The XML Toolbox

A User's Guide

Geodise

Release:	XML Toolbox V.3.2.1 standalone
Version:	xml_toolbox 3.2.1
Title:	The XML Toolbox – A User's Guide
Authors:	Dr Marc Molinari, m.molinari@soton.ac.uk
PI:	Prof Simon Cox, s.j.cox@soton.ac.uk
Web:	http://www.geodise.org
Note:	This is an extract from "The Geodise User Manual – A User's Guide"
Copyright:	Copyright © 2007, The Geodise Project, University of Southampton

Acknowledgement:

The development of the Geodise toolboxes for public release has been supported by the managed programme of the Open Middleware Infrastructure Institute (http://www.omii.ac.uk/).

Contents

Introduction from "The Geodise Toolboxes"

The Geodise Toolboxes provide a collection of functions that extend the capability of the Matlab®[1] technical computing environment. The Geodise Compute, Database and XML toolboxes contain routines that facilitate many aspects of Grid computing and data management including:

- The submission and management of computational job on remote compute resources via the Globus GRAM service.
- File transfer and remote directory management using the GridFTP protocol.
- Single sign-on to the Grid with Globus proxy certificates.
- Storage and grouping of files and variables, annotated with user defined metadata, in an archive.
- Graphical and programmatic interfaces for querying the metadata to easily locate files and variables.

[1] Matlab® is a registered trademark of The Mathworks, Inc, http://www.mathworks.com

- Sharing and reuse of data among distributed users. Users may grant access to their data to other members of a *Virtual Organisation*.
- Conversion of Matlab structures and variables into a non-proprietary, plain text format (XML) which can be stored and used by other tools.
- Conversion of almost any type of XML document including WSDL descriptions of Web Services into Matlab's struct format or cell data type.

Grid computing provides the infrastructure for the collaborative use of computers, networks, data, storage and applications across distributed organisations. A computational job can be run on the Grid to make use of resources unavailable on the user’s desktop, for example to exploit software licenses or greater computational power. The Geodise Compute Toolbox provides Matlab functions for submitting and monitoring jobs on the Grid, transferring files to and from remote compute resources, and managing the certificates used to identify users and authorise use of the resources.

Compute intensive applications often use and produce many data files and data structures. It can become difficult to find, reuse and share data from various applications that have been run repeatedly with different parameters. The Geodise Database Toolbox can be used to store additional user-defined information (called metadata) describing files and Matlab variables, so that they can be located and retrieved more easily with metadata queries. Files and variables can also be grouped together, and data can be shared with other users by granting access permissions.

XML is a flexible standard data format that is widely used to structure and store information, and to exchange data between various computer applications. The XML Toolbox functions convert and store Matlab variables and structures from the internal format into XML and vice versa. This allows parameter structures, variables and results from computational applications to be stored in a non-proprietary file format, or in XML-capable databases, and can be used to transfer Matlab data across the Grid. Comparing arbitrary Matlab structures was not previously possible, but this can be now achieved by comparing their XML representation.

This user guide is an extract of the full Geodise manual and introduces the reader to the XML toolbox giving an overview of the functionality provided. The function reference contains detailed information about the syntax its functions.

Introduction

The XML Toolbox for Matlab allows users to convert and store variables and structures from the Matlab workspace into the plain text XML format, and vice versa. This XML format can be used to store parameter structures, variables and results from engineering applications in non-proprietary files, or XML-capable databases, and can be used for the transfer of data across the Grid. The toolbox contains bi-directional conversion routines implemented as four small intuitive and easy-to-use Matlab functions. As an additional feature, this toolbox allows the comparison of internal Matlab structures by comparing their XML representation, which was not previously possible.

- Almost any type of XML document can be read and converted into Matlab's struct format or cell data type.
- Matlab structures and variables can be stored in a non-proprietary format and used by other tools.
- XML representations can be stored and queried using the functions provided by the Geodise Database Toolbox.
- The ability to leverage XML and database technologies makes the data available beyond the Matlab environment, and facilitates data sharing and reuse between users.
- Access to XML data-driven tools such as Web Services becomes more transparent to engineering users.

The following definitions are valid for XML Toolbox Version ≥ 2.0 (2.0, 2.1, 2.2, 3.0a, 3.1, 3.2). The size of data structures the XML Toolbox can deal with is only limited by the available memory; as an indication, 60MB large data structures can be easily converted on a 256MB PC running Matlab.

