The Geodise Toolboxes

A User's Guide

Geodise

Release: GeodiseLab v.1.1

Version: GeodiseManual 1.2

Title: The Geodise Toolboxes – A User's Guide

Authors: Dr Graeme Pound, g.e.pound@soton.ac.uk
Dr Jasmin Wason, j.l.wason@soton.ac.uk
Dr Marc Molinari, m.molinari@soton.ac.uk
Dr Hakki Eres, hakki.eres@soton.ac.uk
Dr Zhuoan Jiao, z.jiao@soton.ac.uk

PI: Prof Simon Cox, s.j.cox@soton.ac.uk

Web: http://www.geodise.org/

Acknowledgement:

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

Contents

The Geodise Toolboxes 1
Contents 3
Introduction 5
  Use Cases 6
Function Arguments 9
  Input Arguments 9
  Output Arguments 14
Geodise Compute Toolbox 16
  Introduction 16
  Tutorial 19
  Function Reference 30
    gd_certinfo 30
    gd_chmod 31
    gd_condorsubmit 33
    gd_compute_version 37
    gd_createproxy 38
    gd_destroyproxy 39
    gd_fileexists 40
    gd_getfile 41
    gd_jobkill 43
    gd_jobpoll 44
    gd_jobstatus 45
    gd_jobsubmit 46
    gd_listdir 47
    gd_makedir 48
    gd_proxyinfo 49
    gd_proxyquery 50
    gd_putfile 51
    gd_rmdir 53
    gd_rmfile 54
    gd_rmuniquedir 55
    gd_servermetrics 56
    gd_submitunique 58
    gd_testauthentication 60
    gd_testfiletransfer 61
    gd_testjobsubmission 62

gd_timeauthentication....63
gd_timefiletransfer....64
gd_timejobsubmission....65
gd_transferfile....66
Geodise Database Toolbox....67
Introduction....67
Tutorial....68
Function Reference....80
gd_addusers....80
gd_archive....82
gd_datagroup....86
gd_datagroupadd....89
gd_dbsetup....91
gd_db_help....92
gd_db_version....93
gd_display....94
gd_markfordeletion....96
gd_query....98
gd_querydeleted....105
gd_retrieve....108
gd_unmarkfordeletion....111
XML Toolbox....113
Introduction....113
Tutorial....115
Function Reference....121
xml_format....121
xml_formatany....124
xml_help....126
xml_load....127
xml_parse....129
xml_parseany....131
xml_save....134

Introduction

The Geodise Toolboxes provide a collection of functions that extend the capability of the Matlab® 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 jobs 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 stored files and variables.
- 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. The XML toolbox also enables the transparent exchanges of data between the Matlab technical computing environment and the Jython scripting environment.

This user guide introduces the reader to the Compute, Database and XML toolboxes, with tutorials that give an overview of the functionality provided by each of the toolboxes. The function reference for each toolbox contains detailed information about the syntax of its functions.

Use Cases

The GeodiseLab toolboxes have applications in a wide range of scenarios. Here we will outline three use cases that describe the potential benefits of Grid computing to the daily practice of the scientist or engineer.

The use cases that we will discuss are:

- Engineering Design Search and Optimisation
- Data management in computational electromagnetics
- Transparent collaboration between Problem Solving Environments

Engineering Design Search and Optimisation

Engineering Design Search and Optimisation (EDSO) is a compute and data intensive task which is well matched to Grid computing. Optimisation algorithms are used to search the parameter space of an engineering problem to discover an optimal design subject to certain criteria. During EDSO the optimisation algorithm must repeatedly evaluate some measure of the quality of a design; this may involve one or more lengthy numerical calculations. For example, an engineer wishing to improve the aerodynamic performance of a wing design may configure an optimiser to vary key design parameters, whilst invoking simulations of Computational Fluid Dynamics (CFD) to determine the *quality* of alternative geometries.

Depending upon the complexity of the numerical calculations and the number of evaluations required to determine the optimum design, EDSO may be a lengthy and computationally intensive task. When the evaluation of the objective function involves complex simulations (i.e. CFD) numerous large data files may be required, or produced, by the numerous calculations. The Grid client functionality makes it straightforward for the engineer to leverage computational resources available on the Grid to perform EDSO.

When undertaking EDSO using one of the many optimisation algorithms available in the Matlab environment the engineer may use the Geodise Compute toolbox to automate the transfer of files, and the submission and management of computational jobs required during the evaluation of a design. By exploiting Grid resources not only is the engineer able to leverage the greater computational power available, but he can also drive any applications that he requires on a multitude of platforms from the comfort of his desktop PSE.

Data management in computational electromagnetics

Data management is an issue in a number of scientific and engineering application domains, including that of computational electromagnetics. For example, when performing simulations of electromagnetic phenomena a large volume of data may be generated, typically in the form of the input and output files. It is a non-trivial problem for the researcher to store, manage and reuse this data. The investment associated with the computationally expensive Finite Difference Time Domain modelling technique used to explore the properties of electromagnetic devices require that simulation results are suitably managed for reuse at a later data.

At present the most common solution for this problem is to store these flat files within a hierarchical directory structure on a local file system. As the volume of data grows over time this solution is frequently inadequate for long term storage since it may become increasingly difficult to locate and reuse data within the collection. The Geodise Database toolbox provides a solution as a client to a managed data archive on the Grid.

The Geodise Database Toolbox allows the researcher to archive data files to a managed repository from the Matlab environment and annotate these files with metadata. In addition to standard metadata the user may define custom metadata specific to the problem. The researcher can then query the metadata to find these files using straightforward syntax within the Matlab environment. In addition the Geodise Database Toolbox supports the archiving of variables from the Matlab environment. Items in stored the repository can be associated together into datagroups, allowing the creation of annotated hierarchies within which the user's results can be organised.

Transparent collaboration between Problem Solving Environments

The Geodise XML toolbox provides a collection of straight-forward functions which convert variables in the Matlab environment to and from the external XML format. Variables in the Matlab workspace can be saved to and loaded from an XML file with minimal effort on the part of the researcher. XML is a structured format that can be interpreted by third party applications. By encoding the Matlab variables in the XML format there are a number of benefits.

The provision of the Geodise XML toolbox for Jython allows the transparent exchange of variables between the Matlab technical computing environment and Jython scripting environment. Variables are mapped to the appropriate built-in datatypes in the two languages. This allows researchers working with these two Problem Solving Environments to collaborate on shared datasets.

The Geodise XML toolbox is also leveraged by the Geodise Database Toolbox to store variables and metadata in a database. The contents of variables and metadata in the database can then be queried and searched across. The Geodise Database toolbox may be used to share variables stored in the managed repository between members of a virtual organisation because researchers can authorise other users to access their data. When variables are retrieved from the repository they will be transparently converted into the built-in datatypes of that PSE.

Function Arguments

The input and output arguments used by all of the functions of the Geodise toolboxes are summarised below.

Input Arguments

Argument	Description	Used by Functions
attswitch	A string specifying whether to use attributes ('on' = use attributes, 'off' = no attributes).	xml_format xml_load xml_parse xml_save
classAD	A structure describing the job to be submitted to the Condor pool.	gd_condorsubmit
command	The absolute path of the chmod command on the Globus resource.	gd_chmod
datagroupID	The unique identifier of a datagroup.	gd_addusers gd_archive gd_datagroupadd
datagroupname	A user defined name for a datagroup.	gd_datagroup
datasource	Specifies what type of metadata or data to query ('file', 'datagroup', 'varmeta', 'var' or 'monitor').	gd_query gd_querydeleted
directory	The path of a local directory.	gd_retrieve
filename	The path of a local file.	gd_archive gd_certinfo gd_retrieve xml_load xml_save
files	A cell array of filenames.	gd_submitunique
filesystem	The type of the filesystem used	gd_condorsubmit

Argument	Description	Used by Functions
	by the Globus server.	

`filetype`	A string specifying the GridFTP transfer type (‘ASCII’ or ‘binary’).	`gd_getfile`
		`gd_putfile`
		`gd_transferfile`

`groups`	A user group ID string or list of user group IDs.	`gd_addusers`

`host`	A string specifying the Globus server to be used.	`gd_chmod`
		`gd_condorsubmit`
		`gd_fileexists`
		`gd_getfile`
		`gd_jobsubmit`
		`gd_listdir`
		`gd_makedir`
		`gd_putfile`
		`gd_rmdir`
		`gd_rmfile`
		`gd_rmuniquedir`
		`gd_submitunique`
		`gd_testauthentication`
		`gd_testfiletransfer`
		`gd_testjobsubmission`
		`gd_timeauthentication`
		`gd_timefiletransfer`
		`gd_timejobsubmission`

`host1`	The Globus server that sends the file.	`gd_transferfile`
`host2`	The Globus server that receives the file.	`gd_transferfile`
`hostprompt`	Indicates whether to prompt user for file host configuration during setup (1=true, 0=false).	`gd_dbsetup`

`ID`	The unique identifier of a file or variable.	`gd_addusers`
		`gd_datagroupsadd`
		`gd_retrieve`

Argument	Description	Used by Functions
`IDs`	A cell array which may contain the unique identifiers of files, variables and datagroups.	`gd_markfordeletion` `gd_unmarkfordeletion`
`interval`	Interval (in seconds) at which the status of the job is polled.	`gd_jobpoll`

`jobhandle`	A Globus GRAM job handle.	`gd_jobkill`
		`gd_jobpoll`
		`gd_jobstatus`

`listhidden`	Indicates whether hidden files should be listed (1 = true, false otherwise).	`gd_listdir`
`localfile`	A filename on the local machine.	`gd_getfile` `gd_putfile`
`maxtime`	Upper limit (in seconds) for the period over which the job is polled.	`gd_jobpoll`
`metadata`	A metadata structure containing information about a file, variable or datagroup.	`gd_archive` `gd_datagroup`
`minvalue`	The minimum acceptable value for the property of the proxy certificate examined (in hours or bits).	`gd_proxyquery`
`mode`	Permissions to be set on the file.	`gd_chmod`
`name`	Name to use for the root element.	`xml_format` `xml_formatany`
`proxyattrib`	A string specifying the property of the proxy certificate to be examined (‘time’ or ‘strength’).	`gd_proxyquery`
`qresults`	Cell array of structure(s) containing results returned from a query.	`gd_display`

Argument	Description	Used by Functions
query	A query string which compares fields with values.	gd_query gd_querydeleted
remotedir	The path of a directory on a Globus server.	gd_listdir gd_makedir gd_rmdir gd_rmuniquedir gd_submitunique gd_testfiletransfer gd_testjobsubmission gd_timefiletransfer gd_timejobsubmission
remotefile	A filename on the remote server.	gd_chmod gd_fileexists gd_getfile gd_putfile gd_rmfile gd_servermetrics
remotefile1	The path of the file to be sent.	gd_transferfile
remotefile2	The path of the file to be received.	gd_transferfile
resultfields	A string specifying selected fields to return from a query.	gd_query gd_querydeleted
RSL	A string specifying the properties of a Globus GRAM job.	gd_jobsubmit gd_submitunique
RSLstruct	A structure specifying the properties of a Globus GRAM job.	gd_condorsubmit
servers	A structure specifying the name and working directories of the Globus servers to be tested.	gd_servermetrics
subdatagroupID	The unique identifier of a datagroup that is added to another datagroup.	gd_datagroupadd

Argument	Description	Used by Functions
`users`	A user ID string or cell array of user IDs.	`gd_addusers`

`v`	A generic structure or variable.	`gd_archive`
		`xml_format`
		`xml_formatany`
		`xml_save`

`xmlstr`	An XML string.	`xml_parse` `xml_parseany`

Output Arguments

Argument	Description	Used by Functions
datagroupID	The unique identifier of a datagroup.	gd_datagroup
details	A cell array containing structures that describe the details of the files and directories contained in the remote directory.	gd_listdir
exists	The existence of the file on the Globus server (1 = exists, 0 = does not exist).	gd_fileexists
filename	The path of a local file.	gd_retrieve
files	A cell array of filenames.	gd_listdir
ID	The unique identifier of a file or variable.	gd_archive
isdone	Indicates whether the job complete successfully (1 = done, 0 = not done).	gd_jobpoll
isvalid	Indicates whether the proxy certificate is valid (1 = valid, 0 = not valid).	gd_proxyinfo, gd_proxyquery
jobhandle	A Globus GRAM job handle.	gd_condorsubmit gd_jobsubmit gd_submitunique
marktotal	Total number of IDs successfully marked for deletion.	gd_markfordeletion
metadata	A metadata structure containing information about a file, variable or datagroup.	gd_retrieve
qresults	Cell array of structure(s) containing results returned from a query.	gd_query gd_querydeleted

Argument	Description	Used by Functions
`status`	The status of the Globus GRAM job.	`gd_jobstatus`
`subject`	The certificate subject line in the Globus format.	`gd_proxyinfo` `gd_certinfo`

`success`	The result of the operation or test (1 = success, 0 = failure).	`gd_addusers`
		`gd_datagroupadd`
		`gd_testauthentication`
		`gd_testfiletransfer`
		`gd_testjobsubmission`

`time`	The elapsed time in milliseconds or -1 if failed.	`gd_timeauthentication`
		`gd_timefiletransfer`
		`gd_timejobsubmission`

`testresults`	A structure containing the results of tests upon an array of servers.	`gd_servermetrics`
`uniquedir`	The path of the unique working directory created on the server.	`gd_submitunique`
`unmarktotal`	Total number of IDs successfully unmarked for deletion.	`gd_unmarkfordeletion`

`v`	A generic structure or variable.	`gd_retrieve`
		`xml_parse`
		`xml_parseany`
		`xml_load`

`version`	Version of the Database or Compute toolbox.	`gd_compute_version` `gd_db_version`
`xmlstr`	An XML string.	`xml_format` `xml_formatany`

Geodise Compute Toolbox

Introduction

The Geodise Compute Toolbox exposes the power of the Grid to the Matlab technical computing environment. With this toolbox the engineer can programmatically access Globus GT2 resources which provide the backbone of many computational Grids. In this manner the Geodise Compute Toolbox promotes the integration of Grid resources into the complex engineering workflows which can be described within the Matlab environment.

