

Atmospheric Chemistry on the Web

Professor Simon Clegg
School of Environmental Sciences
University of East Anglia

Professor Anthony Wexler
Air Quality Research Centre
University of California at Davis

The *E-AIM* model (<http://www.aim.env.uea.ac.uk/aim/aim.php>) is a Fortran atmospheric chemistry model with a web interface which has been used for over a decade by researchers and students who carry out ~13,000 individual calculations annually. The primary application of the model is the chemical thermodynamics of atmospheric particles related to air quality and climate science, both of which are recognised as important environmental problems. NAG software has been a key part of the model from its inception, and in this article we outline how we have benefited from collaboration with NAG. We have found that the usage and impact of our model have been greatly increased by its availability on the web – a feature that is relatively easy to implement – and we provide a simple example showing how to do this, and guidance that may be valuable to Fortran programmers and NAG customers in many fields.

The Science

Suspended, micron sized, particles in the air affect atmospheric visibility and are central to problems such as air quality and climate change through their ability to scatter light and their role in cloud formation as condensation nuclei. In order to predict the effects of these aerosols, atmospheric scientists must calculate the water uptake of the particles (many of which are water soluble) in response to changes in temperature and relative humidity, whether they are solid or liquid, and how volatile components such as nitric acid and ammonia partition between the air and the particles.

Our model calculates the equilibrium state of a volume of atmosphere, containing specified amounts of aerosol components, as a linearly constrained multi-dimensional minimisation of the Gibbs energy of the system. The amounts of the different chemical species in the system – which are the ‘x’ variables in the calculation – can differ by more than 15 orders of magnitude which makes the problem difficult to solve. The use of non-linear constraints, for example that the partial pressure of ammonia in the air is related to its activity in liquid aerosol particles, is precluded by the fact that it is not known initially whether liquid particles will exist, and therefore whether the constraint applies.

NAG provided the solution by adapting the NAG Fortran Library routine E04UCF to run in extended precision (REAL*16), and so accommodate very large ranges in the values of the variables and their contributions to the objective function, and by showing us how to add scaling of the variables and constraints. The E04UCF routine has proved reliable in over a decade of use as our chemical model has increased in complexity – most recently with the expansion of the number of atmospheric organic compounds contained in the model, and the addition of a hydrophobic (water free) particle phase which increases the number of variables. We continue to work with NAG to port our code to different machines and compilers, and occasionally evaluate competing minimisers. Our work is supported by funding agencies in the U.K., in Europe, and in the USA.

Fortran Applications and the Web

Adding web interfaces to complex scientific computer models brings the following benefits:

- access, for anyone in the world with an internet connection;
- easy-to-use interfaces for different types of calculation, created using html;
- greatly increased usage and impact relative to “command line” versions of the program, which might only be distributed by email;
- only a single copy of the software to maintain, on the server that runs the application;
- usage can be tracked in great detail – not just the types of calculations carried out and how often the application is used, but also statistical information about the users based on their IP addresses.

After placing our Fortran *Aerosol Inorganics Model* (<http://www.aim.env.uea.ac.uk/aim/aim.php>) on the web in 1998 we expected, even within our own field, to be followed by many other web-based applications. This has not happened, perhaps because Fortran programmers do not always appreciate how simple it is to call a Fortran program from a web page, read input from it, and write output that appears in the user’s browser window. In this section we show how this is done, using two simple examples in which a Fortran program reads two numbers entered by a user into a web form, adds them together, and writes the sum to a browser window.

In the first example, the results are written to a new page in the browser, and in the second they are written to a ‘box’ on the same html page that was used for input. We assume the use the NAG Fortran compiler, but also identify some compiler dependencies and explain the minor code changes needed for users of the Intel and Lahey/Fujitsu compilers with which we also have experience. We also outline how to set up a development environment on a Windows PC, so that it acts as a web server to itself.

The evolution of our own site over time – starting with something only a little more complex than the examples below – can be seen at the Internet Archive (enter “www.uea.ac.uk/~e770/aim.html” in the address box on the <http://www.archive.org/index.php> page). Contact Simon Clegg (s.clegg@uea.ac.uk) if you have any questions regarding this article.