`xml_format`	Converts Matlab data to an XML string
`xml_formatany`	Converts Matlab data to an XML string with user-defined attributes
`xml_parse`	Converts an XML string into Matlab data
`xml_parseany`	Converts an XML string with attributes into Matlab data
`xml_load`	Loads an XML file and returns Matlab data
`xml_save`	Saves Matlab data into an XML file
`xml_help`	Displays help for each xml_ function

Table 1 XML Toolbox functions

Supported Versions

This version of the XML Toolbox is supported in and has been tested with the following Versions of the Matlab scripting environment:

Matlab

Version 6.5 (R13)	Windows + Linux
Version 7.0 (R14)	Windows + Linux
Version 7.0.1 (R14)	Windows + Linux
Version 7.0.4 (R14) SP2	Windows + Linux
Version 7.1.0 (R14) SP3	Windows + Linux
Version 7.2.0 (R2006a)	Windows + Linux
Version 7.3.0 (R2006b)	Windows + Linux

An additional package is available for the Jython scripting environment:

Python

Jython Version 2.1	Windows + Linux

The p-code for Matlab Releases 13 and 14 differs so that there are differing zip-packages with the relevant functions for each release.

Tutorial

The XML Toolbox for Matlab can be used independently of the Compute and Database Toolboxes. No proxy certificate is required to make use of its functionality.

Converting Matlab data types to XML

All common Matlab data types can be converted into XML with the simple-to-use commands `xml_format` (with or without attributes) or `xml_formatany`. We highlight the differences in XML output structure in the following three examples.

```
>> v.a = 1.2345
>> v.b = [1 2 3 4; 5 6 7 8]
>> v.c = 'This is a string.'
>> v.d = {'alpha', 'beta'}
>> v.e = (1==2)
>> v.f.sub1.subsub1 = 1
>> v.f.sub1.subsub2 = 2
>> v.g(1).aa(1) = {'g1aa1'}
>> v.g(1).aa(2) = {'g1aa2'}
>> v.g(2).aa(1) = {'g2aa1'}
```

This first example shows the formatting of the Matlab variable with no additional input parameters specified. The XML is formatted in such a way that any subsequent parsing of the created XML string with `xml_parse` reconstructs an exact copy of the original Matlab variable.

```
>> xmlstr = xml_format(v)
```

```
xmlstr =

<root xml_tb_version="3.2.0" idx="1" type="struct" size="1 1">
  <a idx="1" type="double" size="1 1">1.2345</a>
  <b idx="1" type="double" size="2 4">1 5 2 6 3 7 4 8</b>
  <c idx="1" type="char" size="1 17">This is a string.</c>
  <d idx="1" type="cell" size="1 2">
    <item idx="1" type="char" size="1 5">alpha</item>
    <item idx="2" type="char" size="1 4">beta</item>
  </d>
  <e idx="1" type="boolean" size="1 1">0</e>
  <f idx="1" type="struct" size="1 1">
    <sub1 idx="1" type="struct" size="1 1">
      <subsub1 idx="1" type="double" size="1 1">1</subsub1>
      <subsub2 idx="1" type="double" size="1 1">2</subsub2>
    </sub1>
  </f>
  <g idx="1" type="struct" size="1 2">
    <aa idx="1" type="cell" size="1 2">
      <item idx="1" type="char" size="1 5">g1aa1</item>
      <item idx="2" type="char" size="1 5">g1aa2</item>
    </aa>
    <aa idx="2" type="cell" size="1 1">
      <item idx="1" type="char" size="1 5">g2aa1</item>
    </aa>
  </g>
</root>
```

The Matlab-specific attributes `idx`, `type` and `size`, which allow the exact reconstruction of the Matlab data types, can be turned off by specifying the second parameter in the `xml_format` function call as 'off'. This results in a more generic formatting of the structure, however, the XML contents are now interpreted purely as strings when parsed back into Matlab as type and size information are lost:

```
>> xmlstr = xml_format(v, 'off')
```

```
xmlstr =

<root>
  <a>1.2345</a>
  <b>1 5 2 6 3 7 4 8</b>
  <c>This is a string.</c>
  <d>
    <item>alpha</item>
    <item>beta</item>
  </d>
  <e>0</e>
  <f>
    <sub1>
      <subsub1>1</subsub1>
      <subsub2>2</subsub2>
    </sub1>
  </f>
  <g>
    <aa>
      <item>g1aa1</item>
      <item>g1aa2</item>
    </aa>
    <aa>
      <item>g2aa1</item>
    </aa>
  </g>
</root>
```

The user can write the XML representation of a Matlab variable immediately into a XML file using the command `xml_save`. This command uses the same XML format as the function `xml_format`.

