XML Toolbox for Matlab

Release Version 2.0
October 2003

Summary: The XML Toolbox for Matlab converts Matlab variables into XML format and vice versa. This document describes where & how to obtain the software, its installation and usage. It details the transformation of Matlab variables to XML, lists some frequently asked questions and contains an extensive range of examples.

Authors: Marc Molinari <m.molinari@soton.ac.uk>
Jasmin Wason <jasmin@soton.ac.uk> (Schema in matlab.xsd)

Investigator: Simon Cox <sjc@soton.ac.uk>

Status: This document describes the XML Toolbox for Matlab Release Version 2.0. The functions have been tested and are stable.

Changes: Release 2.0 includes some patches to version 1.1 and some new features. The main novelty is the ability to read (almost) any XML format and turn it into a Matlab structure data type.

Comments: Please get in touch with us via http://www.geodise.org if you have any comments regarding changes, alterations, improvements or future versions.

CONTENTS

VERSION CONTROL

20/12/2002 Release 1.0

28/02/2003 Release 1.1 (corrected <…/> empty elements and space arrays).

20/10/2003 Release 2.0 replaces type-based by name-based XML formatting and can read most XML documents into Matlab variables/ struct data types (if data is not contained within attributes). Corrected writing and reading of doubles with more than 6 decimals.

Future Work: Version 3.0 will contain additional functionality for attribute handling.

1 SOFTWARE LICENCE / COPYRIGHT

---

* Please note that "Matlab" is a registered trademark of The Mathworks, Inc, http://www.mathworks.com, with which the authors of the XML Toolbox are not affiliated.

2 INTRODUCTION

The XML Toolbox for Matlab* contains bi-directional conversion routines between the XML format and Matlab data types. These routines are available through Matlab functions and can be used for formatting, parsing, storing and retrieving Matlab variables in human-readable XML format and vice versa. They can also be used to compare Matlab datatypes and structures to any level of nesting by comparing their XML string representation.

The requirement for such a tool originated from the fact that it is not possible to store a representation of (nested) structures or more complex variables in the more and more popular XML format from within Matlab with its current built-in `save()` function and the provided xmlread and xmlwrite methods require extensive parsing using java functionality before they can be used as conversion tools.

The XML Toolbox for Matlab has been developed jointly between the Grid-enabled Optimisation and Design Search in Engineering project (Geodise, http://www.geodise.org) and the DTI-funded Grid-enabled Electromagnetic Optimisation (GEM, http://www.soton.ac.uk/~gridem) project at the Southampton Regional e-Science Centre (www.e-science.soton.ac.uk).

Prerequisites:
The toolbox is programmed entirely in Matlab and hence requires no specific Java environment and should work with most Matlab versions (tested with Matlab 5.3, 6.1, 6.5).
The `strsplit()` tool function is required (version ≥ 0.4, included).
--
**MATLAB is a registered trademark of The MathWorks, Inc. (http://www.mathworks.com). Other product or brand names are trademarks or registered trademarks of their respective owners.*

3 DOWNLOAD & INSTALLATION

The XML Toolbox for Matlab can be found on the web at `http://www.geodise.org` or at `http://www.soton.ac.uk/~gridem` . Please follow the links to the XML Toolbox download form.

Unpack the xml_toolbox_2.0.zip file obtained from the webpage to your local file system. You will require a password for unzipping some of the files. Please use the password you obtained from us during registration.

The following files should be present:

```
INSTALL.TXT, LICENCE.TXT,
xml_help.m, xml_format.m, xml_parse.m, xml_save.m, xml_load.m,
matlab.xsd, strsplit.m, tests/xml_tests.m, doc/xml_toolbox
```

Finally, start up Matlab, change into the Toolbox directory, and type `xml_help` at the Matlab prompt. If everything went fine, the help for the XML Toolbox functions should be displayed. Each function is easy to use and contains its own help section in the header (standard Matlab practice).

To test, try to execute the following at the Matlab command prompt:

```
>> xml_format( 'Hello World!' )
>> str = xml_format(17.29)
```

```
>> a = xml_parse(str)
>> whos( 'a' )
```

If you intend to make frequent use of the XML Toolbox, you can add a command such as
`addpath('C:/your/path/to/xml_toolbox')`
to your `startup.m` file in the `matlab` subdirectory of your home directory.