The Geodise Compute Toolbox provides Matlab functions which support the job submission, file transfer and certificate management in a familiar and intuitive syntax.

- Globus GRAM jobs can be submitted, queried and terminated.
- File transfer and remote directory management is supported using the GridFTP protocol.
- Single sign-on to the Grid is supported with Globus proxy certificates.

The Geodise Compute Toolbox functions for certificate management are listed in Table 1. Table 2 lists functions for the submission the computational jobs to a Globus GRAM service, and Table 3 lists the functions for GridFTP file transfer. In addition there are a number of functions to define the availability and performance of a GridFTP server (Table 4).

`gd_certinfo`	Returns information about the user's certificate.
`gd_createproxy`	Creates a Globus proxy certificate.
`gd_proxyinfo`	Returns information about the user's proxy certificate.
`gd_proxyquery`	Queries whether a valid proxy certificate exists.
`gd_destroyproxy`	Destroys the local copy of the user's Globus proxy certificate.

Table 1 Certificate management functions

gd_jobstatus	Gets the status of a Globus GRAM job.
gd_jobsubmit	Submits a compute job to a Globus GRAM job manager.
gd_jobpoll	Queries the status of a Globus GRAM job until complete.
gd_jobkill	Kills a Globus GRAM job specified by a job handle.
gd_chmod	Changes file permissions of a file on a Globus resource.
gd_condorsubmit	Submits a job through a Globus resource to a Condor pool.
gd_submitunique	Submits a GRAM job to a unique working directory.

Table 2 GRAM job submission functions

gd_getfile	Retrieves a remote file using GridFTP.
gd_putfile	Puts a file on a remote server using GridFTP.
gd_transferfile	Performs a third-party file transfer using GridFTP.
gd_makedir	Creates a remote directory using GridFTP.
gd_listdir	Lists the contents of a directory on a GridFTP server.
gd_fileexists	Tests the existence of files and directories on a Globus resource.
gd_rmdir	Deletes a remote directory using GridFTP.
gd_rmfile	Deletes a remote file using GridFTP.
gd_rmuniquedir	Deletes a remote directory and its contents.

Table 3 GridFTP file transfer functions

gd_servermetrics	Performs a number of tests upon a list of Globus resources.
gd_testauthentication	Tests authentication with a Globus resource.
gd_testfiletransfer	Tests file transfer to a Globus resource.
gd_testjobsubmission	Tests the job submission to a Globus resource.
gd_timeauthentication	Times authentication to a Globus resource.
gd_timefiletransfer	Times file transfer to a Globus resource.
gd_timejobsubmission	Times a job submission to a Globus resource.

Table 4 Globus resource testing functions

Tutorial

Grid Certificates

To access Globus compute resources all users must be authenticated, and must also be authorised to access the resource. Authentication under the Globus toolkit is based upon X.509 certificates. X.509 certificates are digital tokens that have been cryptographically signed by a trusted third party, the Certificate Authority (CA). Using X.509 certificates the identity of a user or server can be verified.

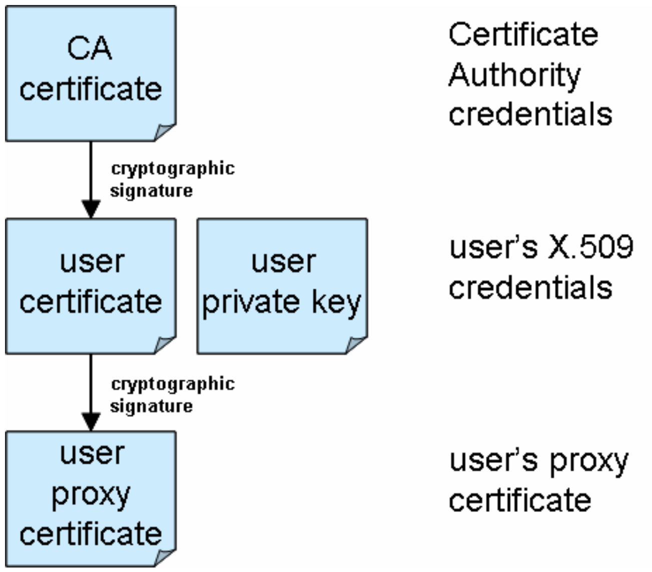

Figure 1 - Hierarchy of trust for user credentials

It is necessary to obtain a Grid certificate from a Certificate Authority that is acceptable to the administrators of the Globus resources that you wish to use. For step-by-step instructions about how to apply for an X.509 certificate, and how to export it into the format required by Compute Toolbox, a tutorial is available from the Geodise web-site (http://www.geodise.org/files/tutorials/Obtaining_Certificates.pdf).

The Globus toolkit authorises users to access resources by mapping their certificate to a user account on the resource. Therefore to use a Globus resource to run computational jobs you must be in possession of an X.509 certificate signed by a CA that is trusted by the administrators of the resource that you wish to access. You must then apply for permission to access the resource by having the subject line of your certificate mapped to a user account on that machine.

To enable users to delegate their identity, allowing Grid processes to submit jobs and transfer files on their behalf, the Globus toolkit also uses a technology called ‘proxy certificates'. Proxy certificates are temporarily limited credentials that can be used to devolve the user's identity across the Grid. In practice proxy certificates also provide a convenient single sign-on to the Grid; users enter the passphrase to the private key of their X.509 certificate just once when generating the proxy certificate.

Before accessing a Globus resource you should generate a valid proxy certificate, which will typically expire after 12 hours. The Geodise Compute Toolbox provides Matlab functions that allow the user to create, examine and destroy Globus proxy certificates within the Matlab environment.

Before using the Geodise Compute Toolbox you should configure the location of the credentials on your machine. Your X.509 certificate and corresponding private key should be separately encoded in PEM format (see the obtaining certificates tutorial for details). To do this create a file called 'cog.properties' located in a directory '.globus' of the home directory on your workstation. Then configure the location of your X.509 certificate and private key, in addition to the certificates of trusted CA.

For example the 'cog.properties' file on a Windows PC would contain the following lines:

```
cacert=C\:\\Documents and Settings\\<USER>\\.globus\\01621954.0
proxy=C\:\\DOCUME~1\\<USER>\\LOCALS~1\\Temp\\509up_u_<USER>
usercert=C\:\\Documents and Settings\\<USER>\\.globus\\usercert.pem
userkey=C\:\\Documents and Settings\\<USER>\\.globus\\userkey.pem
proxy.strength=512
proxy.lifetime=12
```

Please note that throughout this manual the term <USER> represents your username on any given machine.

The properties 'usercert' and 'userkey' refer to locations of the PEM encoded user certificate and corresponding private key. The file 'cacert' contains the certificate of the CA which signed the user's X.509 certificate (in PEM format). Where 'proxy' will be the location of the user's proxy certificate once it has been generated by `gd_createproxy`. The properties 'proxy.strength' and 'proxy.lifetime' contain default settings for the cryptographic strength and lifetime of the proxy certificate. Note that the file separator on a Windows PC must be defined with double backslashes, "\\".

Once the user’s credentials have been configured in the ‘cog.properties’ file they are accessible to the Geodise Compute Toolbox. To verify the configuration from within the Matlab environment query the X.509 certificate:

```
>> subject = gd_certinfo
```

```
subject: C=UK,O=eScience,OU=Southampton,L=SeSC,CN=some user
issuer: C=UK,O=eScience,OU=Authority,CN=CA,E=ca-
operator@grid-support.ac.uk
start date: Tue Oct 07 13:00:31 BST 2003
end date: Wed Oct 06 13:00:31 BST 2004

subject =
/C=UK/O=eScience/OU=Southampton/L=SeSC/CN=some user
```

The details of the user’s certificate are printed to the screen. The subject line returned by `gd_certinfo` is in the Globus format and can be used to apply for access to a Globus resource. By supplying this subject line to the administrator of a Globus resource your credentials can be mapped to a user account on that machine.

To create a proxy certificate the `gd_createproxy` command is used:

```
>> gd_createproxy
```

When this command is entered a GUI will prompt the user for the passphrase to their private key. The details of the proxy certificate can be configured using the 'Options' button. The proxy certificate is generated by pressing the 'Create' button. After the proxy has been generated, click 'Cancel' to dismiss the GUI, and press 'Enter' at the Matlab prompt.

Now you may query the details of the proxy certificate:

```
>> gd_proxyinfo;
```

```
Subject: C=UK,O=eScience,OU=Southampton,L=SeSC,CN=some
user,CN=proxy
issuer: C=UK,O=eScience,OU=Southampton,L=SeSC,CN=some user
type: full legacy globus proxy
strength: 512 bits
timeleft: 11 h, 59 min, 39 sec
```

The details printed to the screen indicate that the proxy certificate will remain valid for almost 12 hours. We may also query the validity of the proxy certificate programmatically, for example:

```
>> isvalid = gd_proxyquery('time',11)
```

```
isvalid =
     1
```

This indicates that our proxy certificate will remain valid for at least 11 hours.

Job submission and file transfer

The primary services offered by Globus GT2 resources are GRAM job submission and GridFTP file transfer. Typically Globus resources can simply be specified by the machine name, for example:

```
>> host = 'server1.domain.com';
```

However some Globus computational resources may offer GRAM job submission to a number of alternative job managers or non-default ports. These can be specified as follows:

```
>> GRAM1 = 'server1.domain.com/jobmanager-fork';
>> GRAM2 = 'server1.domain.com/jobmanager-pbs';
>> GRAM3 = 'server1.domain.com:2119/jobmanager';
```

Globus resources offering GridFTP will typically listen on the default port (2811), however a non-default port can be specified as follows:

```
>> GridFTP1 = 'server1.domain.com:2812';
```

For all examples in this tutorial we will assume that a single Globus resource (`host`) is used offering GRAM and GridFTP services on default ports, and using the default job manager.

To submit a job to a computational resource via a Globus GRAM service you must describe the attributes of the job using a Resource Specification Language (RSL) string. An RSL string is a list of property/values pairs each enclosed by brackets (see the example below). The most frequently used GRAM RSL parameters are listed in Table 5, these and other GRAM RSL parameters are further documented on the Globus website (http://www.globus.org/).

`executable`	The name of the executable file to be run. This is the only required parameter.
`directory`	The name of the default working directory.
`arguments`	The arguments to be passed to the executable.
`stdin`	The name of the file containing the standard input for the executable.
`stdout`	The name of the file that will contain the standard output from the executable.
`stderr`	The name of the file that will contain the standard error from the executable.
`count`	The number of times that the executable should be executed.
`environment`	The environment variables to be set. A list of name/value pairs each enclosed by brackets.
`maxTime`	The maximum execution time in minutes.
`jobType`	A string specifying the job types. Possible values include “single”, “multiple”, “mpi” and “condor”.