If the user wishes to define XML attributes other than the default `idx`, `type` and `size` parameters, these can be added using a substructure called 'ATTRIBUTE' in the Matlab structure and performing the formatting with the command `xml_formatany`. This command converts Matlab cell data vectors into several XML elements with the same name tag without using the 'item' tag as in the previous example.
`xml_formatany` may be preferable to xml_format when converting Matlab data into XML which is processed in other applications, however, some of the information

about the original data types may be lost when converting the XML back into Matlab using `xml_parseany`:

```
>> xmlstr = xml_formatany(v)
```

```
xmlstr =

<root>
  <a>1.2345</a>
  <b>1 5 2 6 3 7 4 8</b>
  <c>This is a string.</c>
  <d>alpha</d>
  <d>beta</d>
  <e>0</e>
  <f>
    <sub1>
      <subsub1>1</subsub1>
      <subsub2>2</subsub2>
    </sub1>
  </f>
  <g>
    <aa>g1aa1</aa>
    <aa>g1aa2</aa>
  </g>
  <g>
    <aa>g2aa1</aa>
  </g>
</root>
```

We can specify additional attributes for the subfields `f.sub1` and `g(2)`

```
>> v.f.sub1.ATTRIBUTE.fontname = 'Helvetica'
>> v.g(2).ATTRIUTE.fontname = 'Helvetica2'
```

which then results in the following XML string:

```
>> xmlstr = xml_formatany(v)
```

```
xmlstr =

<root>
  [...]
  <f>
    <sub1 fontname="Helvetica">
      <subsub1>1</subsub1>
      <subsub2>2</subsub2>
    </sub1>
  </f>
  <g>
    <aa>g1aa1</aa>
    <aa>g1aa2</aa>
  </g>
  <g fontname="Helvetica2">
    <aa>g2aa1</aa>
  </g>
</root>
```

Converting XML to Matlab data types

As XML can contain any arbitrary contents as long as they follow the W3C XML Recommendation (www.w3.org), parsing and translating of these constructs into a Matlab-specific environment can be complex. The functions `xml_parse` and `xml_parseany` allow the conversion of XML strings into Matlab data structures in a sensible way.

There are three distinct ways of importing XML into Matlab data structures. These correspond to the techniques shown above for `xml_format` and `xml_formatany`. (There are actually four ways; however, we no longer support the old method from version 1.x).

If the XML contains Matlab specific descriptors, such as created by `xml_format` with attributes switched on (that means the `idx`, `type`, `size` attributes), the XML Toolbox will be able to re-create exactly the Matlab data type and content described by the XML string.

For example,

```
>> xmlstr = ...

<root xml_tb_version="3.2.0" idx="1" type="struct" size="1 1">
  <a idx="1" type="double" size="1 1">1.2345</a>
  <b idx="1" type="double" size="2 4">1 5 2 6 3 7 4 8</b>
  <c idx="1" type="char" size="1 17">This is a string.</c>
  <d idx="1" type="cell" size="1 2">
    <item idx="1" type="char" size="1 5">alpha</item>
    <item idx="2" type="char" size="1 4">beta</item>
  </d>
  <e idx="1" type="boolean" size="1 1">0</e>
  <f idx="1" type="struct" size="1 1">
    <sub1 idx="1" type="struct" size="1 1">
      <subsub1 idx="1" type="double" size="1 1">1</subsub1>
      <subsub2 idx="1" type="double" size="1 1">2</subsub2>
    </sub1>
  </f>
  <g idx="1" type="struct" size="1 2">
    <aa idx="1" type="cell" size="1 2">
      <item idx="1" type="char" size="1 5">g1aa1</item>
      <item idx="2" type="char" size="1 5">g1aa2</item>
    </aa>
    <aa idx="2" type="cell" size="1 1">
      <item idx="1" type="char" size="1 5">g2aa1</item>
    </aa>
  </g>
</root>
```

can be parsed using the command

```
>> v = xml_parse( xmlstr )
```

and returns the structure

```
v =
    a: 1.2345
    b: [2x4 double]
    c: 'This is a string.'
    d: {'alpha'  'beta'}
    e: 0
    f: [1x1 struct]
    g: [1x2 struct]
```

which corresponds exactly to the Matlab variable used in `xml_format` to create the XML string.