4 IMPLEMENTATION DETAILS

Function description

`xml_help.m`	- displays information about the provided functions
`xml_format.m`	- converts a Matlab variable/structure into XML string
`xml_parse.m`	- converts a XML string back into Matlab variable/structure
`xml_save.m`	- saves a Matlab variable/structure into .xml file
`xml_load.m`	- loads a .xml file and converts it into Matlab variable/structure
`tests/xml_tests.m`	- performs a number of tests to verify that the functions work
`strsplit.m`	- utility function which splits a string at specified characters
`matlab.xsd`	- Schema for Version 1.x XML files, which can still be parsed

Specific function calls:

```
xstr = xml_format( var, [att_switch], [name], [level] )
```

`xml_format` translates the variable `var` into a name-based XML string `xstr`.

`var`	Matlab variable, matrix or structure of type `char, double, complex, struct, sparse, cell, logical`
`att_switch`	Optional. 'on' writes attributes, 'off' ignores attributes. (default: 'on')
`name`	Optional. Overwrites name of XML root element. (default: 'root')
`level`	Internal. Increases tab padding at beginning of `xstr`. Should not be used by user!
`xstr`	Returned string containing XML description of variable `var`.

Examples:

```
>> v.name = 'Project A';
>> v.id.user = 'mm';
>> v.id.proj = datestr(now);

>> str1 = xml_format(v)
>> str2 = xml_format(v, 'off')
>> str3 = xml_format(v, 'on', 'mydata')
>> str4 = xml_format(v, 'off', 'mydata')
```

```
var = xml_parse( xstr, [att_switch], [X], [level] )
```

`xml_parse` parses the XML string `xstr` and translates it into a Matlab variable `var`. This is a non-validating parser. DTDs or Schemas are ignored.

`xstr`	XML string. Can either be with type attributes as written by `xml_format` with att_switch set to "on" or a XML string without element attributes. Please note that only XML element content gets translated into Matlab variable content, not data contained in the attributes themselves.
`att_switch`	Optional. 'on' reads attributes, 'off' ignores attributes and interprets all contents as char data type. (default: 'on')
`X`	Optional. Matlab structure which gets extended by the contents of the XML string (if string can be converted into struct data type).
`level`	Internal. Represents level of child structures. Should not be used by user!
`var`	Matlab variable, matrix or structure.

Version 1.x had an additional output variable: `xmlV`. Version 2.0 does not read the XML version string any more and thus this output is meaningless. For reasons of backwards compatibility this function will return an empty string if a second output is requested, ie `[v, xmlver] = xml_parse(…)` returns `xmlver=''`. This return value may be removed entirely in future versions.

Examples:

```
>> xstr = '<root>
            <alpha type="double">3.1415</alpha>
            <beta type="struct">
              <x type="double"> 1.0 </x>
              <y type="double"> 2.0 </y>
            </beta>
            <gamma type="char">Tea with milk and sugar.</gamma>
           </root>'

>> v1 = xml_parse( xstr )
>> v2 = xml_parse( xstr, 'off' )
>> X.delta = 10^(-4);
>> v3 = xml_parse( xstr, 'on', X )
>> v4 = xml_parse( xstr, 'off', X )
```

```
xml_save( filename, var, [att_switch] )
```

```
var = xml_load( filename )
```

`xml_save` and `xml_load` are wrapper functions for the commands `xml_format` and `xml_parse` respectively. These (non-validating) functions work similar to the standard Matlab `save` and `load` commands and do not take the additional function arguments provided for the `xml_format` and `xml_parse` commands.

`xml_save` takes a `filename` and a Matlab variable/struct `var`, converts it into XML format and stores the XML string in file `filename`.

`xml_load` loads an XML string contained in file `filename`, and converts it into a Matlab variable which is returned in `var`.

`filename`	`xml_save`: Name of file which get written and will contain XML string representation of Matlab variable. `xml_load`: Name of file with XML content. If the file cannot be found and the filename does not contain the extension ".xml", `xml_load` will try to add ".xml" to the filename and locate the file again. If this is still unsuccessful, an error message is displayed.
`att_switch`	Optional. 'on' saves XML type attributes, 'off' does not write XML type attributes. (default: 'on')
`var`	Matlab variable, matrix or structure.

