to use XPath to write scripts to analyse that data. However, there is as yet no XPath interface to Fortran, and sometimes you may need to read XML documents into Fortran. In this practical session, you will learn how.

In the earlier talk on the XML landscape, a number of interfaces were mentioned. The historically earliest was SAX. It is conceptually the simplest interface; and is also the easiest to implement, and there is a Fortran SAX interface in FoX.

This practical is designed to give you an understanding of how to use the FoX SAX interface to import data from an XML file into a Fortran program.

The FoX documentation gives an introduction to the concepts behind SAX, and some initial examples of its use. SAX relies on an event-based callback programming model - the XML parser sends messages to the main program, which are received by handlers provided by the programmer. It can be tricky to get your head around programming in this style, so in this practical session, you will work through some example exercises to learn how to do this.

We will be using as input the earthquake data XML files that we taught *hypoDD* to produce in the second practical session, and the example programs you will write are designed to work towards the sort of data handling you might want to do in a real Fortran program using scientific data.

Download the file `Practical_4.tgz`, and unpack it. Inside you will find several directories - one for each practical. Start in `Exercise_1` for the first exercise, and continue onwards. Each practical involves taking the skeleton of an existing program already set up for FoX SAX (in every case, this program is called `exercise-X.f90`), and adapting it to perform some task. If you get stuck on any of the exercises, complete solutions are available in the same directory, as a file called `solution-X.f90`.

Compiling the exercises:

In each case, the tutorial program comes with a copy of FoX in a subdirectory called `FoX`. Each exercise can be compiled by the following command line:

```
g95 exercise-X.f90 -o exercise-X `FoX/FoX-config --sax`
```

Exercise 4.1: Watching SAX events

A sample SAX test program is provided. It consists of two things; a module containing a list of “handlers”, and a driver program. The handlers do nothing except print out when they are called; and the driver program does nothing but pass the handlers into FoX.

Compile the program, and execute it. It will read in the file `input.xml`, and output a list of events. Compare the list of events to the file `input.xml`, and you should see the correspondence.

You may see rather more `Character` events than you were expecting. This is because SAX will report all characters it finds between tags - even if they are just whitespace. Most of those character events are reporting empty space.

Edit the handlers, so that the start and end element handlers print out the name of the element, and the characters handler prints out the characters it receives.

Exercise 4.2: Attribute dictionaries

Attributes are handled in a slightly more complicated way within SAX.

If you look at the `startElement` handler, you'll see that it takes an argument "atts". This contains a dictionary of attributes - that is to say a list of name/value pairs. There is a long list of functions and subroutines to manipulate this dictionary, which are explained in full in the FoX documentation, but there is a brief summary below.

Attribute dictionaries:

Attribute dictionaries consist of a series of entries, which hold information about each entry. If you need to worry about namespaced attributes, these can be quite complex, but that is fairly rare. For the purposes of these exercises, you need only think about attributes as name-value pairs.

`startElement` will pass a dictionary containing all the attributes for a given element. You can get at this list of attributes using the following Fortran calls:

- `len(atts)` gives the number of attributes
- `getQName(atts, i)` gives the name of the *i*th attribute
- `getValue(atts, i)` gives the value of the *i*th attribute
- `hasKey(atts, name)` gives true or false according to whether the attribute 'name' exists
- `getValue(atts, name)` gets the value of the attribute called 'name' if it exists.

Alter the `startElement_handler` subroutine so that it prints out all of the attribute name/value pairs.

Now, write a `startElement_handler` which looks for the units of latitude, and stores it in a module variable. At the end of the program, once the document has been parsed and the file closed, print out that module variable.

Exercise 4.3: Parsing a simple XML document

So far, all the XML data manipulation you have done has disposed of the data immediately. For a real program, usually you want to read in the data, and keep it around to do something with it later. In order to do this, you need to make use of persistent variables somewhere else to record the data, and also to record where you are in the process of reading the document.

For example, if you want to read the characters within a certain tag, you can't rely on the characters callback telling you where it is; it doesn't give you that information. So you need to keep track of which `startElement/endElement` events you have been received, and only read the character data when appropriate.

The easiest way to do this is using module variables. If you place a single logical variable within your module, you can switch it on and off as you enter and leave an element, and then choose what to do with any characters events received in the interim.

Similarly, if you retrieve some data from an event, and want to keep it around, you can put it into a module string variable, and then use it again later. (see the box below for an explanation of module variables)

Using your input file and framework, write a program that reads the text of the latitude and longitude variables, as well as their units, and then prints it out like so:

```
The earthquake location was latitude XXX UNITS and longitude YYY UNITS.
```

You'll need to keep track of your location in the file, in order to read the correct characters, and then preserve the character and attribute values.