If we use the same command, `xml_parse`, but tell the parser to ignore the attributes with the command

```
>> v_wo_att = xml_parse( xmlstr, 'off' )
```

we obtain a structure where types and sizes of the data will not be adapted to match standard Matlab data types, that means that all alphanumeric content will be returned as strings.

```
v_wo_att =
    a: '1.2345'
    b: '1 5 2 6 3 7 4 8'
    c: 'This is a string.'
    d: {'alpha'  'beta'}
    e: '0'
    f: [1x1 struct]
    g: [1x2 struct]
```

The structural information (in fields `f` and `g`) is still preserved, although matrix contents, such as in field `b`, and numeric values, such as in fields `a` and `e`, are returned as pure strings.

The third possibility is to use `xml_parseany` which is able to convert most XML strings to Matlab data structures while taking care of namespaces and attributes. As the structure in XML strings can be very complex (for example in WSDL documents), the variable returned is a struct variable with sub-structures defined as cells.

If we parse, for example,

```
>> xmlstr = ...

  <gem:project name="MyProject">
    <username type="string">Me</username>
    <date_created type="date">2004-10-12</date_created>
    <description fontsize="10"> cool! </description>
    <parameters n="4">
      <eps1 type="dielectric" units="1"> 8.92 </eps1>
      <eps2 type="dielectric" units="1"> 1.00 </eps2>
      <StT  type="structuretype"> rod </StT>
      <nofEV> 47 </nofEV>
    </parameters>
  </project>
```

with

```
>> v = xml_parseany( xmlstr )
```

we obtain the variable

```
v =
       ATTRIBUTE: [1x1 struct]
        username: {[1x1 struct]}
    date_created: {[1x1 struct]}
     description: {[1x1 struct]}
      parameters: {[1x1 struct]}
```

with the following variable structure

```
v.ATTRIBUTE(1).name                          MyProject
v.ATTRIBUTE(1).NAMESPACE                     gem
v.username{1}.ATTRIBUTE.type                 string
v.username{1}.CONTENT                        Me
v.date_created{1}.ATTRIBUTE.type             date
v.date_created{1}.CONTENT                    2004-10-12
v.description{1}.ATTRIBUTE.fontsize          10
v.description{1}.CONTENT                      cool!
v.parameters{1}.eps1{1}.ATTRIBUTE.type       dielectric
v.parameters{1}.eps1{1}.ATTRIBUTE.units      1
v.parameters{1}.eps1{1}.CONTENT              8.92
v.parameters{1}.eps2{1}.ATTRIBUTE.type       dielectric
v.parameters{1}.eps2{1}.ATTRIBUTE.units      1
v.parameters{1}.eps2{1}.CONTENT              1.00
v.parameters{1}.StT{1}.ATTRIBUTE.type        structuretype
v.parameters{1}.StT{1}.CONTENT               rod
v.parameters{1}.nofEV{1}.ATTRIBUTE.type      numeric
v.parameters{1}.nofEV{1}.CONTENT             47
v.parameters{1}.ATTRIBUTE.n                  4
```

Function Reference

xml_format

Converts a Matlab variable into an XML string.

Syntax

```
    xmlstr = xml_format(v)
    xmlstr = xml_format(v,attswitch)
    xmlstr = xml_format(v,attswitch,name)
```

Description

`xml_format` converts Matlab variables and data structures (including deeply nested structures) into XML and returns the XML as string.

Input Arguments

`v`	Matlab variable of type "struct", "char", "double"(numeric), "complex", "sparse", "cell", or "logical"(boolean).
`attswitch`	optional, default='on': 'on' writes header attributes `idx`, `size`, `type` for identification by Matlab when parsing the XML later; 'off' writes "plain" XML without header attributes.
`name`	optional, give root element a specific name, eg. 'project'.