Table 5 GRAM RSL parameters

This example demonstrates the submission of a simple job to the Globus GRAM service on `host`. The first argument to `gd_jobsubmit` is an RSL string that specifies the file name of the executable to be run, ‘sleep’, and the argument to be passed to that executable which specifies that the process will sleep for 1 minute.

```
>> jobhandle = gd_jobsubmit(
'&(executable="/bin/sleep")(arguments="1m")',host)
```

```
jobhandle =
https://server1.domain.com:30001/27531/1096385757/
```

The function `gd_jobsubmit` returns a GRAM job handle that can be used to check the status of the job, and if necessary to kill the job. In the following example we use the job handle returned by `gd_jobsubmit` to query the status of the job. The integer returned by `gd_jobstatus` indicates the state of the job, where “2” indicates that the job is active and “3” indicates that the job has completed.

```
>> status = gd_jobstatus(jobhandle)
```

```
status =
     2
```

We can also poll the status of the job until the job has completed.

```
>> gd_jobpoll(jobhandle)
```

In addition to high-performance, high-volume file transfer GridFTP offers all of the standard FTP file operations. We can use GridFTP to create a working directory on the Globus resource.

```
>> gd_makedir(host,'/home/<USER>/demo')
```

We will now run a second job, piping the output to a file ‘date.out’ in our working directory on `host`. We will then use the GridFTP command `gd_getfile` to retrieve the output to a temporary file on the local machine, and print the results.

```
>> rsl = '&(executable="/bin/date")(arguments="-u")
(directory="/home/<USER>/demo")(stdout="date.out")';
>> jobhandle = gd_jobsubmit(rsl,host)
```

```
jobhandle =
https://server1.domain.com:30001/27531/1096385757/
```

```
>> gd_jobpoll(jobhandle);
>> localfile = tempname;
>> gd_getfile(host,'/home/<USER>/demo/date.out',localfile);
>> type(localfile)
```

```
Tue Sep 28 16:46:25 BST 2004
```

We can now use the GridFTP commands `gd_rmfile` and `gd_rmdir` to clean-up the file and directory on the server:

```
>> gd_rmfile(host,'/home/<USER>/demo/date.out')
>> gd_rmdir(host,'/home/<USER>/demo/')
```

Frequently an engineer may wish to submit and run several jobs independently upon a Globus resource, for example when conducting a parameter sweep. To prevent conflicts between the input and output parameters of the different jobs it is convenient to run the jobs in separate directories. The function `gd_submitunique` handles the submission of compute jobs into unique directories, returning a job handle and the path of the unique directory. In the following example we use the function `gd_submitunique` to submit two concurrent jobs, we will then retrieve the results and delete unique directories and their contents using `gd_rmuniquedir`.

```
>> rsl = '&(executable="/bin/date")(arguments="-u")
(stdout="date.out")';
>> [jobhandle1,uniquedir1] =
gd_submitunique(rsl,host,[],'/home/<USER>/')
>> [jobhandle2,uniquedir2] =
gd_submitunique(rsl,host,[],'/home/<USER>/')
```

```
jobhandle1 =
https://server1.domain.com:30002/27658/1096386586/

uniquedir1 =
/home/<USER>/20040928T164946_176266/

jobhandle2 =
https://server1.domain.com:30002/27671/1096386587/

uniquedir2 =
/home/<USER>/20040928T164947_405706/
```

```
>> gd_jobpoll(jobhandle1);
>> localfile = tempname;
>> gd_getfile(host,[uniquedir1,'date.out'],localfile);
>> type(localfile)
>> gd_rmuniquedir(host,uniquedir1);
```

```
Wed Sep 29 12:12:21 UTC 2004
```

```
>> gd_jobpoll(jobhandle2);
>> localfile = tempname;
>> gd_getfile(host,[uniquedir2,'date.out'],localfile);
>> type(localfile)
>> gd_rmuniquedir(host,uniquedir2);
```

```
Wed Sep 29 12:12:23 UTC 2004
```

Scripting the Grid

The Geodise Compute Toolbox allows engineers to script Grid processes in the Matlab environment. Unfortunately due to the dynamic nature of the Grid the resources that you wish to use may become unavailable, or may be more or less reliable. In these situations, when a function in the Geodise Compute Toolbox is unable to complete its operation, the function will typically throw an error with a diagnostic message.

```
>> gd_getfile(host,'\tmp\fileDoesNotExist.txt','demo.txt')
```

```
??? Error using ==> gd_getfile
Server Exception: No such file or directory.
```

If a Matlab function throws an error, this will cause the Matlab script or function which invoked it to stop executing. Therefore it is important if you wish to write a robust Matlab script or function that communicates with the Grid that you use Matlab exception handling to deal with errors appropriately if and when they occur.

Matlab exception handling is based upon `try`, `catch` statements. By placing a block of code between a pair of `try`, `catch` statements means that if an error occurs when Matlab evaluates this code the script will not stop executing. Instead the code inside the `catch`, `end` block is evaluated and the script continues. This behaviour is demonstrated by the example below.

```
>> try
       gd_getfile(host,'\tmp\fileDoesNotExist.txt','demo.txt')
   catch
       disp('An error has occurred with the following
message:')
       disp(lasterr)
   end
```

```
An error has occurred with the following message:
Error using ==> gd_getfile
Server Exception: No such file or directory.
```

In this way errors that occur when communicating with the Grid can be ‘caught’ by the script and dealt with appropriately. The diagnostic error message can be examined with the `lasterr` function, and if appropriate the script can continue, or stopped by throwing another error (using `error` or `rethrow`).

We recommend that when writing a script or function that communicates with the Grid that you enclose all Grid functions with `try`, `catch` statements. You should also consider how the script should behave if an error occurs; should it tidy up and exit, or should it continue? This way you will be prepared for the unexpected, and your Matlab scripts and functions will be more robust as a result.

Testing Grid resources

The unpredictability of Grid resources mean that is often wise to determine whether a resource is functioning and responsive before attempting to use it. The Geodise Compute Toolbox provides a suite of functions to test the availability and responsiveness of the Globus services running on a resource.

To determine whether a resource is responding, and whether or not you are authorised to access it, the following commands may be used:

```
>> success = gd_testauthentication(host)
>> time = gd_timeauthentication(host)
```

```
success =
     1


time =
   171
```

To test the availability of the GRAM job submission service, the following commands will submit a small job to the job manager specified by `host`.

```
>> success = gd_testjobsubmission(host)
>> time = gd_timejobsubmission(host)
```

```
success =
     1


time =
   610
```

To test the availability and speed of GridFTP file transfers to a Globus resource the following commands will transfer a small file to the specified directory on `host`:

```
>> success = gd_testfiletransfer(host)
>> time = gd_timefiletransfer(host)
```

```
success =
     1

time =
   890
```

Function Reference

gd_certinfo

Returns information about the user's certificate.

Syntax

```
subject = gd_certinfo
subject = gd_certinfo(filename)
```

Description

This command prints information about the user's certificate to the screen. The command also returns the certificate subject line in a format which is suitable for use in a Globus gridmap file. The default location of the user's certificate is specified by the cog.properties file.

`subject = gd_certinfo` where `subject` is the certificate subject in the *Globus* format.

`subject = gd_certinfo(filename)`as above, where `filename` is the filename of the certificate to be queried. The certificate must be encoded in pem format.

See also

`gd_proxyinfo`, `gd_createproxy`, `gd_destroyproxy`

gd_chmod

Changes file permissions of a file on a Globus resource.

Syntax

`gd_chmod(host,remotefile,mode)`

`gd_chmod(host,remotefile,mode,command)`

Description

`gd_chmod(host,remotefile,mode)` where `host` is a string describing the resource. It could be in one of the following formats:

- hostname
- hostname:port
- hostname/service
- hostname:port/service

The second argument `remotefile` is a string describing the full name of the file starting from root '/'. The final argument `mode` is a string describing the permissions of the file. The permission of a file can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new permissions (see below).

`gd_chmod(host,remotefile,mode,command)` as above, except the argument `command` is a string specifying the absolute path of the chmod command on the Globus resource.

Input arguments

`mode` The argument `mode` may have two alternative forms:

1. Symbolic representation:
A combination of the letters `ugoa' controls which users' access to the file will be changed: the user who owns it (u), other users in the file's group (g), other users not in the file's group (o), or all users (a).

The operator '+' causes the permissions selected to be added to the existing permissions of each file; '-' causes them to be removed; and '=' causes them to be the only permissions that the file has.

The letters 'rwxXstugo' select the new permissions for the affected users: read (r), write (w), execute (or access for directories) (x), execute only if the file is a directory or already has execute permission for some user (X), set user or group ID on execution (s), sticky (t), the permissions granted to the user who owns the file (u), the permissions granted to other users who are members of the file's group (g), and the permissions granted to users that are in neither of the two preceding categories (o).

2. Octal number representation:
A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Any omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

For example, 0750 gives rwx permissions to the owner and rx permissions to the group.

Examples

To give read/write/execute permissions to the owner and read/execute permissions to the group of a file named '/tmp/foo' which is on a Globus resource called 'server.domain.com', you can use:

```
gd_chmod('server.domain.com','/tmp/foo','0750');
```

To remove group execute permissions from of the same file you can use:

```
gd_chmod('server.domain.com','/tmp/foo','g-x');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_fileexists`, `gd_listdir`

gd_condorsubmit

Submits a job through a Globus resource to a Condor pool.

Syntax

```
handle = gd_condorsubmit(classAD,RSLstruct,host)
handle = gd_condorsubmit(classAD,RSLstruct,host,filesystem)
```

Description

`handle = gd_condorsubmit(classAD,RSLstruct,host)` returns a string `handle` containing the Globus job handle for a successfully submitted job. Where `classAD` is a structure describing the job to be submitted to the Condor pool, the structure, `RSLstruct` describes the command used to submit the job to the Condor pool, and `host` is a string describing the Globus resource to be used. The argument `host` can have one of the following formats:

- hostname
- hostname:port
- hostname/service
- hostname:port/service

`handle = gd_condorsubmit(classAD,RSL,host,filesystem)` as above where the argument `filesystem` defines the filesystem on `host`. When `filesystem` = 'NFS' a shared filesystem is assumed, otherwise a non-NFS filesystem is assumed.

Input arguments

`classAD` The `classAD` structure contains a description of the requirements of the job to be submitted to the Condor pool. The fields of the structure specify the attributes of the Condor classAD file that is used to submit the job. The valid fields include:

`executable`	The name of the executable to be submitted to the Condor pool
`requirements`	A string specifying the requirements from the machine upon which to the job should be run. These requirements may include:

	• Operating system: `OpSys` • Architecture: `Arch` • Memory: `Memory`
`arguments`	The arguments to the executable
`transfer_input_files`	A string containing a comma separated list of files to be submitted with executable
`output`	The filename to pipe the output from the job
`error`	The filename to pipe the error from the job
`log`	The filename to which to write the Condor log
`universe`	A string specifying the type of Condor job to be run. Possible values include: • `STANDARD` • `VANILLA` • `MPI` • `JAVA`

Other possible fields in the `classAD` structure include all of the valid classADs attributes. These attributes are documented at the Condor project homepage (http://www.cs.wisc.edu/condor/).

`RSLstruct` The `RSLstruct` structure contains the RSL attributes which specifies the command used to submit the job to the Condor pool. The required fields include:

`executable`	The path to the 'condor_submit' executable on `host`
`arguments`	The name of the Condor classAD file produced by `gd_condorsubmit`
`directory`	The name of the working directory on `host`

Examples

The following example demonstrates the submission of a Linux and a Windows job to a Condor pool via the Globus server 'server.domain.com'. The Linux job is described by the structure `classAD_Linux`, and the Windows job is described by the structure `classAD_Windows`.

```
%Specify classAD_Linux
classAD_Linux.requirements = 'Arch == "INTEL" && OpSys ==
"LINUX"';
classAD_Linux.executable = 'sleep.sh';
classAD_Linux.output = 'sleep.output';
classAD_Linux.error = 'sleep.error';
classAD_Linux.log = 'sleep.log';
classAD_Linux.universe = 'VANILLA';
classAD_Linux.transferfiles = 'ONEXIT';
classAD_Linux.should_transfer_files = 'YES';
classAD_Linux.when_to_transfer_output = 'ON_EXIT';
classAD_Linux.arguments = '1m';

% Specify classAD_Windows
classAD_Windows.requirements = 'Arch == "INTEL" && OpSys ==
"WINNT51"';
classAD_Windows.environment = 'path=c:\windows\system32';
classAD_Windows.executable = 'printname.bat';
classAD_Windows.output = 'printname.output';
classAD_Windows.error = 'printname.error';
classAD_Windows.log = 'printname.log';
classAD_Windows.universe = 'VANILLA';
classAD_Windows.transferfiles = 'ALWAYS';
classAD_Windows.should_transfer_files = 'YES';
classAD_Windows.when_to_transfer_output = 'ON_EXIT';
classAD_Windows.transfer_input_files = 'file1.txt, file2.txt,
file3.txt';

% Specify RSL
rsl.executable = '/usr/local/condor/bin/condor_submit';
rsl.arguments = 'myJob.sub';
rsl.directory = '/home/<USER>/';
```

```
rsl.stdout = 'myJob.stdout';
rsl.stderr = 'myJob.stderr';

% Make the Condor job submission
handle_Linux =
gd_condorsubmit(classAD_Linux,rsl,'server.domain.com');
handle_Windows =
gd_condorsubmit(classAD_Windows,rsl,'server.domain.com');
```

Notes
A valid proxy certificate is required to use this function.

The field names of the ClassAD and RSL structures should be lower case characters. ClassAD string variables should be in upper case characters, e.g. 'LINUX' not 'Linux', or 'WINNT51' not WinNT51'.

See also
`gd_jobsubmit`, `gd_submitunique`

gd_compute_version

Returns the current version of the Geodise Compute Toolbox

Syntax

```
version = gd_compute_version
```

Description

`version = gd_compute_version` returns the version of the current Geodise Compute Toolbox release as a string of the form MAJOR.MINOR.POINT.

See also

`README.txt`

gd_createproxy

Creates a Globus proxy certificate.

Syntax

```
gd_createproxy
```

Description

This command creates a Globus proxy certificate for the user's credentials at the location specified by the cog.properties file. The user is queried for the passphrase to their private key by a pop-up window.

See also

`gd_proxyinfo`, `gd_proxyquery`, `gd_certinfo`, `gd_destroyproxy`

gd_destroyproxy

Destroys the local copy of the user's Globus proxy certificate.

Syntax

```
gd_destroyproxy
```

Description

This command deletes the local copy of the Globus proxy certificate for the user's credentials at the location specified by the cog.properties file.

See also

`gd_createproxy`, `gd_proxyinfo`, `gd_certinfo`

gd_fileexists

Tests the existence of files and directories on a Globus resource.

Syntax

```
exists = gd_fileexists(host,remotefile)
```

Description

`exists = gd_fileexists(host,remotefile)` returns an integer `exists` indicating whether the file or directory specified by `remotefile` exists on the Globus server specified by the string `host`. The argument `exists` will equal 1 is the file exists on `host`, otherwise it will equal 0.

Example

```
result =
gd_fileexists('server.domain.com','/home/<USER>/test.dat');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_listdir`

gd_getfile

Retrieves a remote file using GridFTP.