Example:

```
>> v = pi;
>> xml_save( 'sample.xml', v )
>> v_copy = xml_load( 'sample.xml' )
>> v_copy - v   % should be zero! :-)
```

Supported Matlab Data Types

Currently implemented are the conversions of the following Matlab types in any valid combination and level of nesting:

`double`	contains numbers of type integer, double, long, etc.
`char`	contains alphanumerical entities (characters, strings, etc.)
`struct`	contains children of any of these types
`cell`	contains cell elements which can be of any of these types
`sparse`	contains sparse matrices of type double or complex. (3 arrays: i,j,k)
`complex`	contains complex numbers (2 arrays of type double [a]+i[b])
`logical`	contains logical states, represented by 0 and 1; stored as type boolean

In addition, `xml_parse` will also understand the following types:

`string`	entry gets interpreted in the same way as Matlab type **`char`**
`integer`, `float`, `numeric`	entry gets converted into Matlab type **`double`**
`boolean`	entry gets converted into Matlab type **`logical`**

Unsupported Matlab variable types are:

- 8-bit, 16-bit, 32-bit signed and unsigned integers (`int8`, `int16`, `int32`, `uint8`, etc.)
- single precision floating point
- function handles
- Java class objects

Most of the unsupported data types can be represented by supported ones, for example `logical`, `single precision` numbers and `integers` can be represented through the type `double`, and function handles may be represented by strings (see also Matlab function `func2str`). There is no need at present to implement the `int` or `uint` type as Matlab does not even provide simple arithmetic operations (such as +, -, *) for these (although some might argue that images are stored in `uint` format).

XML formatting with the `att_switch` parameter

This section describes the format of the XML string into which a Matlab variable gets converted when the function parameter `att_switch` is used in `xml_format` and `xml_parse`.

With `att_switch` parameter set to ‘on’ (default)

The default setting of the `att_switch` parameter is ‘on’ if not explicitly set to 'off'.
With the 'on' setting, `xml_format` writes metadata about data into attributes of the XML entry. `xml_parse` tries by default (with the `att_switch` set to 'on') to interpret the attributes in the headers of the XML entries and to apply these to the data so that Matlab data types can be reconstructed (see section "Supported Matlab Data Types").

Example:

```
>> sample.alpha = 3.14159265358979;
>> sample.beta = [1 2 3 4 5];
>> sample.gamma = 'fascinating'
>> str = xml_format(sample)    or    xml_format(sample, 'on')

str =

<root xml_tb_version="2.0" idx="1" type="struct" size="1 1">
  <alpha idx="1" type="double" size="1 1">3.14159265358979</alpha>
  <beta idx="1" type="double" size="1 5">1 2 3 4 5</beta>
  <gamma idx="1" type="char" size="1 11">fascinating</gamma>
</root>
```

The all-enclosing parent element is called `root`. It contains as attributes the XML Toolbox version number `xml_tb_version`, an index number `idx` and a type definition `type` as well as the size of the root element in `size`. The format of the XML string is described in section "Element Definition".

With att_switch parameter set to 'off'

With the `att_switch` parameter set to 'off', we obtain from `xml_format` an XML string without any attributes carrying metadata about the content of the elements. This causes `xml_parse`, if subsequently applied to `str`, to interpret all contents as strings rather than numbers.

If parameter `att_switch` is set to 'off' in the function call of `xml_parse`, the parser will read and interpret all contents as strings, even if there are attributes defined.

Examples:

```
>> sample.alpha = 3.14159265358979;
>> sample.beta = [1 2 3 4 5];
>> sample.gamma = 'fascinating'
>> str = xml_format(sample, 'off')

str =

<root>
  <alpha>3.14159265358979</alpha>
  <beta>1 2 3 4 5</beta>
  <gamma>fascinating</gamma>
</root>

>> v = xml_parse(str)     % no attributes in str → all contents are interpreted as strings

v = alpha: '3.14159265358979'
     beta: '1 2 3 4 5'
    gamma: 'fascinating'
```

XML Element Definition

The general XML element definition is as follows:

<*name* *attributes*> *content* </*name*>

or for empty elements:

<*name* *attributes* />

More specific:

```
<name  [xml_tb_version="x.y"]
 type="{double | char | struct | cell | sparse | complex | logical}"
 idx="N"
 [size="N N [N…]" ("1 1")]
 [name="A"] >
content
</name>
```

Special Element Names

(a) Names starting with `<!`
These elements contain comments or execute statements and are ignored.

(b) Names starting with `<?`
These elements contain execute statements or version information and are ignored.