Example 1

First create a text file (called `example1.html`) containing a simple html page on which the two numbers to be added together can be entered:

```
<html>
<body>
<p>Enter numbers in the two boxes, and press the "Calculate Sum" button:</p>
<form action="/cgi-bin/example1.exe" method="post">
  <p>First number: <input type="text" name="number1" id="num1" value="" size="10" /> </p>
  <p>Second number: <input type="text" name="number2" id="num2" value="" size="10" /> </p>
  <input type="submit" value=" Calculate Sum" />
</form>
</body>
</html>
```

This page contains an html form, between the opening `<form>` and closing `</form>` tags. The form contains two input text boxes, with names “number1” and “number2”, and a submit button. When this button is pressed the contents of the two boxes are sent to the Fortran executable `example1.exe` which must be present in the `cgi-bin` directory of the web server. The `<p>` and `</p>` tags mark the beginning and ending of paragraphs, and in this example are used to ensure that the text and two input boxes appear on different lines on the screen. The boxes also have unique `id` values (“num1” and “num2”), which are only used in our second example. The use of web forms, and html markup language, is described at length in many books, and at <http://www.w3schools.com/tags/default.asp>.

The [Common Gateway Interface](#) standard (CGI) is used for communication between the Fortran executable `example1.exe` and the web server, and the data entered on the html form are sent to the Fortran executable using the “post” method, specified in the `<form>` tag. The data are made available to the program by the web server software (Apache, in our case) so that they can be read from the standard input (unit 5, or *). Similarly, the output from the program to the standard output (unit 6, or *) will appear in the user’s browser window.

The data from the form are sent to the Fortran executable as a text string, consisting of successive “name=value” pairs, separated by ampersands. So, if the user enters 1.5 for the first number, and 2.9 for the second, the text string sent to the Fortran executable will be: “number1=1.5&number2=2.9”. The maximum length of data strings sent by the “post” method appears to be determined by web server settings, but is generally very large and unlikely to be a limitation.

Some characters are [URL encoded](#) before being transmitted to the executable. For example, a space entered by the user in the text box appears in the transmitted string as a “+”. A “+” entered by the user appears as “%2B”. The characters that are transmitted unchanged are: *, -, _, a-z, A-Z, 0-9, and the decimal point “.”. The encoded characters should be returned to their original values (in the Fortran program) before attempting to read values from the string. For simplicity we have omitted this from the example program, as well as code that is needed to trap errors in the input. Note that NAG have written a Fortran CGI interface library module, in which html form data are read into a Fortran character array (<http://www.nag.co.uk/nagware/Examples/cgi.f90>).

The example program to read the data from the html page is shown below, followed by brief descriptions of the code in each of the numbered sections.

```

program example1
implicit none

C ..(1) Declarations
C -----
character (len=100) :: LineIn = ' '
character (len=10)  :: string = ' '
integer :: lenLineIn, startValue, endValue, ichar
real :: sum_of_numbers, var1, var2

C ..(2) Determine the length of the data string
C that is to be read in the next section.
C -----
Call Get_Environment_Variable('CONTENT_LENGTH', string)
READ(string,*) lenLineIn

C ..(3) Read the data from the html form into the
C variable LineIn, a single character at a time.
C -----
DO ichar=1,lenLineIn
  READ(*,ADVANCE='NO',FMT='(A)') LineIn(ichar:ichar)
ENDDO

```

```

C ..(4a) Locate and read the value of 'number1' from LineIn
C -----
startValue = INDEX(LineIn,'number1=') + 8
endValue = startValue + INDEX(LineIn(startValue:),'&') - 2
READ(LineIn(startValue:endValue),*) var1

C ..(4b) Locate and read the value of 'number2' from LineIn
C -----
startValue = INDEX(LineIn,'number2=') + 8
READ(LineIn(startValue:),*) var2

sum_of_numbers = var1 + var2 ! add the numbers together

C ..(5) Send a header to the browser, identifying
C the type of information that will be sent.
C -----
WRITE(*,('Content-type: text/html',//)) ! remove for the second example

C ..(6) Write the html results page to the browser,
C with the sum of the two numbers.
C -----
WRITE(*,(' (1X,'<html><body>')) ! remove for the second example
WRITE(*,100) sum_of_numbers
WRITE(*,(' (1X,'</html></body>')) ! remove for the second example

STOP

100 FORMAT(1X, ! change "1X" to "/1X" for the second example
> '<p>Success! The sum of the two numbers is given below:</p>',
> E12.4)

END

```

(1) There are character variables `LineIn` (to hold the data posted from the web page), and `string` (to receive the value of the `CONTENT_LENGTH` environment variable). The integer variables are `lenLineIn`, which is the number of characters in `LineIn`; and `startValue` and `endValue` which are the index numbers of the of the first and last characters of the value of “number1” in the text string, and then of the first character of “number2”. Real variables `var1` and `var2` are the variables to which the values of `number1` and `number2` are assigned, and `sum_of_numbers` is the sum of `var1` and `var2` which is printed out as the result.