Output Arguments

`xmlstr`	string, containing XML description of the variable `v`.

The root element of the created XML string is called 'root' by default but this can be overwritten with the `name` input parameter. The default attribute `xml_tb_version` is added to the root element unless `attswitch` is set to 'off'.

If `attswitch` is left empty, [], or set to 'on', the default attributes `idx`, `type`, and `size` will be added to the XML element headers. This allows `xml_parse` to parse and convert the XML string correctly back into the original Matlab variable or data structure.

If `attswitch` is set to 'off', some of the information is lost and subsequently the contents of XML elements will be read in as strings when converting back using `xml_parse`.

Examples

This example shows how to convert a simple number into an XML string. Note that we could have used `xml_format(5)` instead.

```
v = 5;
xmlstr = xml_format(v)
```

```
xmlstr =
<root xml_tb_version="3.0" idx="1" type="double" size="1 1">
5</root>
```

We can tell the command to ignore all the attributes and obtain the following XML:

```
xmlstr = xml_format(v,'off')
```

```
xmlstr =
<root>5</root>
```

The root elements can be assigned a different name by adding this as third parameter to the `xml_format` function:

```
xmlstr = xml_format(v,'off','myXmlNumber')
```

```
xmlstr =
<myXmlNumber>5</myXmlNumber>
```

This example shows how pre-defined Matlab data (here pi) is translated into XML. The number of decimals stored is the number required to reconstruct the exact same variable in Matlab from XML with the `xml_parse` function.

```
v = pi;
xmlstr = xml_format(v,[],'pi')
xmlstr =
<pi xml_tb_version="3.0" idx="1" type="double" size="1 1">
3.141592653589793</pi>
```

Character arrays or strings can also be converted into XML:

```
v = 'The Hitchhikers Guide to the Galaxy';
xmlstr = xml_format(v);
```

```
xmlstr =
<root xml_tb_version="3.0" idx="1" type="char" size="1 35">
The Hitchhikers Guide to the Galaxy</root>
```

One of the most powerful ways to use the XML Toolbox is to convert whole data structures (with substructures) which can contain any Matlab data type.

```
v.project.name = 'my Project no. 001';
v.project.date = datestr(now,31);
v.project.uid  = '208d0174-a752-f391-faf2-45bc397';
v.comment = 'This is a new project';

xmlstr = xml_format(v,'off');
```

```
xmlstr =
<root>
  <project>
    <name>my Project no. 001</name>
    <date>2004-09-09 16:18:29</date>
    <uid>208d0174-a752-f391-faf2-45bc397</uid>
  </project>
  <comment>This is a new project</comment>
</root>
```

Notes

If different attributes are required in the output string, please see description for `xml_formatany`.

See also

xml_parseany, xml_formatany, xml_parse, xml_load, xml_save, xml_help

xml_formatany

Converts a Matlab variable into an XML string with user-defined attributes.

Syntax

```
xmlstr = xml_formatany(v)
xmlstr = xml_formatany(v,attswitch)
xmlstr = xml_formatany(v,attswitch,name)
```

Description

`xml_formatany` converts Matlab variables and structures (including deeply nested structures) into an XML string. The user can specify attributes for each XML element in substructures of the struct variable, `v`.

Input Arguments

`v`	Matlab variable of type "struct", "char", "double"(numeric), "complex", "sparse", "cell", or "logical"(boolean).
`attswitch`	optional, default='on': 'on' writes header attributes `idx`, `size`, `type` for identification by Matlab when parsing the XML later; 'off' writes "plain" XML without header attributes.
`name`	optional, give root element a specific name, eg. 'project'.

Output Arguments

`xmlstr`	string, containing XML description of the variable `v`.

The root element of the created XML string is called 'root' by default but this can be overwritten with the `name` input parameter. A default `xml_tb_version` attribute is added to the root element unless `attswitch` is set to 'off'.

If `attswitch` is left empty, [], or set to 'on', the default attributes idx, type, and size will be added to the XML element headers. This allows xml_parse to parse and convert the XML string correctly back into the original Matlab variable or data structure.

If `attswitch` is set to 'off', some of the information is lost and subsequently the contents of XML elements will be read in as strings when converting back using `xml_parse`.