Syntax

```
gd_getfile(host,remotefile,localfile)
gd_getfile(host,remotefile,localfile,filetype)
```

Description

This command retrieves a file from a remote server using GridFTP. The user must specify the remote file location on a remote server and the local destination for the file. The user can also specify the file type.

`gd_getfile(host,remotefile,localfile)` transfers the remote ASCII file `remotefile` from the machine `host`. The file is saved to the path and file specified by the string `localfile`.

`gd_getfile(host,remotefile,localfile,filetype)` as above except the string `filetype` sets the file transfer type. When `filetype` = 'ASCII' the file transfer type will be ASCII (this is the default setting), alternatively when `filetype` = 'binary' the file transfer type is set to binary.

Examples

The following command copies the file 'data2.dat' from the users home directory on the remote host 'server' to the local file 'C:\data1.dat'. The file is transferred as a binary file type.

```
gd_getfile('server.domain.com','data2.dat','C:\data1.dat',
'binary');
```

This example behaves as above except the file is copied from the subdirectory 'tmp' in the users home directory.

```
gd_getfile('server.domain.com','tmp/data2.dat','C:\data1.dat',
'binary');
```

The following example is similar to the first example except the file is copied from the subdirectory 'tmp' of the root directory on the remote machine.

```
gd_getfile('server.domain.com','/tmp/data2.dat','C:\data1.dat',
'binary');
```

Notes

A valid proxy certificate is required to use GridFTP. Suitable credentials may be required to transfer files from a remote server.

See also

gd_putfile, gd_createproxy

gd_jobkill

Kills a Globus GRAM job specified by a job handle.

Syntax

```
gd_jobkill(jobhandle)
```

Description

`gd_jobkill(jobhandle)` terminates the Globus job specified by the Globus job handle.

Notes

A valid proxy certificate for the correct user credentials is required to kill a GRAM job.

See also

`gd_createproxy`, `gd_jobsubmit`, `gd_jobstatus`

gd_jobpoll

Queries the status of a Globus GRAM job until complete.

Syntax

```
gd_jobpoll(jobhandle)
gd_jobpoll(jobhandle,interval)
isdone = gd_jobpoll(jobhandle,interval,maxtime)
```

Description

This command polls the status of a Globus GRAM job specified by the job handle until the job is complete. This function can be used to block the process of a Matlab script until a job has finished. If the job fails an error is thrown.

`gd_jobpoll(jobhandle)`where `jobhandle` is the handle to a Globus GRAM job.

`gd_jobpoll(jobhandle,interval)` where `jobhandle` is the handle to a Globus GRAM job and `interval` is the interval (in seconds) between polling the job handle.

`isdone = gd_jobpoll(jobhandle,interval,maxtime)` as above. The argument `maxtime` allows an upper limit (in seconds) to be placed on the period over which the job is polled. The return value `isdone` indicates whether the job handle returned the DONE state (1), or whether polling was aborted (0).

Notes

The state DONE returned by job handle does not necessarily indicate that the job completed successfully. A valid proxy certificate is required to query a GRAM job.

See also

`gd_jobstatus`, `gd_jobsubmit`, `gd_jobkill`

gd_jobstatus

Gets the status of a Globus GRAM job.

Syntax

```
status = gd_jobstatus(jobhandle)
```

Description

`status = gd_jobstatus(jobhandle)` returns the status of a Globus GRAM job, where `status`:

-1 is UNKNOWN
1 is PENDING
2 is ACTIVE
3 is DONE
4 is FAILED
5 is SUSPENDED
6 is UNSUBMITTED

Notes

A valid proxy certificate is required to query a GRAM job.

See also

`gd_createproxy`, `gd_jobsubmit`, `gd_jobkill`

gd_jobsubmit

Submits a compute job to a Globus GRAM job manager.

Syntax

```
jobhandle = gd_jobsubmit(RSL,host)
```

Description

This command submits the compute job described by a Resource Specification Language (RSL) string to a Globus server running a GRAM job manager. Upon a successful submission the command returns a job handle that may be used to query the status of, or terminate, the job.

`jobhandle = gd_jobsubmit(RSL,host)` where `RSL` is a string describing the submitted job, `host` is the name of the Globus server, and `jobhandle` is the handle for a successfully submitted job. An error is thrown if job submission is unsuccessful.

Example

```
jobhandle =
gd_jobsubmit('&(executable=/bin/date)','server.domain.com')
```

Notes

A valid proxy certificate is required to submit a GRAM job. For more information about RSL see http://www.globus.org/gram/.

See also

gd_createproxy, gd_jobkill, gd_jobstatus

gd_listdir

Lists the contents of a directory on a GridFTP server.

Syntax

```
files = gd_listdir(host)
files = gd_listdir(host,remotedir)
files = gd_listdir(host,remotedir,listhidden)
[files,details] = gd_listdir(...)
```

Description

`files = gd_listdir(host)` where `files` is a cell array containing the filenames of files in the user's home directory on the GridFTP server `host`.

`files = gd_listdir(host,remotedir)` where `files` is a cell array containing the filenames of files in the directory `remotedir` on the GridFTP server `host` (if `remotedir` is empty the contents of the user's home directory will be listed).

`files = gd_listdir(host,remotedir,listhidden)` the list of filenames will include hidden files if the argument `listhidden` is true (equal to 1). Otherwise the names of hidden files will not be returned (default behaviour).

`[files,details] = gd_listdir(host)` as above where `details` is a cell array containing structures that describe the details of the files and directories contained in the remote directory.

Notes

A valid proxy certificate is required to use GridFTP.

See also

`gd_putfile`, `gd_getfile`, `gd_createproxy`

gd_makedir

Creates a remote directory using GridFTP.

Syntax

```
gd_makedir(host,directory)
```

Description

`gd_makedir(host,directory)` Creates a directory specified by the string `directory` on the GridFTP server specified by the string `host`.

Notes

A valid proxy certificate is required to use GridFTP. Suitable credentials will be required to create a directory on a GridFTP server.

See also

`gd_getfile`, `gd_putfile`, `gd_rmdir`, `gd_rmfile`

gd_proxyinfo

Returns information about the user's proxy certificate.

Syntax

```
exists = gd_proxyinfo
[exists,subject] = gd_proxyinfo
```

Description

This command checks the existence of the user's proxy certificate and prints information to the screen. The command also returns the subject line of the proxy certificate.

`exists = gd_proxyinfo` where `exists` is 1 if the proxy certificate exists at the default location, otherwise 0.

`[exists,subject] = gd_proxyinfo` where `subject` is the subject line of the proxy certificate.

See also

`gd_proxyquery`, `gd_certinfo`, `gd_createproxy`, `gd_destroyproxy`

gd_proxyquery

Queries whether a valid proxy certificate exists.

Syntax

```
isvalid = gd_proxyquery
isvalid = gd_proxyquery(proxyattrib,minvalue)
```

Description

This command determines whether a valid proxy certificate exists for user's certificate. The strength or time remaining for the proxy certificate may also be queried. The location of the user's proxy certificate is specified by the cog.properties file.

`isvalid = gd_proxyquery` where `isvalid` is 1 if a valid proxy certificate exists at the default location, otherwise 0.

`isvalid = gd_proxyquery(proxyattrib,minvalue)` where `isvalid` is 1 if the proxy certificate meets the requirements of remaining lifetime or cryptographic strength, otherwise 0. If `proxyattrib` = 'time' the time remaining for the proxy certificate is queried against `minvalue` hours. If `proxyattrib` = 'strength' the cryptographic strength of the proxy certificate is queried against `minvalue` bits.

Example

The following example returns `isvalid` = 0 for a proxy certificate of strength 512.

```
isvalid = gd_proxyquery('strength',1024)
```

```
isvalid =

     0
```

See also

`gd_proxyinfo`, `gd_certinfo`, `gd_createproxy`, `gd_destroyproxy`

gd_putfile

Puts a file on a remote server using GridFTP.

Syntax

```
gd_putfile(host,localfile,remotefile)
gd_putfile(host,localfile,remotefile,filetype)
```

Description

This command puts a local file upon a remote server using GridFTP. The user must specify the remote server name, the local file path, and the remote file path. The user can also specify the filetype.

`gd_putfile(host,localfile,remotefile)` transfers the ASCII file `localfile` to the machine `host`. The file is saved to the path and file specified by the string `remotefile`.

`gd_putfile(host,localfile,remotefile,filetype)` as above except the string `filetype` sets the file transfer type. When `filetype` = 'ASCII' the file transfer type will be ASCII (this is the default setting), alternatively when `filetype` = 'binary' the file transfer type is set to binary.

Examples

The following command places the local file 'C:\data1.dat' on the remote host 'server' in the users home directory with the file name 'data2.dat'. The file is transferred as a binary file type.

```
gd_putfile('server.domain.com','C:\data1.dat','data2.dat',
'binary');
```

This example behaves as above except the file is placed in the existing subdirectory to the users home directory; 'tmp'.

```
gd_putfile('server.domain.com','C:\data1.dat','tmp/data2.dat',
'binary');
```

This example is similar to the first example except the file is placed in the subdirectory to the root directory; 'tmp'.

```
gd_putfile('server.domain.com','C:\data1.dat','/tmp/data2.dat',
'binary');
```

Notes

A valid proxy certificate is required to use GridFTP. Suitable credentials may be required to transfer files to remote servers.

See also

`gd_getfile`, `gd_createproxy`

gd_rmdir

Deletes an empty remote directory using GridFTP.

Syntax

```
gd_rmdir(host,remotedir)
```

Description

`gd_rmdir(host,remotedir)` Deletes an empty directory specified by the string `remotedir` on the GridFTP server specified by the string `host`.

Notes

A valid proxy certificate is required to use GridFTP. Suitable credentials will be required to delete a directory on a GridFTP server.

See also

`gd_getfile`, `gd_putfile`, `gd_makedir`, `gd_rmfile`

gd_rmfile

Deletes a remote file using GridFTP.

Syntax

```
gd_rmfile(host,remotefile)
```

Description

`gd_rmfile(host,remotefile)` Deletes the file specified by the string `remotefile` on the GridFTP server specified by the string `host`.

Notes

A valid proxy certificate is required to use GridFTP. Suitable credentials will be required to delete a file on a GridFTP server.

See also

`gd_getfile`, `gd_putfile`, `gd_makedir`, `gd_rmdir`

gd_rmuniquedir

Deletes a remote directory and its contents.

Syntax

```
gd_rmuniquedir(host,remotedir)
```

Description

This function deletes a remote directory and the files that it contains using GridFTP. The function will not delete the remote directory specified (or any of its contents) if the remote directory contains any sub-directories. This is a safety feature which is intended to mitigate the risks of wildcard deletions on a remote machine.

If the specified directory contains sub-directories an error will be thrown. Errors will also be thrown if the directory does not exist or if permission is denied to delete the directory or its contents.

`gd_rmuniquedir(host,remotedir)` where `host` is the name of the GridFTP server and `remotedir` is the name of the directory to be deleted.

See Also

`gd_rmdir`, `gd_rmfile`, `gd_submitunique`

gd_servermetrics

Performs a number of tests upon a list of Globus resources.

Syntax

```
testresults = gd_servermetrics(servers)
testresults = gd_servermetrics(servers,filename)
```

Description

`testresults = gd_servermetrics(servers)` will perform a suite of diagnostic tests on the Globus servers specified by `servers`. Where `servers` is a structure defining Grid resources which has the following mandatory fields:

`name`	Name of the Globus server.
`directory`	Name of the directory on the server in which the tests should be performed. This may be empty if no directory is specified.

The output structure `testresults` contains the following fields:

`name`	Name of the Globus server.
`directory`	Name of the directory on the server.
`authentication`	The elapsed time in milliseconds required for authentication, or -1 if failed.
`jobsubmission`	The elapsed time in milliseconds required for job submission, or -1 if failed.
`filetransfer`	The elapsed time in milliseconds required for file transfer, or -1 if failed.