Fortran module variables:

If you are not very familiar with Fortran 90/95, module variables may be new to you. Within a Fortran module, you can keep subroutines (as we have been doing for the event handlers) and also variables. Any variables declared in the first section of a module are accessible to all subroutines within that module (and to any other parts of the program which use that module.) *NB - you must remember to SAVE the module variable, or its value may not be preserved.*

```
module tiny
  integer, save :: i = 0
contains
  subroutine in
    i = 1
  end subroutine in
  subroutine out
    print*, i
  end subroutine out
end module tiny
```

In this module, both subroutines can read and write the variable `i`.

Exercise 4.4: Namespaces and SAX

You have learnt that namespaces can be used to disambiguate various elements, and to distinguish different XML vocabularies.

SAX will provide data to do this namespace disambiguation for you. For every element it encounters, it will report both the Name of the element, but also its Namespace URI. When dealing with namespaced documents, you often don't care about the elements "*name*" any more - you care about the combination of its "*localname*" and its "*namespaceURI*".

Namespaces in SAX:

When you receive a `startElement` or `EndElement` event, it comes with three strings. So far, you've only paid attention to the name. However, in the presence of namespaces, names don't matter - what matters is the other two strings, `namespaceURI` and `localname`. Here are some examples of what SAX will report to you on finding various tags:

1. `<description xmlns="http://purl.org/dc/elements/1.1/">`
 `name = "dc:description"`
 `localname = "description"`
 `namespaceURI = "http://purl.org/dc/elements/1.1/"`
2. `<dc:description xmlns:dc="http://purl.org/dc/elements/1.1/">`
 `name = "description"`
 `localname = "description"`
 `namespaceURI = "http://purl.org/dc/elements/1.1/"`
3. `<description xmlns="http://earth.google.com/kml/2.0">`
 `name = "description"`
 `localname = "description"`
 `namespaceURI = "http://earth.google.com/kml/2.0"`
4. `<kml:description kml:xmlns="http://earth.google.com/kml/2.0">`
 `name = "kml:description"`
 `localname = "description"`
 `namespaceURI = "http://earth.google.com/kml/2.0"`

The first and second tag are equivalent; as are the third and fourth.

Note in particular how the first and third tag have the same `QName`, and are only distinguishable by their `namespaceURI`.

You can see from this how testing a combination of `localname` and `namespaceURI` will let you use SAX to check for the right tag.

The dictionary functions `hasKey` and `getValue` can both be used with namespaces:

- `hasKey(attrs, uri, localname)` gives true or false according to whether the attribute with the specified uri and localname exist
- `getValue(attrs, uri, localname)` gets the value of the attribute with the specified uri and localname

To illustrate this, the `input.xml` for this exercise has `description` elements from both KML and Dublin Core. Starting with the same skeleton program as before, write a program to separately extract and print out the description metadata from KML and from DC.

Reminder: the KML namespace is `"http://earth.google.com/kml/2.0"`, and the DC namespace is `"http://purl.org/dc/elements/1.1/"`.

Exercise 4.5: Extracting and presenting data from QuakeML

For this exercise, the input file is the final output from *hypoDD* as generated yesterday. It is often useful to take data from an XML format, and reformat it for human consumption in a more familiar way.

In exercise 4.3 you learnt how to use module variables to keep track of where you were in the document. You probably used a simple logical switch to see if you were in the correct element. However, that approach gets quite unwieldy when large numbers of tags are in evidence. In such cases, it is useful to use a *state variable*, which is a module variable which takes on different values according to the *state* of the parser - ie where it is in the document. For example, you could assign the values:

ID	LATITUDE	LONGITUDE	DEPTH	MAGNITUDE
16484	3.728530120850e1	16802	19860	17723
16527	3.728570175171e1	16821	19888	17526
17496	3.729219818115e1	16827	19992	17779
16521	3.728530120850e1	16838	20113	17794
17842	3.728829956055e1	16841	20328	17834
16635	3.729570007324e1	19159	20622	18466
16576	3.728699874878e1	16885	20978	18608
17929	3.728919982910e1	16911	21561	18708
16659	3.728929901123e1	17074	21644	18857
16701	3.729169845581e1	17272	22144	19686

0: not in an interesting tag.
1: in a latitude tag
2: in a longitude tag
etc.

Then, when reading in characters, you can simply check the value of the state variable to choose what to do.

Starting from the same skeleton SAX program, write a program to read data about the first ten earthquakes, and produce a table for presentation something like the box above.

In addition to the state variable, you'll need to keep account of how many values you've read - use another module variable for that.

Reminder - to convert a Fortran string to a number:

```
read(string,*) number
```

Query: how would you go about doing this if you didn't know ahead of time the number of data items you wanted to read?

Exercise 4.6: Perform a calculation

Above and beyond simple presentation, a real program will probably want to manipulate the data it's read in. This presents a new problem though - so far, we've been treating the data only as strings, but if you want to manipulate it, you'll need to treat it as numerical data.