(2) The number of characters of data sent from the web page is made available in the environment variable `CONTENT_LENGTH`. The value of `CONTENT_LENGTH` is itself a string and, using the intrinsic subroutine `Get_Environment_Variable`, we place it in a local character variable `string`. Finally, we read the integer value `lenLineIn` from `string` using a simple `READ` statement and list-directed formatting.

The routine `Get_Environment_Variable` is a Fortran extension offered by the NAG compiler. The Intel and Lahey/Fujitsu compilers (and probably others) instead provide the routine `getenv`, which has a similar functionality. (This routine is also present in NAG Fortran, in the module `f90_unix_env`, and can be used as an alternative to `Get_Environment_Variable`).

(3) Here the data from the html form are read, one character at a time, from the standard input. This seems to be the most compiler-dependent part of the program, and the `READ` statement in the example does not work with either the Intel or Lahey/Fujitsu compilers. For these compilers, it should be replaced by:

```
iErr = getC(LineIn(ichar:ichar))
```

where `iErr` is an integer variable that must be declared with the other integer variables above. It receives the error code from `getC` each time a character is read (a value of zero indicates success).

(4) The values of text boxes with names “number1” and “number2” are read from `LineIn` into real variables `var1` and `var2` and then added together.

(5) Before writing the result to the user’s browser as a new page, it is essential to write a header, followed by blank lines, to identify the type of output that is being sent. This header need only be written once. In a `WRITE` statement to a printer, the first character is always used for carriage control but there is no standard defined for other output devices, and compilers differ in their behaviour. Using NAG Fortran the correct statement for output of the header is:

```
WRITE(*, '( "Content-type: text/html",//)')
```

With Intel and Lahey/Fujitsu compilers an initial space is needed:

```
WRITE(*, '( " Content-type: text/html",//)')
```

If this header is not written correctly then the browser is likely to display an obscure error message, or treat the output of the Fortran program as content to be downloaded rather than displayed.

(6) Finally, the results of the calculation can be written to the standard output, as an html page. This could be done in a single `WRITE` statement but, for clarity, we have divided it into three.

The first statement starts the page with `<html>` and `<body>` tags. At this point there are two choices for formatting the output. If the Fortran program produces extensive output which is already formatted for output to screen or a printer then this can be used as-is by writing it within `<pre>` and `</pre>` (for “pre-formatted”) tags. The output will appear in the user’s browser window in a Courier (monospace) font, formatted in the same way as if it had been written to a printer or a video screen by the program. Alternatively, the user can include other html markup language in the format statements to design the page (in terms of font, layout, and colour) as required. In this example we do the latter, placing the text between `<p>` and `</p>` (paragraph) tags, so that the sum of the numbers that the program has calculated appears immediately below the text.

Last, we close the html page with `</body>` and `</html>` tags.

Example 2

Many calculations are likely to have only single line answers, and it can be desirable for practical reasons – and to achieve a good user interface – for the calculated result from the Fortran program to appear on the same page as the problem was submitted from, and without altering the other contents of the page. We have implemented this on our website in a simple estimator for densities of liquid organic compounds (<http://www.aim.env.uea.ac.uk/aim/density/density.php>).

How is this done? We make use of Ajax (Aynchronous JavaScript and XML) to submit the problem and receive the results, while still using the Fortran application almost unchanged from the first example. Although Ajax methods are now very widely used there are differences in their implementation by different browsers. Fortunately these complications can be avoided by using the

Prototype JavaScript framework, which deals with these dependencies internally, and also provides a simple function with which we can call our Fortran executable from *any* browser. The Prototype source code (it is free) can be downloaded from <http://www.prototypejs.org/download> as the text file `prototype.js`. We will include this code in the example html page so that it is downloaded to the user's browser – and all the functions made available – when they load the page.