(c) "`root`"
The enclosing, first-level parent gets the name "root" by default. This can be overwritten.

Example:

```
>> a.b = 111;
>> xml_format(a, 'on', 'dad')

  <dad xml_tb_version="2.0" idx="1" type="struct" size="1 1">
     <b idx="1" type="double" size="1 1">111</b>
  </dad>
```

(d) "`item`"
The element name "`item`" is reserved for children of the Matlab `cell` datatype.

Example:

```
>> a = {1, 2, 'a'}     % a is a cell array
>> xml_format(a)

  <root xml_tb_version="2.0" idx="1" type="cell" size="1 3">
    <item idx="1" type="double" size="1 1">1</item>
    <item idx="2" type="double" size="1 1">2</item>
    <item idx="3" type="char" size="1 1">a</item>
  </root>
```

Attributes

(a) `xml_tb_version`
- Only present in the root element. Indicates version of XML Toolbox for Matlab.
- If not present, XML Toolbox Version 1.x is assumed to have created this XML string, unless att_switch is set to "off"; in this case, most name-based XML strings can be read.

(b) `idx`
- Indicates the index of the element in its parent vector. This allows storage of vectors of the supported Matlab data types and also cell content in the correct order.
- If several elements with the same name exist, but `idx` is not given, `xml_parse` tries to assign the last read element to the last empty element in the vector, e.g.

```
<root> <a>1.1</a> <a>2.2</a> <b>0.0</b> <a>3.3</a> </root>
```

results – when converted with `v=xml_parse(str)` – in a Matlab structure `v`:

```
v(1).a=1.1  v(2).a=2.2  v(3).a=3.3
v(1).b=0.0  v(2).b=[]   v(3).b=[]
```

(c) `type`

- Represents the data type contained in the element, such as `double`, `char`, `struct`, `cell`, etc. For more details see section Matlab Data Types.

(d) `size`

- This attribute represent the size of the variable, struct or matrix in Matlab. The default value is `size="1 1"`. Important for `xml_parse()` for reshaping vectors and matrices from XML vector format into their original size. The minimum dimension of `size` is 2, higher dimensions have more entries, for example `size="2 7 3 5"` for a 4-dimensional matrix of size 2x7x3x5.

(e) `name`

- Not required from Version 2.0 upwards as the XML element tag is used as the variable name. If present in XML format of Version <=2.0, this attribute is used as name for the element.

(f) other attributes

- Support for additional attributes will be added in XML Toolbox Version 3.0.

Matlab Data Types

The following most common Matlab data types can be declared within the `type` attribute.

`double`	contains numbers of type integer, double, long, etc.
`char`	contains alphanumerical entities (characters, strings, etc.)
`struct`	contains children of any of these types
`cell`	contains cell elements which can be of any of these types
`sparse`	contains sparse matrices of type double or complex. (3 arrays: i,j,k)
`complex`	contains complex numbers (2 arrays of type double [a]+i[b])
`logical`	contains logical states, represented by 0 and 1; stored as type boolean

In addition, `xml_parse` will also understand the following types:

`string`	entry gets interpreted in the same way as Matlab type `char`
`integer, float, numeric`	entry gets converted into Matlab type `double`
`boolean`	entry gets converted into Matlab type `logical`

Changes from Version 1.x:

- The root element is now called 'root' by default. This can be overwritten by the user when converting variables to XML with `xml_format`.
- Additional attribute `xml_tb_version="2.0"` in root element. `xml_format` now produces only XML according to version 2.0 which will be upwards compatible.
- `idx="1"` as attribute in root element compared to `idx="0"` in Version 1.x.
- Type `double` gets now stored with all valid decimals rather than only the first six decimals. (Thanks to Luc Van Immerseel and Jasmin Wason for spotting this). Already implemented from Version 1.1 onwards.
- Biggest change: XML is stored with name tags instead of type tags.

5 EXAMPLES

This section contains a number of examples of how to use the XML Toolbox to convert Matlab variables to XML and also of how to convert general XML strings to Matlab variables using the `xml_format` and `xml_parse` commands (which get called by `xml_save` and `xml_load`).