Using your answer from exercise 4.5, at the end of the run, once you've collected all the data for presentation, calculate the centroid of the earthquake locations.

Exercise 4.6: Teaching hypoDD to read QuakeML

You are probably not going to get this far through the practical session with much time to spare; but having got here, you understand how to read data from an XML file into Fortran data structures.

hypoDD could be adapted, instead of reading its input data from text files as it does now, to read its data from XML files of the sort we have been dealing with.

Think about how this might be done, and look at the *hypoDD* source code again. Note that earthquake event data is read in at about line 150 of `getdata.f`.

Conclusions

This practical was designed to do three things

- Understand how to program against a callback API.
- Understand how to program with the FoX SAX API.
- Understand how to use a SAX API to build up data structures from an XML document.

It is important to realise, though, that although SAX is a conceptually simple API to work with, it becomes rapidly very complicated to use on complicated XML documents with lots of variation in structure, many different child elements and attributes etc. It was usable in this case only because our XML format is simple and predictable.

It is also important to understand that we could only do this because we knew the format of the XML we were expecting, so we didn't do any checking that the XML was of the right structure, we just assumed it. In XML-reading applications which will accept input from a wider variety of sources, much more robust error checking is necessary.

In either case, it might well be more sensible to do the XML input in a higher-level-language, using higher-level APIs, like XPath, as you were shown this morning.

Sources of further information

In putting together this workshop we have made use of many books and other resources that you may find useful as you continue to explore the XML ecosystem. We list these sources and other resources below, grouped by theme.

Fortran

- The best up to date text book on Fortran95 is probably *Fortran 95/2003 explained* (2004) by Michael Metcalf, John Reid and Malcolm Cohen (Oxford University Press, ISBN 0-19-852693-8). This is the latest in a long series of *Fortran explained* books that have followed the development of the language.
- Other useful Fortran resources include Starlink's Theory and Modelling Resources Cookbook at: <http://www.starlink.rl.ac.uk/star/docs/sc13.htx/index.html> and the various Numerical Recipes books (see: <http://www.nr.com/>).

General information about XML

- XML development is co-ordinated the World Wide Web Consortium (W3C). Links to the various XML standards ("recommendations") can be found at <http://www.w3.org/> and a brief outline of current activities can be found at <http://www.w3.org/XML/>.
- Also www.w3schools.com has a good series references and tutorials on XML and related technologies.

The Fortran XML library (FoX)

- The FoX home page is at <http://www.uszla.me.uk/software/FoX.html>, the up to date documentation is at <http://www.uszla.me.uk/FoX/DoX/> and the source (for various versions) is at <http://www.uszla.me.uk/software/source/FoX/>.
- There is also a low traffic mailing list. Subscription is via a web based form located at <http://www.uszla.me.uk/cgi-bin/mailman/listinfo/FoX/>.

Information about specific XML languages

- **XHTML**: Much useful information about XHTML and related technologies can be found from the World Wide Web Consortium (W3C) home page at <http://www.w3.org/>. Useful tools and documents include a validator, the specification(s) and information about related technologies.
- **MathML**: The most useful resource for MathML is probably the W3C page on the language located at <http://www.w3.org/Math/>. Of particular interest is the specification for version 2 located at <http://www.w3.org/TR/2003/REC-MathML2-20031021/>, content MathML is described in section 4.
- **KML**: Documentation, tutorials and the specification are available from Google, linked from <http://earth.google.com/kml/>. Google Earth is available from <http://earth.google.com/>.
- **quakeML**: Documentation can be found at <http://www.quakeml.ethz.ch>.
- **Dublin Core**: Documentation at <http://dublincore.org>.
- **The Open Geospatial Consortium**: Looks after Geography Markup Language (GML). <http://www.opengeospatial.org>.

- **Other XML languages:** A fairly comprehensive list of other XML languages can be found at <http://xml.coverpages.org/xmlApplications>.

XPath

- A useful (if perhaps dated) book on XPath is *XPath and XPointer* (2002) by John E. Simpson (O'Reilly, ISBN: 0-596-00291-2).
- The ultimate reference work is the specification, which is a W3C Recommendation located at <http://www.w3.org/TR/xpath>.
- The Python XPath library and documentation can be downloaded at <http://codespeak.net/lxml>.
- The Perl XPath library is called XML::XPath. CPAN (the Comprehensive Perl Archive Network, <http://www.cpan.org/>) and the library can be installed using the CPAN module. Documentation is at: <http://search.cpan.org/dist/XML-XPath/XPath.pm>. To the best of our knowledge this is the only XPath library for perl.
- The standard C XML library can be found at <http://xmlsoft.org>, this has bindings for many other scripting languages.