Examples

In this example, we define a data structure in Matlab and add attributes to it before converting it into an XML string.

```
v.project.name = 'my Project no. 002';
v.project.date = datestr(now, 31);
v.project.uid  = '2004-0909-1618-29af-04c7';
v.project.ATTRIBUTE.id = 'AA5119278466';
v.comment.CONTENT = 'This is a new project';
v.comment.ATTRIBUTE.fontname = 'Times New Roman';

xmlstr = xml_formatany(v);
```

```
xmlstr =
<root>
  <project id="AA5119278466">
    <name>my Project no. 002</name>
    <date>2004-09-09 16:18:29</date>
    <uid>2004-0909-1618-29af-04c7</uid>
  </project>
  <comment fontname="Times New Roman">This is a new
    project</comment>
</root>
```

Notes

If attributes are required for string data, the string must be explicitly assigned to a CONTENT field of the Matlab structure. In the above example, the comment field is defined as

```
comment.ATTRIBUTE.fontname = 'Times New Roman'
comment.CONTENT = 'This is a new project';
```

This is due to the ATTRIBUTE field overwriting the contents otherwise.

See also

`xml_parseany`, `xml_format`, `xml_parse`, `xml_load`, `xml_save`, `xml_help`

xml_help

Shows a one-page summary of the usage for all XML Toolbox commands.

Syntax

```
xml_help
```

```
-------------------------
XML TOOLBOX FOR MATLAB X.Y
-------------------------

FUNCTIONS:
 xml_format converts a Matlab variable/structure into an XML string
 xml_parse  parses and converts an XML string into Matlab variable
 xml_save   saves a Matlab variable/structure in XML format in a file
 xml_load   loads an .xml file written with xml_save back into Matlab
 xml_help   this file, displays info about available xml_* commands

 tests/xml_tests    tests the xml toolbox by writing/reading a number
                    of xml test files

FILES:
 doc/xml_toolbox.*  documentation containing info on installation,
                    usage, implementation, etc.
 matlab.xsd         contains a Schema to validate XML files for the
                    toolbox (V.1.0) (if not present, look at
                    http://www.geodise.org/matlab.xsd)

RELATED:
  xmlread, xmlwrite (shipped with Matlab from version 6.5)

Further information can be obtained by using the help command on
a specific function, e.g. help xml_format.

-------------------------------------------------------------
  Copyright (C) 2002-2005
  Author: Marc Molinari <m.molinari@soton.ac.uk>
  $Revision$ $Date$
```

See also

xml_parseany, xml_formatany, xml_format, xml_parse, xml_load, xml_save

xml_load

Loads an XML file and converts its content into a Matlab structure or variable.

Syntax

```
v = xml_load(filename)
v = xml_load(filename,attswitch)
```

Description

`xml_load` reads the file given in parameter filename and uses `xml_parse` to convert it into a Matlab data structure or variable. If the file cannot be found, an error will be displayed.

Input Arguments

`filename`	filename of xml file to load (if extension .xml is omitted, `xml_load` tries to append it if the file cannot be found).
`attswitch`	optional, default='on': 'on' takes into account attributes `idx`, `size`, `type` for creating corresponding Matlab data types; 'off' ignores attributes in XML element headers.

Output Arguments

`v`	Matlab structure or variable.

Examples

This example simply loads the sample file from the given location and converts its contents to a Matlab data structure. (The file has previously been created using `xml_save`).

```
v = xml_load('c:/data/myfavourite.xml')
```

```
v =
       name: 'Google'
        url: 'http://www.google.com'
     rating: 5
description: 'Great search functionality for the web'
```

In the following example, we perform the same action, however, as we are specifying the additional parameter 'off' for attributes, the `idx`, `size`, and `type` attributes are ignored and the result is slightly different: `v.rating` in this case is returned as a Matlab string variable, `'5'`.

```
v = xml_load('c:/data/myfavourite.xml','off')
```

```
v =
       name: 'Google'
        url: 'http://www.google.com'
     rating: '5'
description: 'Great search functionality for the web'
```

See also

`xml_parseany`, `xml_formatany`, `xml_format`, `xml_parse`, `xml_save`, `xml_help`

xml_parse

Parses an XML string, xmlstr, and returns the corresponding Matlab structure v.

Syntax

```
v = xml_parse(xmlstr)
v = xml_parse(xmlstr,attswitch)
```

Description

This is a non-validating parser. XML processing entries or comments starting with '<?' or '<!', are ignored by the parser.