Example 1: xml_format and strings

```
>> str = xml_format( 'Hello World!' )

    <root xml_tb_version="2.0" idx="1" type="char" size="1 12">Hello World!</root>
```

```
>> str = xml_format( 'Hello World!', 'off' )

    <root>Hello World!</root>
```

```
>> str = xml_format( 'Hello World!', 'off', 'UNIVERSE' )

    <UNIVERSE>Hello World!</UNIVERSE>
```

```
>> str = xml_format( 'Hello World!', '', 'UNIVERSE' )
>> str = xml_format( 'Hello World!', [], 'UNIVERSE' )

    <UNIVERSE xml_tb_version="2.0" idx="1" type="char" size="1 12">Hello World!</UNIVERSE>
```

Example 2: xml_format and doubles

```
>> str = xml_format([])                          % empty double

    <root xml_tb_version="2.0" idx="1" type="double" size="0 0"/>
```

```
>> str = xml_format(pi)

    <root xml_tb_version="2.0" idx="1" type="double" size="1 1">3.141592653589793</root>
```

```
>> str = xml_format([10 20 30 40])              % vector

    <root xml_tb_version="2.0" idx="1" type="double" size="1 4">10 20 30 40</root>
```

```
>> str = xml_format([10 20 30 40]')          % transposed vector

    <root xml_tb_version="2.0" idx="1" type="double" size="4 1">10 20 30 40</root>
```

```
>> str = xml_format([], 'off')               % empty double, attributes off

    <root/>
```

```
>> str = xml_format(pi, 'off')               % attributes off

    <root>3.141592653589793</root>
```

```
>> str = xml_format(pi, 'off', 'alpha')      % attributes off, root='alpha'

    <alpha>3.141592653589793</alpha>
```

```
>> str = xml_format([10 20 30 40], 'off')    % attributes off

    <root>10 20 30 40</root>
```

```
>> str = xml_format([10 20 30 40]', 'off')   % transposed, attributes off

    <root>10 20 30 40</root>
```

Example 3: xml_format and structs

```
>> v.user = 'mm';
>> v.date = datestr(now);
>> v.project = 'myProject';
>> v.ID   = 123456789;

>> str = xml_format( v )                     % struct

    <root xml_tb_version="2.0" idx="1" type="struct" size="1 1">
      <user idx="1" type="char" size="1 2">mm</user>
      <date idx="1" type="char" size="1 20">14-Nov-2003 10:33:18</date>
      <project idx="1" type="char" size="1 9">myProject</project>
      <ID idx="1" type="double" size="1 1">123456789</ID>
    </root>

>> str = xml_format( v, 'off' )              % struct, attributes off

    <root>
      <user>mm</user>
      <date>14-Nov-2003 10:33:18</date>
      <project>myProject</project>
      <ID>123456789</ID>
    </root>
```

Example 4: xml_format and cells

```
>> str = xml_format( {} )                    % empty cell

    <root xml_tb_version="2.0" idx="1" type="cell" size="0 0"/>
```

```
>> str = xml_format( {'aaa', 'bb', 'c'} )    % cell with chars

    <root xml_tb_version="2.0" idx="1" type="cell" size="1 3">
      <item idx="1" type="char" size="1 3">aaa</item>
      <item idx="2" type="char" size="1 2">bb</item>
      <item idx="3" type="char" size="1 1">c</item>
    </root>
```

```
>> str = xml_format( {1, 2, 3} )             % cell with doubles

    <root xml_tb_version="2.0" idx="1" type="cell" size="1 3">
      <item idx="1" type="double" size="1 1">1</item>
      <item idx="2" type="double" size="1 1">2</item>
      <item idx="3" type="double" size="1 1">3</item>
    </root>
```

```
>> str = xml_format( {'a', 1, 'b', 2} )      % cell with mixed content

    <root xml_tb_version="2.0" idx="1" type="cell" size="1 4">
      <item idx="1" type="char" size="1 1">a</item>
      <item idx="2" type="double" size="1 1">1</item>
      <item idx="3" type="char" size="1 1">b</item>
      <item idx="4" type="double" size="1 1">2</item>
    </root>
```

```
>> v.project = 'mypro'; v.alpha = 0; v.beta = 999;
>> str = xml_format( v )                     % cell with struct content

    <root xml_tb_version="2.0" idx="1" type="struct" size="1 1">
      <project idx="1" type="char" size="1 5">mypro</project>
      <alpha idx="1" type="double" size="1 1">0</alpha>
      <beta idx="1" type="double" size="1 1">999</beta>
    </root>
```