`results = gd_servermetrics(servers)` as above where the results of the tests are output to the file specified by the string `filename`.

Example

The following example will run the diagnostic tests upon the servers 'server1.domain.com' and 'server2.domain.com'. The results of the tests will be output to the structure `testresults`, and to the file 'metrics.dat' in the current directory on the local machine.

```
servers(1).name = 'server1.domain.com';
servers(1).directory = '/home/<USER>/';
servers(2).name = 'server2.domain.com';
servers(2).directory = '';
testresults = gd_servermetrics(servers,'metrics.dat')
disp(testresults(1))
```

```
testresults =
1x2 struct array with fields:
    name
    directory
    authentication
    jobsubmission
    filetransfer

              name: 'server1.domain.com'
         directory: '/home/<USER>/'
    authentication: 141
     jobsubmission: 375
      filetransfer: 4234
```

Notes

A valid proxy certificate is required to use this function.

See also

gd_testauthentication, gd_testfiletransfer, gd_testjobsubmission, gd_timeauthentication, gd_timefiletransfer, gd_timejobsubmission

gd_submitunique

Submits a GRAM job to a unique working directory.

Syntax

```
[jobhandle,uniquedir] = gd_submitunique(RSL,host)
[jobhandle,uniquedir] = gd_submitunique(RSL,host,files)
[jobhandle,uniquedir] = gd_submitunique(RSL,host,files,
remotedir)
```

Description

This command creates a unique working directory on a Globus server, transferring files as required, and submits the compute job to the GRAM job manager. Upon a successful submission the command returns a job handle and the name of the unique directory.

`[jobhandle,uniquedir] = gd_submitunique(RSL,host)` where `RSL` is a string describing the submitted job, and `host` is the name of the Globus server. `jobhandle` is the handle for a successfully submitted job and `uniquedir` is the location of the working directory created on `host`.

`[jobhandle,uniquedir] = gd_submitunique(RSL,host,files)`
as above, where `files` is a cell array containing a list of the files to be transferred to the working directory on the `host`.

`[jobhandle,uniquedir] = gd_submitunique(RSL,host,files, remotedir)` as above, where `remotedir` is the directory on the `host` within which the unique working directory is created. `files` can be empty if no files are required.

Example

This command creates a directory '20040427T130607_643492' in the user's home directory on the machine `host`. The working directory in the user supplied RSL string is set to the unique directory.

```
[jobhandle,dirname] = gd_submitunique('&(executable=/bin/date)
(stdout="test.out")',host)
```

```
jobhandle =
https://host.domain.com:40001/15678/1083067567/

dirname =
20040427T130607_643492/
```

Notes

A valid proxy certificate is required to submit a GRAM job. For more information about RSL see http://www.globus.org/gram/.

See also

`gd_jobsubmit`, `gd_createproxy`, `gd_jobkill`, `gd_jobstatus`

gd_testauthentication

Tests authentication with a Globus resource.

Syntax

```
success = gd_testauthentication(host)
```

Description

`success = gd_testauthentication(host)` where `success` is the outcome of authentication with the Globus server `host`. The value of `success` is 1 on success and 0 on failure.

Example

```
success = gd_testauthentication('server.domain.com');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testfiletransfer`, `gd_testjobsubmission`, `gd_timeauthentication`, `gd_timefiletransfer`, `gd_timejobsubmission`

gd_testfiletransfer

Tests file transfer to a Globus resource.

Syntax

```
success = gd_testfiletransfer(host)
success = gd_testfiletransfer(host,remotedir)
```

Description

`success = gd_testfiletransfer(host)` where `success` is the outcome of the file transfer of a small file to the Globus GridFTP server `host`. The value of `success` is 1 on success and 0 on failure.

`success = gd_testfiletransfer(host,remotedir)` as above where the file will be transferred into the directory `remotedir` on `host`.

Example

```
remotedir = gd_testfiletransfer('server','/home/<USER>/');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testauthentication`, `gd_testjobsubmission`, `gd_timeauthentication`, `gd_timefiletransfer`, `gd_timejobsubmission`

gd_testjobsubmission

Tests the job submission to a Globus resource.

Syntax

```
success = gd_testjobsubmission(host)
success = gd_testjobsubmission(host,remotedir)
```

Description

`success = gd_testjobsubmission(host)` where `success` is the outcome of a job submission to the Globus server `host`. The value of `success` is 1 on success and 0 on failure.

`success = gd_testjobsubmission(host,remotedir)` as above where the job will run in the directory `remotedir` on `host`.

Example

```
success =
gd_testjobsubmission('server.domain.com','/home/<USER>/');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testauthentication`, `gd_testfiletransfer`, `gd_timeauthentication`, `gd_timefiletransfer`, `gd_timejobsubmission`

gd_timeauthentication

Times authentication to a Globus resource.

Syntax

```
time = gd_timeauthentication(host)
```

Description

`time = gd_timeauthentication(host)` where `time` is the elapsed time in milliseconds taken to authenticate with the Globus server `host`. If authentication fails `time` will return -1.

Example

```
time = gd_timeauthentication('server.domain.com');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testauthentication`, `gd_testfiletransfer`, `gd_testjobsubmission`, `gd_timefiletransfer`, `gd_timejobsubmission`

gd_timefiletransfer

Times file transfer to a Globus resource.

Syntax

```
time = gd_timefiletransfer(host)
time = gd_timefiletransfer(host,remotedir)
```

Description

`time = gd_timefiletransfer(host)` where `time` is the elapsed time in milliseconds taken to transfer a small file to the Globus GridFTP server `host`. If file transfer fails `time` will return -1.

`time = gd_timefiletransfer(host,remotedir)` as above where the file will be transferred into the directory `remotedir` on `host`.

Example

```
time = gd_timefiletransfer('server.domain.com','/home/<USER>/')
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testauthentication`, `gd_testfiletransfer`, `gd_testjobsubmission`, `gd_timeauthentication`, `gd_timejobsubmission`

gd_timejobsubmission

Times a job submission to a Globus resource.

Syntax

```
time = gd_timejobsubmission(host)
time = gd_timejobsubmission(host,remotedir)
```

Description

`time = gd_timejobsubmission(host,remotedir)` where `time` is the elapsed time in milliseconds taken to complete a job submission to the Globus server `host`. If the job submission fails `time` will return -1.

`time = gd_timejobsubmission(host,remotedir)` as above where the job will run in the directory `remotedir` on `host`.

Example

```
time =
gd_timejobsubmission('server.domain.com','/home/<USER>/');
```

Notes

A valid proxy certificate is required to use this function.

See also

`gd_testauthentication`, `gd_testfiletransfer`, `gd_testjobsubmission`, `gd_timeauthentication`, `gd_timefiletransfer`

gd_transferfile

Performs a third-party file transfer using GridFTP.

Syntax

```
gd_transferfile(host1,host2,remotefile1,remotefile2)
gd_transferfile(host2,host2,remotefile1,remotefile2,
filetype)
```

Description

`gd_transferfile(host1,host2,remotefile1,remotefile2)` transfers the file specified by the string `remotefile1` on the GridFTP server `host1` to the file specified by `remotefile2` on `host2`.

`gd_transferfile(host1,host2,remotefile1,remotefile2, filetype)` as above except the string `filetype` sets the file transfer type. When `filetype` = 'ASCII' the file transfer type will be ASCII (this is the default setting), alternatively when `filetype` = 'binary' the file transfer type is set to binary.

Examples

The following command will transfer a file called '/tmp/test1' from 'server1' to a file called '/tmp/test2' on 'server2' in ASCII mode,:

```
gd_transferfile('server1.domain.com','server2.domain.com',
'/tmp/test1','/tmp/test2')
```

See also

`gd_putfile`, `gd_getfile`, `gd_createproxy`

Geodise Database Toolbox

Introduction

The Geodise Database Toolbox consists of client and server tools which enable distributed users to easily manage, share and reuse their data from within the Matlab environment. Users with no database experience can integrate data management into their applications by calling the archive, query and retrieve functions provided by the toolbox. Any data files or Matlab variables can be stored in the Geodise archive. User defined Matlab structures specify additional descriptive information (metadata), which can be queried to easily locate data of interest. The Geodise Database Toolbox allows you to:

- Manage data from the local Matlab environment or remotely in scripts.
- Store files and variables with customized descriptive metadata.
- Organise related data into datagroups.
- Query over metadata to easily locate required data using functions or a GUI.
- Retrieve data based on logical data identities, no need to remember file locations.
- Share data with other distributed users by granting them access permissions.

There are a separate set of server side tools for the Geodise Database Toolbox. Variables and metadata are stored in an Oracle 9i and 10g database as XML, converted using the XML Toolbox. The Geodise Database Toolbox functions call data management services which utilise Grid, Web Service and database technologies with certificate based authentication and authorisation. The server side tools are not described in any detail in this document.

Tutorial

Getting started

Before using the Geodise Database Toolbox you need to register your details in the database by providing your certificate subject to an administrator, who will then assign you a username. To get your certificate subject call `gd_certinfo` from the Compute Toolbox.

```
>> subject = gd_certinfo
```

```
subject: C=UK,O=eScience,OU=Southampton,L=SeSC,CN=some user
issuer: C=UK,O=eScience,OU=Authority,CN=CA,E=ca-
operator@grid-support.ac.uk
start date: Tue Oct 07 13:00:31 BST 2003
end date: Wed Oct 06 13:00:31 BST 2004

subject =
/C=UK/O=eScience/OU=Southampton/L=SeSC/CN=some user
```

To setup the Database Toolbox call `gd_dbsetup` which will create a .geodise directory in your home directory and copy the necessary configuration files into it.

```
>> gd_dbsetup
```

You will be prompted for details of your file store host (where `gd_archive` will store your files). Set `hostname` to a Globus enabled server you have GridFTP permission on, and set `hostdir` to an existing directory on that server where files can be stored. These settings will be saved in *<home_dir>/.geodise/ClientConfig.xml*.

A valid proxy certificate is required to use the Database Toolbox functions, and this can be created using the function `gd_createproxy` from the Compute Toolbox.

```
>> gd_createproxy
```

A GUI will appear and prompt you for your certificate passphrase. Click the ‘Create’ button to generate the proxy certificate. When this is finished click ‘Cancel’ to close the GUI and press 'Enter' at the Matlab prompt.

See the Compute Toolbox Tutorial for more information on certificates and proxy certificates.

Archiving files

To archive a file from the local filesystem, first create a metadata structure containing some information that describes your file. This can be any combination of doubles, strings, arrays, cell arrays, complex numbers and substructures.

```
>> m.model.name = 'test_design';
>> m.model.params = [1 4.7 5.3];
>> m.product = 25.5431;
```

Add some standard information (`localName`, `format`, `comment`, `version` or `tree`) about the file.

```
>> m.standard.comment = 'Test design model file';
>> m.standard.version = '1.2.0';
```

The file can then be archived with the metadata.

```
>> fileID = gd_archive('C:\file.dat',m)
```

```
fileID =
file_dat_c6afa4b4-03cb-49a4-8c4e-008c38aae413
```

In addition to the optional metadata structure, `gd_archive` takes a string representing the path and filename of a local file. It stores this file on a remote file store (specified in `<user_home>/.geodise/ClientConfig.xml`). An ID is returned which is a unique handle that can be used to retrieve the file.

The metadata is stored in a database and can be queried to help you find relevant files. When the file is archived some additional metadata is automatically generated and stored in the `standard` substructure, regardless of whether user defined metadata was also provided. This consists of `localName` (the original name of the file), `byteSize`, `format`, `archiveDate`, `createDate` (when the original file was created/modified) and `userID`. See `gd_query` for further information on these fields. You can specify your own overriding values for `standard.localName` and `standard.format` if you prefer. You can also include the optional user defined metadata fields `comment`, `version` and `tree`. To help data organisation the `tree` field can be assigned a hierarchy string, similar to a directory path, e.g. `'myuserID/designs/testmodel'`.

Querying file metadata

To query file metadata pass a query string to the `gd_query` function. A query takes the form `'field = value'`, where `=` can be replaced by other comparison operators. More than one query condition can be included in the string using `&` to join them together. A call to `gd_query` returns a cell array of structures, one for each matching result.

```
>> result = gd_query('standard.version=1.2.0 & product>25.4')
```

```
result =
    [1x1 struct]
```

```
>> result{1}
```

```
ans =
    standard: [1x1 struct]
       model: [1x1 struct]
     product: 25.5431
```

`gd_display` is a convenient way to view your query results.

```
>> gd_display(result)
```

```
*** Content of the structure result{1} (Total structures: 1)
  standard.ID: file_dat_c6afa4b4-03cb-49a4-8c4e-008c38aae413
  standard.localName: file.dat
  standard.byteSize: 24
  standard.format: dat
  standard.createDate: 2004-09-15 15:25:33
  standard.archiveDate: 2004-10-07 11:03:10
  standard.userID: jlw
  standard.comment: Test design model file
  standard.version: 1.2.0
  standard.datagroups:
  model.name: test_design
  model.params:
    1.0000    4.7000    5.3000
  product: 25.5431
*** No more results. ***
```

It is possible to select which metadata fields are returned in the query results. This is done by passing a string containing a comma separated list of these fields as the third argument to `gd_query`. The second argument specifies that we want to query files, but is normally omitted because it is the default.