Input Arguments

xmlstr	XML string, for example read from a file with `xmlstr = fileread(filename)`
attswitch	optional, default='on': 'on' reads XML header attributes `idx`, `size`, `type` if present and interprets these to create the correct Matlab data types. 'off' ignores XML element header attributes and interprets contents as strings.

Output Arguments

v	Matlab variable or structure.

Examples

This example shows how to define a simple XML string and parse it into a Matlab variable. As the `idx`, `type`, and `size` attributes are defined, the resulting Matlab data type conforms to these specifications (class double vector of size [1x2]).

```
xmlstr = ...
'<root idx="1" type="double" size="1 2">3.1416 1.4142</root>';

V1 = xml_parse(xmlstr)
```

```
V1 =
     [3.1416, 1.4142]  % (class double)
```

Again, setting the `attswitch` parameter to 'off' lets the parser ignore the attributes and the returned variable is interpreted as a string.

```
V2 = xml_parse(xmlstr,'off')
```

```
V2 =
     '3.1416 1.4142'   % (class char)
```

Let's define a more complex data set in XML:

```
xmlstr =
'<root>
  <project>
    <name>myProjectName</name>
    <date>2004-09-13</date>
    <bytes>10472</bytes>
  </project>
  <project>
    <name>myProject Two</name>
    <date>2004-09-13</date>
    <bytes>9851</bytes>
  </project>
</root>'
```

```
v = xml_parse(xmlstr);
```

```
v:  1x2 struct array with fields:
     project
v(1).project:
     name: 'myProjectName'
     date: '2004-09-13'
     bytes: '10472'
v(2).project:
     name: 'myProject Two'
     date: '2004-09-13'
     bytes: '9851'
```

See also

xml_parseany, xml_formatany, xml_format, xml_load, xml_save, xml_help

xml_parseany

Parses an XML string with attributes and returns corresponding Matlab structure.

Syntax

```
v = xml_parseany(xmlstr)
```

Description

Parses XML string xmlstr and returns the corresponding Matlab structure, `v`. In comparison with xml_parse, this command reads all XML element attributes and returns these in additional attribute fields, thus enabling the user to read most types of XML into a Matlab variable.
This is a non-validating parser. XML entries starting with the exclamation mark tag "<!" and "<?" are ignored by the parser.
Any substructure is returned as a cell data type in Matlab as the parser assumes that child elements can contain any kind of complex XML element.

Input Arguments

`xmlstr`	XML string, for example read from file with `xmlstr = fileread(filename)`

Output Arguments

`v`	Matlab variable or structure with field .ATTRIBUTE if XML element attributes are present.

Examples

In this example, we specify an XML string and look at the difference between the `xml_parse` and `xml_parseany` functions:

```
xmlstr = ...
'<root idx="1" type="double" size="1 2">3.1416 1.4142</root>';

v1 = xml_parse(xmlstr);
```

```
v1:  [3.1416, 1.4142]  % (class double)
```

```
v2 = xml_parseany(xmlstr);
```

```
v1.ATTRIBUTE.idx = '1'
v1.ATTRIBUTE.type = 'double'
v1.ATTRIBUTE.size = '1 2'
v1.CONTENT = '3.1416 1.4142'
```

We see that the `xml_parse` command uses the specific attributes to convert the content into the corresponding Matlab data types. The function `xml_parseany`, however, returns all attributes in a substructure called ATTRIBUTE and the content in a field called CONTENT. `xml_parseany` does not use the attributes for type conversions to Matlab data types as these may not have originated from the XML Toolbox.