Example 5: xml_format of sparse and complex data

```
>> S = sparse(10,10);                        % sparse matrix 10x10
>> S(3,4) = 1; S(5,5) = 42;                  % with 2 entries
>> str = xml_format( S )

    <root xml_tb_version="2.0" idx="1" type="sparse" size="10 10">
      <item type="double" idx="1" size="2 1">3 5</item>
      <item type="double" idx="2" size="2 1">4 5</item>
      <item type="double" idx="3" size="2 1">1 42</item>
    </root>

>> str = xml_format( S, 'off' )              % sparse matrix, attributes off

    <root>
      <item>3 5</item>
      <item>4 5</item>
      <item>1 42</item>
    </root>
```

```
>> str = xml_format( 2+i )                    % complex

    <root xml_tb_version="2.0" idx="1" type="complex" size="1 1">
      <item type="double" idx="1" size="1 1">2</item>
      <item type="double" idx="2" size="1 1">1</item>
    </root>
```

```
>> str = xml_format( [i, 1+i, 2+2i] )         % vector of complex

    <root xml_tb_version="2.0" idx="1" type="complex" size="1 3">
      <item type="double" idx="1" size="1 3">0 1 2</item>
      <item type="double" idx="2" size="1 3">1 1 2</item>
    </root>
```

```
>> S = sparse( 10, 10, 3.14 + 15i );          % sparse with complex content
>> str = xml_format( S )

    <root xml_tb_version="2.0" idx="1" type="sparse" size="10 10">
      <item type="double" idx="1" size="1 1">10</item>
      <item type="double" idx="2" size="1 1">10</item>
      <item type="complex" idx="3" size="1 1">
        <item type="double" idx="1" size="1 1">3.14</item>
        <item type="double" idx="2" size="1 1">15</item>
      </item>
    </root>

>> str = xml_format( S, 'off' )               % sparse complex, no attributes

    <root>
      <item>10</item>
      <item>10</item>
      <item>
        <item>3.14</item>
        <item>15</item>
      </item>
    </root>
```