```
>> r = gd_query('product>25','file','standard.ID, model.*');
>> gd_display(r)
```

```
*** Content of the structure  ***
  standard.ID: file_dat_c6afa4b4-03cb-49a4-8c4e-008c38aae413
  model.name: test_design
  model.params:
    1.0000    4.7000    5.3000
```

To search for some text within a metadata value use the 'like' operator together with % to specify any characters, or _ to specify one character.

```
>> gd_query('standard.comment like %design m_del%');
```

The * wildcard can be used to represent an anonymous subfield, or any number of subfields if it appears at the beginning.

```
>> gd_query('*.name = test_design');
```

Use `gd_query` without any input arguments to start the Query Graphical User Interface (GUI), see Figure 2. You can set query conditions for standard metadata by selecting an operator (=, > etc) from the drop down list next to the relevant metadata item and typing in a value. Further query conditions for user defined metadata can be entered in the ‘Query custom metadata or variables’ text field. In the following text field you can enter a comma separated list to specify which metadata items are returned for each matching query result.

Click the ‘Submit Query’ button to run your query. The corresponding `gd_query` script command is displayed, followed by the results of the query.

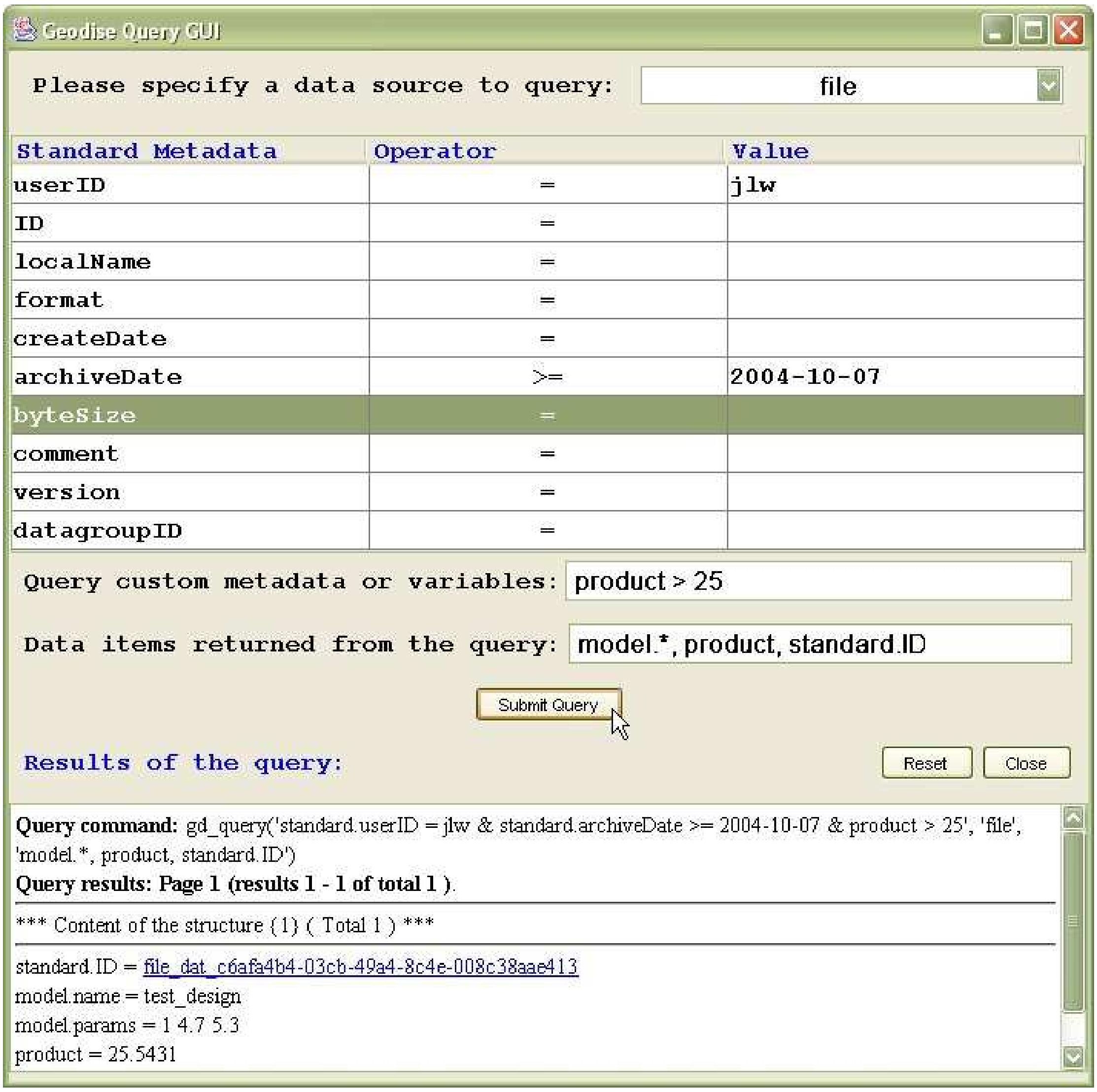

Figure 2 The Query GUI can be used to submit queries and view results.

Hyperlinks are provided in the query results for downloading and browsing data. Figure 3 demonstrates that a file can be downloaded by clicking on its standard.ID hyperlink. In the Save dialog box you can use the default file name value (original name of file) or specify a new file name. Browsing data is further discussed in the *Grouping data* section.

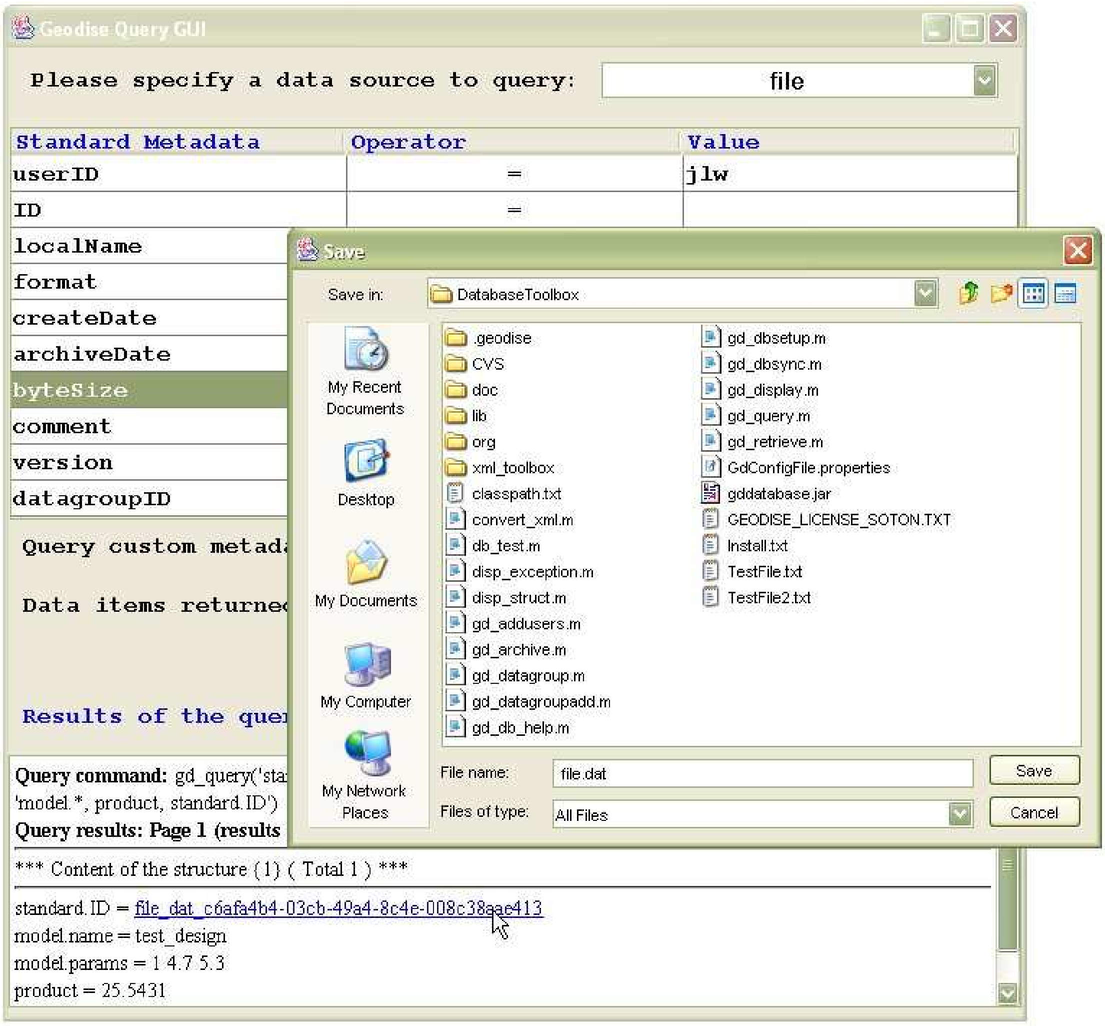

Figure 3 Click on a file's standard.ID link to download that file.

Retrieving files

A file can be retrieved to the local filesystem by specifying its unique ID. This string is returned by `gd_archive` when the file is archived, and also appears in the metadata query results as `standard.ID`.

```
>> ID = result{1}.standard.ID
```

```
ID =
file_dat_c6afa4b4-03cb-49a4-8c4e-008c38aae413
```

The file can be retrieved to a specific file location.

```
>> gd_retrieve(ID,'C:\filesdir\myfile.dat')
```

```
ans =
C:\filesdir\myfile.dat
```

Alternatively the file can be retrieved to a specified directory (the original file name is used).

```
>> gd_retrieve(ID,'C:\filesdir')
```

```
ans =
C:\filesdir\file.dat
```

Archiving, querying and retrieving Matlab variables

To archive a variable simply pass it to `gd_archive` with an optional metadata structure.

```
>> v.width = 12;
>> v.height = 6;
>> metadata.standard.comment = 'measurements variable';
>> varID = gd_archive(v,metadata);
```

It is possible to query the contents of an archived structure. Including 'var' as the second argument indicates that you want to query the contents of a variable (as opposed to the metadata of the variable).

```
>> result = gd_query('height=6','var');
>> gd_display(result{1})
```

```
*** Content of the structure  ***
  standard.varID: var_7c73ac04-cb90-4b28-988c-1e0562e4659d
  standard.datagroups:
  width: 12
  height: 6
```

The contents of the variable are returned along with a small subset of its metadata (`standard.varID` and `standard.datagroups`) which may be required for further queries. You can also query a variable's full metadata by including ‘varmeta’ as the second argument.

```
>> r = gd_query('standard.comment like measure%','varmeta');
>> gd_display(r{1})
```

```
*** Content of the structure  ***
  standard.ID: var_7c73ac04-cb90-4b28-988c-1e0562e4659d
  standard.archiveDate: 2004-10-07 11:35:19
  standard.userID: jlw
  standard.comment: measurements variable
  standard.datagroups:
```

A variable can be retrieved into the local Matlab workspace by specifying its unique ID. This string is returned when the variable is archived (e.g. `varID`) and also appears in the variable query results as `standard.varID` and in the metadata query results as `standard.ID`.

```
>> v2 = gd_retrieve(varID)
```

```
v2 =
     width: 12
    height: 6
```

Grouping data

Related data can be logically grouped together using a datagroup as follows:

Specify metadata that applies to the whole group.

```
>> dgmetadata.standard.comment = 'Group for experiment 123';
```

Call `gd_datagroup` to create a datagroup, giving it a name.

```
>> datagroupID=gd_datagroup('Experiment 123',dgmetadata);
```

Add archived files or variables to the datagroup.

```
>> gd_datagroupadd(datagroupID,fileID);
>> gd_datagroupadd(datagroupID,varID);
```

Archive a new file (with no metadata this time) and add it to the datagroup.

```
>> gd_archive('C:\anotherfile.txt',[],datagroupID);
```

The datagroup metadata now contains references to the files and variables it contains. Datagroup metadata can be queried by including ‘datagroup’ as the second argument.

```
>> r = gd_query('standard.datagroupname=Experiment 123',
'datagroup');
>> gd_display(r)
```

```
*** Content of the structure r{1} (Total structures: 1) ***
  standard.ID: dg_111385dd-44b8-4ac4-9ec3-f7f19af85e6e
  standard.datagroupname: Experiment 123
  standard.archiveDate: 2004-10-07 11:42:03
  standard.userID: jlw
  standard.comment: Group for experiment 123
  standard.datagroups:
  standard.subdatagroups:
  standard.files.fileID: file_dat_c6afa4b4-03cb-49a4-8c4e...
  standard.files.fileID: anotherfile_txt_8886aa7a-5464-48...
  standard.vars.varID: var_7c73ac04-cb90-4b28-988c-1e0562...
*** No more results. ***
```

Metadata for the files and variables also contain references to the datagroup(s) that they belong to, with a `standard.datagroups.datagroupID` field for each datagroup.

Datagroups can be added to other datagroups to create a hierarchy as follows:

```
>> parentDatagroupID = datagroupID;
>> childDatagroupID = gd_datagroup('child datagroup');
```

Add the child datagroup (also called a subdatagroup) to the parent datagroup.