For more generic XML, the `xml_parseany` command acts as follows:

```
xmlstr = ...
      '<root color="red" language="en">
         <project id="alpha">
           <name>Project_Alpha</name>
           <author>Arthur</author>
           <link location="url">http://www.com/a</link>
         </project>
         <project id="beta">
           <name>Project_Beta</name>
           <author>Ben</author>
           <link location="file">c:\temp\b.pro</link>
         </project>
      </root>';


v = xml_parseany(xmlstr)
```

```
v =
      project: {[1x1 struct]  [1x1 struct]}
    ATTRIBUTE: [1x1 struct]
```

```
v.ATTRIBUTE
```

```
ans =
       color: 'red'
    language: 'en'
```

```
v.project{1}
```

```
ans =
         name: {[1x1 struct]}
       author: {[1x1 struct]}
         link: {[1x1 struct]}
    ATTRIBUTE: [1x1 struct]
```

```
v.project{2}.name{1}
```

```
ans =
    ATTRIBUTE: [0x0 struct]
      CONTENT: 'Project_Beta'
```

```
v.project{2}.link{1}
```

```
ans =
    ATTRIBUTE: [1x1 struct]
      CONTENT: 'c:\temp\b.pro'
```

```
v.project{2}.link{1}.ATTRIBUTE
```

```
ans =
    location: 'file'
```

Note

All subfields of the returned data structure are Matlab cell data types and therefore indexed with curly braces {.}. This adds a bit more complexity for the developer if the level of nesting is high; however, it also means that XML documents are returned to Matlab in a well-defined form.

Namespaces & valid Matlab variable names:
If an XML element has a namespace attached, for example "soap:services", the "soap" namespace is transferred into a subfield of the ATTRIBUTE structure, called "NAMESPACE". This is done to ensure that the name corresponds to a valid Matlab variable name. For the same reasons are any hyphens, "-" replaced by the underscore "_" during the parsing operation.

See also

`xml_formatany`, `xml_format`, `xml_parse`, `xml_load`, `xml_save`, `xml_help`

xml_save

Stores XML representation of Matlab variable or structure in XML format in a file.

Syntax

```
xml_save(filename,v)
xml_save(filename,v,attswitch)
```

Description

`xml_save` stores a Matlab variable in plain text XML format into the file specified by the user.

Input Arguments

The Matlab variable `v` can be any of the types supported by `xml_format`.

`filename`	full filename (including path and extension).
`v`	Matlab variable or structure to store in file.
`attswitch`	optional, 'on' stores XML type attributes (default), 'off' doesn't store XML type attributes.

See also

`xml_formatany`, `xml_format`, `xml_parse`, `xml_load`, `xml_help`

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="3.2.0" 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="3.2.0" 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="3.2.0" idx="1" type="double" size="0 0"/>

>> str = xml_format(pi)                      % pi

    <root xml_tb_version="3.2.0" idx="1" type="double" size="1 1">
     3.14159265358979</root>

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

    <root xml_tb_version="3.2.0" 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="3.2.0" 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>3.14159265358979</root>

>> str = xml_format(pi, 'off', 'alpha')         % attributes off
                                                   root='alpha'
    <alpha>3.14159265358979</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="3.2.0" 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="3.2.0" idx="1" type="cell" size="0 0"/>

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

    <root xml_tb_version="3.1" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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="3.2.0" 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.
```

Frequently Asked Questions

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

A: Namespaces get currently ignored by the parser `xml_parse`. 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`"). The command `xml_parseany` has the possibility to read namespaces and attributes into Matlab struct variables though.

Q: Why is there not an XML Schema available in this version 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 and thus we decided to ignore such.

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, there 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.

Q: I have problems with the following command as it returns false. Why?

pi == xml_parse(xml_format(pi))

A: This is due to rounding when double values are stored in XML. We support a total of 14 significant decimals, however, the constant pi would need 16. We would like to use 16, however, if you use e.g. sprintf('%0.16g ', [-0.1:0.01:0.1]), the output is rounded in a funny way, i.e. you will get something like [-0.1 -0.09 -0.08 -0.0699999999 -0.05 …] Hence: 14 significant decimals are stored are pi() in Matlab thus differs (although only slightly) from xml_parse(xml_format(pi)).

Q: Why do you only provide .p files?

A: Due to copyright and maintenance reasons. Some people unfortunately change the .m code and then expect support for their version. We are happy to provide support and feature requests, however, only for tested and maintained .p code.

Q: Can I request additional functionality or features?

A: Sure, we are always happy about feedback and try to accommodate requests if we think they are of general use (and find the time). Please email your request to the author for the time being: m.molinari@soton.ac.uk.

Q: How are *<![CDATA[some stuff]]>* entries in the XML string handled by xml_parse?

A: Not at all. These are being ignored as any other entry that starts with "<*!*" .
There might be some problems with the XML during parsing if the CDATA section contains XML code itself (which is allowed according to the XML specification).