Example 6: xml_format with combinations of data types

```
>> A(1,1).a = [1 2 3 4]';
>> A(1,1).b = {'aaa', [123], 'bbb', 'ccc', [456]}

   A = a: [4x1 double]
       b: {'aaa'  [123]  'bbb'  'ccc'  [456]}

>> A(1,1).c = 'This is a string';
>> A(1,1).d(2,2).e = 'This is really great!';
>> A(1,1).d(2,2).f = sparse(5,7,1);
>> A(1,1).BOOL = (1==2);

>> A(2,2) = A(1,1)

    A = 2x2 struct array with fields:
            a   b   c   d  BOOL
```

```
>> str = xml_format( A )                    % data type mix

    <root xml_tb_version="2.0" idx="1" type="struct" size="2 2">

      <a idx="1" type="double" size="4 1">1 2 3 4</a>
      <b idx="1" type="cell" size="1 5">
        <item idx="1" type="char" size="1 3">aaa</item>
        <item idx="2" type="double" size="1 1">123</item>
        <item idx="3" type="char" size="1 3">bbb</item>
        <item idx="4" type="char" size="1 3">ccc</item>
        <item idx="5" type="double" size="1 1">456</item>
      </b>
      <c idx="1" type="char" size="1 16">This is a string</c>
      <d idx="1" type="struct" size="2 2">
        <e idx="1" type="double" size="0 0"/>
        <f idx="1" type="double" size="0 0"/>
        <e idx="2" type="double" size="0 0"/>
        <f idx="2" type="double" size="0 0"/>
        <e idx="3" type="double" size="0 0"/>
        <f idx="3" type="double" size="0 0"/>
        <e idx="4" type="char" size="1 21">This is really great!</e>
        <f idx="4" type="sparse" size="5 7">
          <item type="double" idx="1" size="1 1">5</item>
          <item type="double" idx="2" size="1 1">7</item>
          <item type="double" idx="3" size="1 1">1</item>
        </f>
      </d>
      <BOOL idx="1" type="boolean" size="1 1">0</BOOL>

      <a idx="2" type="double" size="0 0"/>
      <b idx="2" type="double" size="0 0"/>
      <c idx="2" type="double" size="0 0"/>
      <d idx="2" type="double" size="0 0"/>
      <BOOL idx="2" type="double" size="0 0"/>

      <a idx="3" type="double" size="0 0"/>
      <b idx="3" type="double" size="0 0"/>
      <c idx="3" type="double" size="0 0"/>
      <d idx="3" type="double" size="0 0"/>
      <BOOL idx="3" type="double" size="0 0"/>

      <a idx="4" type="double" size="4 1">1 2 3 4</a>
      <b idx="4" type="cell" size="1 5">
        <item idx="1" type="char" size="1 3">aaa</item>
        <item idx="2" type="double" size="1 1">123</item>
        <item idx="3" type="char" size="1 3">bbb</item>
        <item idx="4" type="char" size="1 3">ccc</item>
        <item idx="5" type="double" size="1 1">456</item>
      </b>
      <c idx="4" type="char" size="1 16">This is a string</c>
      <d idx="4" type="struct" size="2 2">
        <e idx="1" type="double" size="0 0"/>
        <f idx="1" type="double" size="0 0"/>
        <e idx="2" type="double" size="0 0"/>
        <f idx="2" type="double" size="0 0"/>
        <e idx="3" type="double" size="0 0"/>
        <f idx="3" type="double" size="0 0"/>
        <e idx="4" type="char" size="1 21">This is really great!</e>
        <f idx="4" type="sparse" size="5 7">
          <item type="double" idx="1" size="1 1">5</item>
          <item type="double" idx="2" size="1 1">7</item>
          <item type="double" idx="3" size="1 1">1</item>
        </f>
      </d>
      <BOOL idx="4" type="boolean" size="1 1">0</BOOL>

    </root>
```

```
>> str = xml_format( A, 'off' )                % data type mix, attributes off

    <root>

      <a>1 2 3 4</a>
      <b>
        <item>aaa</item>
        <item>123</item>
        <item>bbb</item>
        <item>ccc</item>
        <item>456</item>
      </b>
      <c>This is a string</c>
      <d>
        <e/>
        <f/>
        <e/>
        <f/>
        <e/>
        <f/>
        <e>This is really great!</e>
        <f>
          <item>5</item>
          <item>7</item>
          <item>1</item>
        </f>
      </d>
      <BOOL>0</BOOL>

      <a/>
      <b/>
      <c/>
      <d/>
      <BOOL/>

      <a/>
      <b/>
      <c/>
      <d/>
      <BOOL/>

      <a>1 2 3 4</a>
      <b>
        <item>aaa</item>
        <item>123</item>
        <item>bbb</item>
        <item>ccc</item>
        <item>456</item>
      </b>
      <c>This is a string</c>
      <d>
        <e/>
        <f/>
        <e/>
        <f/>
        <e/>
        <f/>
        <e>This is really great!</e>
        <f>
          <item>5</item>
          <item>7</item>
          <item>1</item>
        </f>
      </d>
      <BOOL>0</BOOL>

    </root>
```

Example 7: xml_parse

```
>> str = '<root>hello world!</root>';
>> v = xml_parse(str)

    v = hello world!            % string
```

```
>> str = '<root>3.1415</root>';                  % number without attribute
>> v = xml_parse(str)

    v = 3.1415                  % string
```

```
>> str = '<root type="double">3.1415</root>';   % number with attribute
>> v = xml_parse(str)

    v = 3.1415                  % double
```

```
>> str = '<root type="double">3.1415</root>';   % number with attribute
>> v = xml_parse(str, 'off')                    % ignore attribute

    v = 3.1415                  % string
```

```
>> str = fileread( 'test.xml' )

    <root>
      <project>MyProject</project>
      <description>This is a test data structure</description>
      <metadata>
        <user>mm</user>
        <date>20-Oct-2003 12:14:52</date>
        <var>
          <name>alpha</name>
          <matrix>1 2 3 4 5 6</matrix>
        </var>
      </metadata>
    </root>

>> v = xml_parse( str )

     project: 'MyProject'
  description: 'This is a test data structure'
     metadata: [1x1 struct]

>> v.metadata

         user: 'mm'
         date: '20-Oct-2003 12:14:52'
          var: [1x1 struct]

>> v.metadata.var

         name: 'alpha'
       matrix: '1 2 3 4 5 6'
```

```
>> str = fileread( 'test_with_attributes.xml' )

    <root xml_tb_version="2.0" idx="1" type="struct" size="1 1">
      <project idx="1" type="char" size="1 9">MyProject</project>
      <description idx="1" type="char" size="1 29">This is a test data
        structure</description>
      <metadata idx="1" type="struct" size="1 1">
```

```
        <user idx="1" type="char" size="1 2">mm</user>
        <date idx="1" type="char" size="1 20">20-Oct-2003 12:14:52</date>
        <var idx="1" type="struct" size="1 1">
          <name idx="1" type="char" size="1 5">alpha</name>
          <matrix idx="1" type="double" size="3 2">1 2 3 4 5 6</matrix>
        </var>
      </metadata>
    </root>

>> v = xml_parse( str )

     project: 'MyProject'
  description: 'This is a test data structure'
     metadata: [1x1 struct]

>> v.metadata

         user: 'mm'
         date: '20-Oct-2003 12:14:52'
          var: [1x1 struct]

>> v.metadata.var

         name: 'alpha'
       matrix: [3x2 double]            % note: matrix has now shape and type.
```