The new html example page, `example2.html`, is shown below. Note that html comments are enclosed between `<!-- -->` tags, and in JavaScript text following a `“//”` is also treated as a comment (similarly to the Fortran `“!”`).

```
<html>
<head>

  <!-- Javascript library -->
  <script src="prototype.js" type="text/javascript"></script>

  <!-- define a local function -->
  <script type="text/javascript">
    function ajaxPost() {
      // Post data from the web page to the cgi-bin program using Ajax,
      // and write the results to the named <div> on the same page.
      //
      // The first argument of Ajax.Updater is the name of the container
      // (a <div>) where the output obtained from the url (the second
      // argument) will go. In the {} are a series of options:
      //
      // method: 'post',      ..send information to the url using Post
      // parameters: {..},   ..the posted data
      // onCreate:          a function() {..}, not used
      // onError:           a function() {..}, writes a simple message
      // onComplete:       a function() {..}, not used
      //
      new Ajax.Updater("results_div", "/cgi-bin/example2.exe",
        {method: 'post',
          parameters: {number1: $F('num1'),
                      number2: $F('num2')}}),

        onCreate: function() {
          // do nothing..
        },
        onError: function() {
          // show an error message
          var errorMessage = 'An error occurred before ' +
            'the Fortran program was called. Please try again.';
          $('results_div').innerHTML = errorMessage;
        },

        onComplete: function() {
          // do nothing...
        }

        } //end Ajax.Updater options
    }; //end Ajax.Updater function

  } //end ajaxPost function
</script>

</head>

<body>

<p>Enter numbers in the two boxes, and press "Calculate Sum":</p>
```

```

<form>

  <p>First number: <input type="text" name="number1" id="num1" value="" size="10" /> </p>

  <p>Second number: <input type="text" name="number2" id="num2" value="" size="10" /> </p>

  <p>Results will appear in the box. Click here:
    <input type="button" value=" Calculate Sum " onclick="ajaxPost();" />
  </p>

</form>

<div id="results_div" style="border-style:solid; border-width:2px;
                           border-color:black; height:5em; padding:1em;"> </div>

</body>
</html>

```

Look first at the final sections of `example2.html`, within the `<body>` `</body>` tags. There are two differences from the earlier example. First, the `<form>` tag no longer has an `“action=”` attribute. Second the button on the page is now `type=“button”` (rather than `type=“submit”`) and there is a new attribute `onclick=“ajaxPost();”`. This means literally what it says: when the user clicks on the button using a mouse the JavaScript function `ajaxPost()`, which was defined in the header, is called.

The `<div>` element, which we have given an `id` equal to `“results_div”`, is where the results of the calculation by the Fortran program will be written. We have also specified a fixed height, and a black border to make it visible in the browser window.

In the top part of `example2.html` there is a new tag: `<head>`. Inside this element we do two things: first, load the `prototype.js` source code file using a pair of `<script>` tags. Notice that no path is specified in the `src=“prototype.js”` attribute, meaning that the file should be present in the same directory as `example2.html`.

Second, inside another pair of `<script>` tags we define the JavaScript function `ajaxPost()` that will be called when the ‘Calculate Sum’ button is pressed. This function does only one thing: it calls the Prototype function `ajaxUpdater` (for documentation, see the Prototype website) which has four main arguments:

1. the `id` value of the html `<div>` element to which results of the Fortran program will be written,
2. the URL to be called (here it is the `cgi-bin` program),
3. the method (`post`) by which the form data - i.e., the numbers entered on the page by the user - will be passed to the `cgi-bin` program,
4. the parameters to be sent, expressed as `name:value` pairs.

The names `number1` and `number2` are the names of the input text boxes on the page. `“$F”` is a Prototype function (defined in `prototype.js`). The expression `“$F(‘num1’);”` means that the function returns the value of the html element with an `id` equal to `‘num1’`. In other words, the function gets the value the user has entered in the first text box and associates it with parameter `“number1”`. The `$F` function is called analogously for the second text box. The end result is that the text string posted to the fortran executable is identical to that in the first example.

The remaining `AjaxUpdater` arguments deal with other actions to be taken on creation of the function and on completion (none), and a simple error message that we will not describe here.

Does the Fortran program require any changes to be used in this second example? There are two: first, the header line in part (6) of the code and the first and last `WRITE` statements in part (7) are not required and can be deleted. This is because the program is writing to an existing html page not a new one. Second, the output needs to begin with a blank line so the `FORMAT` for the single remaining `WRITE` statement must begin with `"/1X,"` instead of `"1X"`.