```
>> gd_datagroupadd(parentDatagroupID,childDatagroupID);
```

Find all the datagroups that are in the parent datagroup.

```
>> children = gd_query(['standard.datagroups.datagroupID='
parentDatagroupID],'datagroup');
```

Find all the datagroups that contain the child datagroup.

```
>> parents = gd_query(['standard.subdatagroups.datagroupID='
childDatagroupID],'datagroup');
```

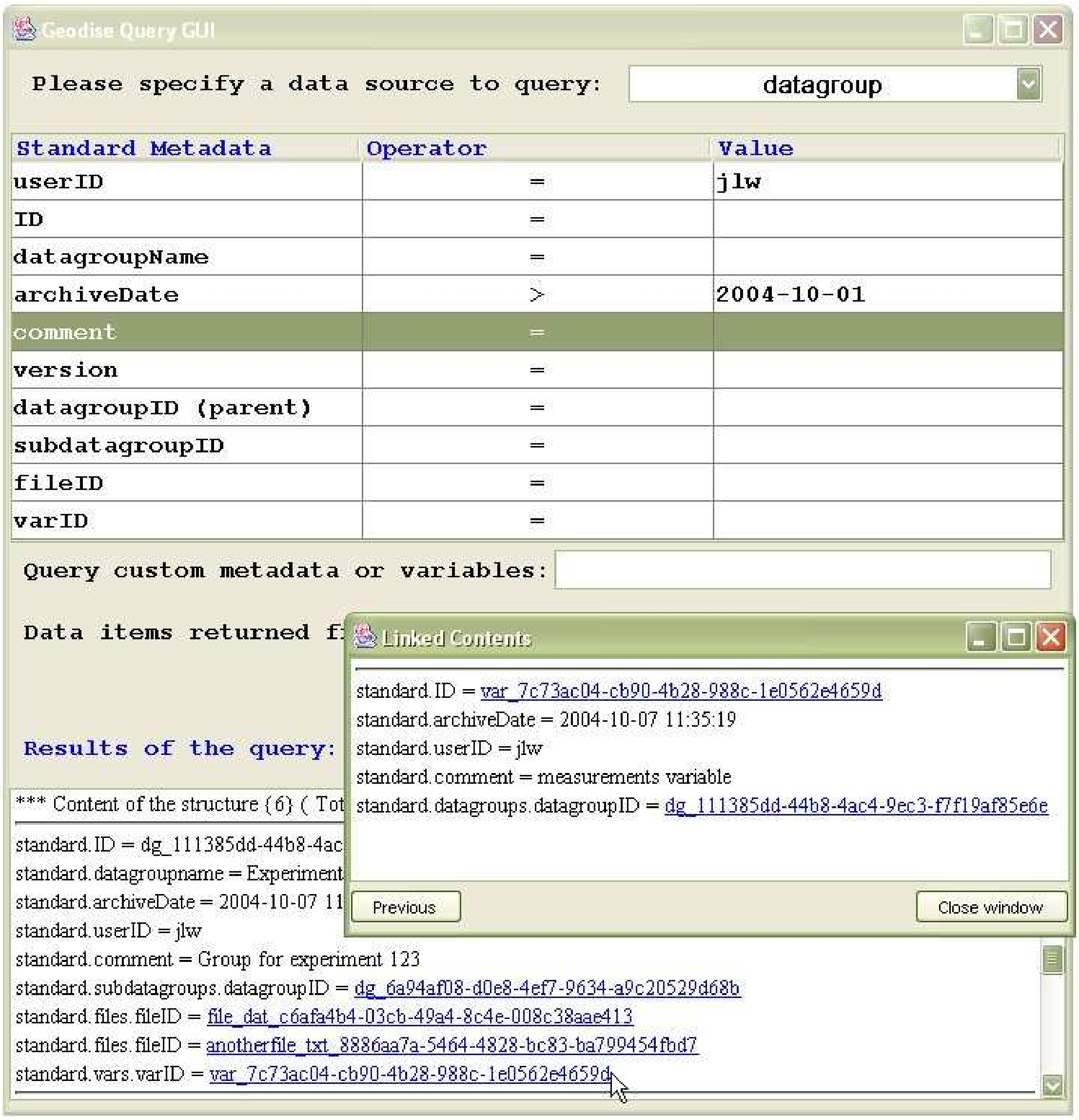

Figure 4 Using hyperlinks to browse between related data in the query GUI.

Using the Query GUI you can browse between related datagroups, files and variables by clicking on hyperlinks. In Figure 4 a query on datagroup metadata has been made by selecting datagroup from the drop down list at the top of the window, then specifying the query conditions. The matching datagroup shown in the figure has related subdatagroups, files and variables which are displayed as hyperlinks. Clicking on the standard.vars.varID link brings up a new window containing the metadata for that variable. Clicking on standard.ID in this window will display the contents of the variable itself.

Granting access to data.

The `gd_addusers` function allows you to grant other users permission to query particular files, variables and datagroups that you own. These users may also retrieve the variables to their local Matlab workspace and the files to their local filesystem (providing they have read permission for the appropriate directory on the Globus file server).

In the following example the user with username ‘bob’ is given access to an archived variable.

```
>> users = {'bob'};
>> gd_addusers(varID, users);
```

Access may also be granted as part of the metadata when a file or variable is archived, or when a datagroup is created.

```
>> m.access.users = {'bob'};
>> gd_archive('C:\file.dat',m);
```

Further information.
All of these functions have help information which can be viewed by using the help command in Matlab.

```
>> help gd_display
  gd_display Displays the results of a query (a cell of
  structures), or a single structure.

  gd_display(qresults) can be used to display a cell array of
  structures, e.g. the results of a call to the gd_query
  function. This is a convenient way of viewing structures to
  get an overview of their contents.

  gd_display(qresults{i}) displays the contents of a structure,
  e.g. a single result from a query, where i is the index of a
  structure in the cell array.
```

Further descriptions and examples for each function are available in the next section of this document.

Function Reference

gd_addusers

Grants an array of users or user groups permission to access some data (file, variable or datagroup).

Syntax

```
success = gd_addusers(ID,users)
success = gd_addusers(datagroupID,users)
success = gd_addusers(ID,groups,'groups')
success = gd_addusers(datagroupID,groups,'groups')
```

Description

`success = gd_addusers(ID,users)` grants other users permission to query or retrieve a file or variable, specified by its ID. A userID for each user should be provided in the `users` cell array. Alternatively a single user can be specified as a string.

`success = gd_addusers(datagroupID,users)` is similar but grants other users permission to query a datagroup, specified by its ID.

`success = gd_addusers(ID,groups,'groups')` grants a group of users permission to query or retrieve a file or variable, specified by its ID. A groupID for each user group should be provided in the `groups` cell array. Alternatively a single group can be specified as a string. Every registered user is a member of the built in group 'allusers' and other user groups can be set up by the database administrator.

`success = gd_addusers(datagroupID,groups,'groups')` is similar but grants a group of users permission to query a datagroup, specified by its ID.

The function returns 1 if successful, or 0 if failed (for example if one of the users already has access permission or does not exist). All valid userIDs or groupsIDs in the array will be granted permission, and a warning message will be displayed for any that fail.

Example

Grant users with user IDs `user1` and `user2` access to an archived file.

```
fileID = gd_archive('C:\file.dat');
users = {'user1','user2'};
gd_addusers(fileID,users);
```

Grant all registered users access to an archived file.

```
gd_addusers(fileID,'allusers','groups');
```

Notes

You must be the owner of the data to give others permission to access it.

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

Your certificate subject must have been added to the authorisation database.

See also

`gd_archive`, `gd_datagroup`, `gd_query`, `gd_createproxy`

gd_archive

Stores a file or variable with some metadata into the archive.

Syntax

```
ID = gd_archive(filename)
ID = gd_archive(filename,metadata)
ID = gd_archive(filename,metadata,datagroupID)
ID = gd_archive(v)
ID = gd_archive(v,metadata)
ID = gd_archive(v,metadata,datagroupID)
ID = gd_archive(v,metadata,datagroupID,'var')
```

Description

`ID = gd_archive(filename)` takes a string representing a filename and archives that file in a file store (specified in the ClientConfig.xml file). Some standard information about the file (metadata) is automatically generated and can be later queried with `gd_query`. A unique identifier (`ID`) for the archived file is returned which can be used to retrieve the file with `gd_retrieve`.

`ID = gd_archive(filename,metadata)` archives a file with some user defined `metadata` which can later be queried with `gd_query`. Standard metadata about the file is also generated.

`ID = gd_archive(filename,metadata,datagroupID)` archives a file and adds it to a datagroup specified by `datagroupID`. A datagroup is used to group together a collection of related files, variables and other datagroups, see `gd_datagroup` and `gd_datagroupadd`. To specify a `datagroupID` without including user defined file metadata, set `metadata` to empty [].

`ID = gd_archive(v)` takes a variable and archives it in a database (accessible via the webservices specified in the ClientConfig.xml file). `v` can be of type char, double, complex, struct, sparse, cell array, or logical. Some standard metadata about the variable is generated automatically and can be later queried with `gd_query`. A unique identifier (`ID`) for the archived variable is returned which can be used to retrieve the variable to the workspace with `gd_retrieve`. A variable can also be assigned user defined `metadata` and added to a datagroup by supplying a `datagroupID` in the same way as a file.

`ID = gd_archive(v,metadata,datagroupID,'var')` should be used when archiving a variable that is a string (char). If `v` has any other type it will be automatically detected, but when it is a string 'var' must be specified to indicate it is a variable and not a filename. If there is no user defined `metadata` or `datagroupID`, set them to empty [].

Input Arguments

`metadata` The `metadata` structure can contain any combination of named variables, matrices and substructures (char, double, complex, struct, sparse, cell or logical) necessary to describe the data. However, there are two special substructures, `standard` and `access`, which may only contain certain values.

Some metadata is automatically generated (even when no metadata is passed to the function) and stored in the `standard` substructure of `metadata`. For files and variables this consists of `ID`, `userID` and `archiveDate`, and for files only: `byteSize`, `format`, `localName` (the original name of the file) and `createDate` (when the original file was created/modified). Optional `comment`, `version` and `tree` fields can be added to `standard` and overriding values for `standard.localName` and `standard.format` can also be specified. The `tree` field is a string which can be used to represent a user defined hierarchy for the data, similar to a directory path, e.g. `'myuserID/designs/testmodel'`. See `gd_query` for further information on these fields. Any other fields set in the `standard` substructure will be overwritten or removed.

The `access` substructure of `metadata` controls who may query and retrieve the data. The person who archived the data automatically has access to it and does not need to be added. `access` can contain two fields, each of which can be a single string or a cell array of strings:

`users`	User ID strings specifying which users may access the data.
`groups`	Group ID strings specifying which groups of users may access the data (currently a group must be created in the database by an administrator).

Examples

Archive a file with no user defined metadata.

```
ID = gd_archive ('C:\file.dat')
```

```
ID =
file_dat_ce868f40-8de0-445e-8ae5-36c05eec25a9
```

Archive a file with some metadata, `m` (user defined metadata and a standard comment), and give access permission to user1 and user2.

```
m.model.name = 'test_design';
m.params = [1 4.7 5.3];
m.iterations = 9000;
m.standard.comment = 'Comment about file';
m.access.users = {'user1','user2'};
gd_archive('C:\file.dat',m);
```

Archive a file and add it to a datagroup, using [] to indicate no user defined metadata.

```
datagroupID = gd_datagroup('design opt 2004-09-03');
gd_archive('C:\file.dat',[],datagroupID);
```

Archive a structure with some user defined metadata.