6 QUESTIONS & ANSWERS

Q: What about the use of namespaces in the XML string?

A: Namespaces get currently ignored by the parser. This is because the XML element tags get used as variable/ struct names which have certain restrictions on them, e.g. a variable in Matlab cannot be called `gem:projectID`. (the colon, ':', is not allowed, hence "`gem:`" gets stripped from the entry and the variable name will be "`projectID`").
XML Toolbox Version 3.0 will have a feature which allows the use of name spaces in structs.

Q: Why is there not an XML Schema available in Version 2.0 as for Version 1.x?

A: As the Toolbox can read almost any XML and is based on a non-validating parser, a Schema can not easily be applied. However, the Schema for Version 1.x and before can be found in the appendices as the current parser still understands the "old" V 1.x format.

Q: What happens to leading & trailing spaces in strings when converted into XML?

A: The XML Toolbox follows the XML standard: leading and trailing spaces in content strings are preserved as in the original. Many XML processing systems (databases, etc), however, do not adhere to the XML standard in this respect and if you have problems with retrieving the same strings as you store, then is probably one of those systems involved in the data flow. The `xml_parse` command will throw an error if a string does not contain as many characters as indicated in the size attribute.

7 APPENDIX: XML SCHEMA (V.1.x)

The XML Schema language provides a means for defining the structure, content and semantics of a set of XML documents. An XML Schema can be used to check if an XML document is valid with respect to certain restrictions and can also be used as a contract for exchanging data with another party.

The XML Schema matlab.xsd contains the schema for validation of XML files produced by the Versions 1.0 and 1.1 of the XML Toolbox for Matlab. The namespace for the schema is *http://www.geodise.org/matlab.xsd*. (The purpose of the namespace is to give the schema a unique name. When this document is released the schema will be available at this URL).

Strings or files produced with the old (V.1.x) `xml_format` command contain this namespace in the xlmns attribute of the root element. These strings or files should all conform to the matlab.xsd schema. The current `xml_parse` function is able to parse any XML document that conforms to the Version 1.x XML Schema described in matlab.xsd.

```
<xs:element name="struct">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="struct"/>
      <xs:element ref="double"/>
      <xs:element ref="char"/>
      <xs:element ref="sparse"/>
      <xs:element ref="cell"/>
      <xs:element ref="complex"/>
    </xs:choice>
    <xs:attributeGroup ref="structAttributes"/>
  </xs:complexType>
</xs:element>
```

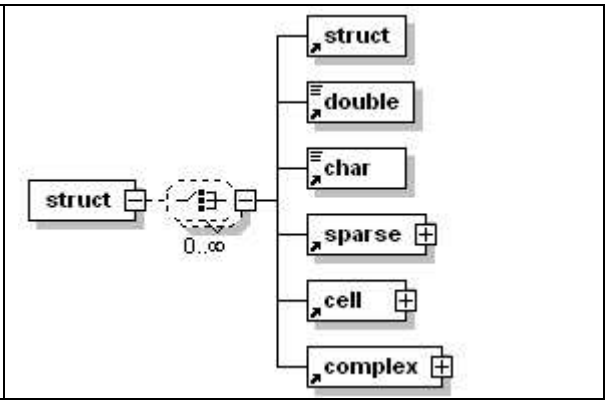

Figure 1 - struct element definition from matlab.xsd in xml format and as a diagram (produced using XML Spy).

As an example, Figure 1 shows the part of matlab.xsd that defines the `struct` element, shown in its original XML format and as a diagram. This specifies that the struct element must contain zero or more struct, double, char, sparse, cell or complex elements, in any order. It should also contain certain attributes, and these are specified elsewhere in the schema, e.g. name and idx.