Other Considerations

This article has shown that calling Fortran programs from web pages requires only simple code. Below we briefly summarise a number of other issues that we have found to be important or useful for real-world applications. At the end of this article are links to sites that describe other Fortran/web applications.

(1) A development environment on a PC. Our entire website, which includes five different Fortran cgi-bin executables, is developed and tested first on a PC and then transferred to a web server that is publicly accessible. We use the [Xampplite](#) package for Windows which is free. It includes Apache web server software, as well as PHP (a scripting language) and MySQL (a database). Xampplite is simple to set up and use, though we recommend that if you are new to this that you get help from your IT support to make sure that Apache is configured to recognize the directories that you have placed your html files and cgi-bin programs in. This is essential, and involves reviewing the settings and entries in the `httpd.conf` text file.

In our PC setup, the directory `e:\www\projects` is where all our html files are placed, and is given an alias in the `httpd.conf` file (with the statement: `Alias /projects "e:/www/projects"`). So, the address `"http://localhost/projects/example1.html"` opens `example1.html` file in the `e:\www\projects` folder when Xampplite is running.

In `example1.html`, the location of the Fortran executable is given as `"/cgi-bin/example1.exe"`. On our PC, the file `example1.exe` is actually present in directory `c:\xampplite\cgi-bin`, and the `httpd.conf` file therefore contains the statement:

```
ScriptAlias /cgi-bin/ "/xampplite/cgi-bin/",
```

and also a set of permissions for this directory, within the tags `<Directory "/xampplite/cgi-bin"> </Directory>`. The [Apache documentation](#) should be consulted for details.

Firefox is the most commonly used browser for development work, because of the many free add-ons that are available. The most useful of these are Firebug, which detects JavaScript errors, and Web Developer which contains numerous tools including html validation (via a link to <http://validator.w3.org>). Code should be tested to make sure it works with all major browsers. Most of our users are academics and students, and Firefox and Internet Explorer are the main browsers with which the site is accessed (49% and 45% of users, respectively), followed by Opera (2%) and Safari (2.5%).

(2) Validation of user input. An alternative to writing error traps in the Fortran code is to use JavaScript to catch invalid user input. This is described in most JavaScript textbooks and works in the following way: the html page is coded so that, when the "Submit" button is pressed, a JavaScript function first checks the form data for validity. If errors are found, the function writes a message to the

user's browser and the cgi-bin program is not called. If there are no errors, the function returns a 'success' value and the data are posted to the Fortran executable in the usual way.

(3) *Execution errors in the Fortran code.* If your Fortran cgi-bin program is compiled with trace and debug options on, any messages describing execution errors are likely to appear **not** on your screen but in the web server error log when the program is called from an html page. If the server software is Apache, then this log is a text file called `error.log` which will be found in the `apache/logs` subdirectory.

(4) *Plotting graphs of results.* By using Fortran libraries such as GINO, which can output a plotted graph as a `.gif` or `.png` file, graphics can be incorporated into a website to plot the results of calculations. We have implemented this on our site (e.g., <http://www.aim.env.uea.ac.uk/aim/model2/mod2t.php>) in the following way:

First, the Fortran program that does calculations on the user's input data writes results to a set of temporary files, and then writes an html page to the browser from which the user can choose what graphs to plot (i.e., select the x and y variables, the ranges of the axes etc.). Next, when the user has made these choices – also in an html form - pressing the "Submit" button sends the information to a second Fortran cgi-bin executable. This first of all reads the data from the temporary files and creates the graph as a `.gif` or `.png` file on the server. It then writes an html frame to the user's browser a page that refers to the file which is then automatically displayed.

(4) *Recording the use of the web site.* Usage logs can be created by the fortran program, recording the types of calculation, when they were carried out (using the `Date_and_Time` intrinsic routine) and, by reading the environment variables `REMOTE_HOST` and `REMOTE_ADDRESS`, the host machine that requested the calculation. Website usage can also be tracked in great detail by using the [Google Analytics](#) service. This is free, and is implemented simply by adding a few lines of JavaScript to each html page on the site.

Other Examples

A Native Fortran-77 CGI Interface: <http://www.nber.org/sys-admin/fortran-cgi/>

Doppler Shift (an example of the POST method): <http://www.fcc.gov/mb/audio/bickel/doppler-shift.html>

COLORIT (an example of the GET method): <http://www.fcc.gov/mb/audio/bickel/colorit.html>