```
v.width = 12;
v.height = 6;
m.standard.comment = 'measurement variables';
gd_archive(v,m);
```

Notes

A valid proxy certificate is required to archive a file or variable (see `gd_createproxy` from the Geodise Compute Toolbox).

You must have access to the host machine the files will be archived on. Your certificate subject must be added to the gridmap file on the host and to the authorisation database.

See also

`gd_addusers`, `gd_retrieve`, `gd_query`, `gd_datagroup`, `gd_datagroupadd`, `gd_createproxy`

gd_datagroup

Creates a new datagroup, used to group together archived files, variables and subdatagroups.

Syntax

```
datagroupID = gd_datagroup(datagroupname)
datagroupID = gd_datagroup(datagroupname,metadata)
datagroupID = gd_datagroup(datagroupname,metadata,
'monitor')
```

Description

`datagroupID = gd_datagroup(datagroupname)` creates a new, empty datagroup with a datagroup name. The `datagroupname` argument can act as a user defined identifier for the datagroup, although it does not have to be unique. Some standard information about the datagroup (metadata) is also generated which can be later queried with `gd_query`. A unique identifier (`datagroupID`) is returned which can then be used to add files and variables to the datagroup while they are being archived with `gd_archive`. Files, variables and other datagroups already in the archive can be added to a datagroup with `gd_datagroupadd`.

`datagroupID = gd_datagroup(datagroupname,metadata)` creates a new, empty datagroup with a datagroup name and some user defined metadata which can later be queried with `gd_query`. Standard metadata about the datagroup is also generated.

`datagroupID = gd_datagroup(datagroupname,metadata, 'monitor')` is useful for monitoring a group of data produced by a computational job. It is similar to an ordinary datagroup but stores extra index information that allows a user of `gd_query` to easily find the datagroup associated with their most recent job, or the most recent job meeting certain metadata criteria. This functionality is provided for convenience so that the user does not have to remember any particular field names, values, or what time the datagroup was created.

Input Arguments

`metadata` The `metadata` structure can contain any combination of named variables, matrices and substructures (char, double, complex, struct, sparse, cell or logical) necessary to describe the datagroup. However, there are two special substructures, `standard` and `access`, which may only contain certain values.

Some metadata is automatically generated (even when no metadata is passed to the function) and stored in the `standard` substructure of `metadata`. For datagroups this consists of `ID`, `userID` and `archiveDate`. Optional `comment`, `version` and `tree` fields can also be added to `standard`. The `tree` field is a string which can be used to represent a user defined hierarchy for the data, similar to a directory path, e.g. `'myuserID/designs/testmodel'`. See `gd_query` for further information on these fields. Any other fields set in the `standard` substructure will be overwritten or removed.

The `access` substructure of `metadata` controls who may query the datagroup. The person who created the datagroup automatically has access to it and does not need to be added. `access` can contain two fields, each of which can be a single string or a cell array of strings:

`users`	User ID strings specifying which users may access the datagroup.
`groups`	Group ID strings specifying which groups of users may access the datagroup (currently a user group must be created in the database by an administrator).

Examples

Create a datagroup with some metadata, `m` (user defined metadata and a standard comment), and give access permission to user1 and user2.

```
m.expnum = 123;
m.standard.comment = 'Data for experiment 123';
m.access.users = {'user1','user2'};
datagroupID = gd_datagroup('design opt 2004-09-03',m)
```

```
datagroupID =
dg_ce868f40-8ds0-455e-9ae5-36c05epc25a9
```

Add a file to the datagroup when it is archived.

```
gd_archive('C:\file.dat', [], datagroupID);
```

Add a variable to the datagroup after it has been archived.

```
v.width = 12;
varID = gd_archive(v);
gd_datagroupadd(datagroupID,varID);
```

Create a monitored datagroup and find it with a query.

```
monID = gd_datagroup('design opt 2004-09-03 job',[],'monitor')
gd_datagroupadd(monID,varID);
gd_query('standard.jobIndex = max','monitor');
```

Further examples are given in `gd_datagroupadd` and `gd_query`.

Notes

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

Your certificate subject must have been added to the authorisation database.

See also

`gd_datagroupadd`, `gd_archive`, `gd_retrieve`, `gd_query`, `gd_createproxy`

gd_datagroupadd

Adds an archived file, variable or subdatagroup to a datagroup.

Syntax

```
success = gd_datagroupadd(datagroupID,ID)
success = gd_datagroupadd(datagroupID,subdatagroupID)
```

Description

`success = gd_datagroupadd(datagroupID,ID)` adds a file or variable to a datagroup. The datagroup is specified by its unique identifier `datagroupID` and the identifier of the file or variable to add is specified with `ID`. The datagroup must have been created with [gd_datagroup]() and the file or variable must have been archived using [gd_archive](). The function returns 1 if successful, or 0 if failed (for example if the datagroup does not exist).

`success = gd_datagroupadd(datagroupID,subdatagroupID)` adds a datagroup (`subdatagroupID`) to another datagroup (`datagroupID`). The datagroup to be added is known as a subdatagroup. Both datagroups must have been created with [gd_datagroup]().

Examples

Add a file and a variable to a datagroup after they have been archived.

```
datagroupID = gd_datagroup('design opt 2004-09-03');

fileID = gd_archive('C:\file.dat');
gd_datagroupadd(datagroupID,fileID);

v.width = 12;
varID = gd_archive(v);
gd_datagroupadd(datagroupID,varID);
```

Add a datagroup to another datagroup

```
datagroupID = gd_datagroup('parent datagroup');
subdatagroupID = gd_datagroup('child datagroup');
gd_datagroupadd(datagroupID,subdatagroupID);
```

Notes

Only the owner of a datagroup can add data to it.

Attempting to add a file, variable or subdatagroup twice to the same datagroup will cause an error.

Attempting to add a datagroup to another datagroup that it is already the parent or ancestor of will cause an error. E.g. If datagroup `b` is added to datagroup `a`, and datagroup `c` is added to `b`, then `a` cannot be added to `b` or `c`.

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

Your certificate subject must have been added to the authorisation database.

See also

`gd_datagroup`, `gd_archive`, `gd_retrieve`, `gd_query`, `gd_createproxy`

gd_dbsetup

Creates and populates the .geodise directory with configuration files.

Syntax

```
gd_dbsetup
gd_dbsetup(hostprompt)
```

Description

`gd_dbsetup` creates a .geodise directory in the user's home directory if one does not exist then copies the necessary configuration files into it. The user is prompted to configure the name of the Globus server and directory where gd_archive will store data files and this information is saved in .geodise/ClientConfig.xml.

Example locations for the .geodise directory are:

Windows	C:\Documents and Settings\your_username\.geodise
Linux	$HOME/.geodise

`gd_dbsetup(0)` creates a .geodise directory as above but does not prompt for the name of the Globus server and directory where gd_archive will store data files, using default values instead. The default settings are either taken from a previous copy of ClientConfig.xml in .geodise or from ClientConfig.xml in the distribution.

Notes

The file .geodise/ClientConfig.xml can be edited to manually configure settings such as which Globus file store to archive files on, see installation document for more information.

gd_db_help

Gives an overview of functions and files in the Geodise Database Toolbox.

Syntax

```
gd_db_help
```

Description

`gd_db_help` displays a summary of the functions and files included in the Geodise Database Toolbox.

gd_db_version

Gets the Geodise Database Toolbox version number.

Syntax

```
gd_db_version()
```

Description

`gd_db_version()` returns the version of the current Geodise Database Toolbox for Matlab release as a string of the form MAJOR.MINOR.POINT.

gd_display

Displays the results of a query (a cell of structures), or a single structure.

Syntax

```
gd_display(qresults)
gd_display(qresults{i})
```

Description

`gd_display(qresults)` can be used to display a cell array of structures, e.g. the results of a call to the `gd_query` or `gd_querydeleted` function. This is a convenient way of viewing structures to get an overview of their contents.

`gd_display(qresults{i})` displays the contents of a structure, e.g. a single result from a query where `i` is the index of a structure in the cell array.

Example

Display all the results from a query.

```
r = gd_query('iterations = 9000');
gd_display(r);
```

```
*** Content of structure r{1} (Total structures: 2) ***
  standard.ID: file_dat_66830074-e749-4de0-b976-61f4d32
  standard.localName: file.dat
  standard.byteSize: 245
  standard.format: dat
  standard.createDate: 2004-08-23 10:40:33
  standard.archiveDate: 2004-09-03 15:25:45
  standard.userID: jlw
  standard.comment: Comment about file
  standard.datagroups:
  model.name: test_design
  params:
    1.0000    4.7000    5.3000
  iterations: 9000
Press ENTER to continue ..., q to quit:
```

To display just one result from a query use that result's index.

```
gd_display(r{1});
```

See also

gd_query, gd_querydeleted

gd_markfordeletion

Marks data for deletion from the archive.

Syntax

```
marktotal = gd_markfordeletion(ID)
marktotal = gd_markfordeletion(IDs)
```

Description

`marktotal = gd_markfordeletion(ID)` takes an ID string and marks the corresponding file, variable or datagroup for deletion from the archive. The function returns 1 if successful or 0 if failed, in which case the reason is displayed in a warning message (for example the ID does not exist). Once data is marked for deletion it is no longer visible using `gd_query`, `gd_retrieve` or any other Database Toolbox functions (apart from `gd_unmarkfordeletion` or `gd_querydeleted`). The data is then eligible for permanent deletion by an administrator.

`marktotal = gd_markfordeletion(IDs)` is similar but takes a cell of ID strings and marks the corresponding files, variables and datagroups for deletion from the archive. The function returns `marktotal`, the total number of IDs successfully marked for deletion, and displays warning messages for those that were unsuccessful.

Examples

Mark a single file for deletion from the archive.

```
ID = gd_archive('C:\file.dat');
marktotal = gd_markfordeletion(ID)
```

```
marktotal =
     1
```

Query variable metadata, and then mark the corresponding variables for deletion from the archive.

```
q = 'standard.archiveDate > 2004-12-01 & a.b < -500';
qresults = gd_query(q, 'varmeta');
for i=1:size(qresults,2)
```

```
    IDs{i} = qresults{i}.standard.ID;
end
marktotal = gd_markfordeletion(IDs)
```

```
marktotal =
     5
```

Notes

Only the owner of the data (the person who archived it) can mark it for deletion.

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

See also

`gd_unmarkfordeletion`, `gd_querydeleted`, `gd_createproxy`

gd_query

Performs queries over metadata or Matlab structures stored in the archive.

Syntax

```
gd_query
qresults = gd_query(query)
qresults = gd_query(query,datasource)
qresults = gd_query(query,datasource,resultfields)
```

Description

`gd_query` with no input arguments starts the query GUI, a Graphical User Interface for querying metadata and structures which also allows hyperlink browsing between related data. See the Geodise Database Toolbox Tutorial for more details.

`qresults = gd_query(query)` sends a `query` string to the database requesting all file metadata that meets the criteria specified in the string. A query takes the form `'field = value'`, where `=` can be replaced by other comparison operators. More than one query condition can be included in the string using `&` to join them together. The function returns a cell array of metadata structures, one for each matching result.

`qresults = gd_query(query,datasource)` sends a `query` string to the database requesting matching archived structures or metadata of a certain type, depending on the value of the `datasource` string. To query metadata set `datasource` to ‘file’ (default), ‘varmeta’ (metadata about variables), ‘datagroup’ or ‘monitor’. A cell array of matching structures is returned, one for each result. To query variables stored in the database set `datasource` to ‘var’. In this case the function will return a cell array of matching variables. The only variables that can be queried in this way are structures, because they contain named fields that can be searched for.

`qresults = gd_query(query,datasource,resultfields)` sends a `query` string to the database as above but only returns selected fields for each matching result. The `resultfields` string is a comma separated list indicating which fields should be returned for each result, for example just the `standard.ID` fields. The default, `*`, returns all fields. To view the query results, use function `gd_display`.

Input Arguments

`query` A query takes the form `'field = value'` where `field` is the name of a field in the archived metadata/variable structure, for example `iterations` or `standard.ID` (dot notation is used to access the subfields of a structure). The `value` is an alphanumeric value the field should contain. The operator `&` (meaning 'and') can be used to specify more than one search condition.

The following operators can be used to compare fields with values:

`=`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`like`	Similar to
`not like`	Not similar to

Similarity matches with `like` and `not like` use the following wildcards:

`_`	Matches any single character.
`%`	Matches any string of any length (including 0).

For example, `'standard.localName like %dat%'` will match strings containing the phrase 'dat', and `'model.name like _est%'` will match strings starting with any character followed by 'est' and then any string. To search for the characters _ and %, precede them with the \ escape character.

The operators do case sensitive comparison when used with string values. To make an operator case insensitive surround it with two # characters. For example, #=#, #!=#, #like#, #not like#.

Another wildcard, *, provides flexibility in describing the field path. For example, `model.name` can be replaced by `*.name` for a less specific search.

In addition to user defined metadata fields, the following standard metadata fields can be queried:

`standard.ID`	ID that uniquely identifies a file, variable or datagroup.
`standard.datagroupname`	Name of datagroup. Only used when querying datagroups.
`standard.localName`	Name of a local file before it was archived.
`standard.byteSize`	Size in bytes of a file.
`standard.format`	Format of file (default is its extension).
`standard.createDate`	Date the file was created/modified.
`standard.archiveDate`	Date the file or variable was archived, or the datagroup was created.
`standard.userID`	ID of the user who archived the data or created the datagroup.
`standard.comment`	Comment about the file, variable or datagroup.
`standard.version`	User defined version number for the file, variable or datagroup.
`standard.tree`	String representing a user defined data hierarchy, similar to a directory path.
`standard.files.fileID`	Each file in a datagroup.
`standard.vars.varID`	Each variable in a datagroup.
`standard.subdatagroups.datagroupID`	Each subdatagroup in a datagroup.
`standard.datagroups.`	Each datagroup a file, variable or

`datagroupID`	subdatagroup belongs to.

Datagroups are collections that can contain files, variables or other datagroups, see `gd_datagroup` and `gd_datagroupadd`.

The fields in an archived structure variable can also be queried in conjunction with the standard metadata fields for that variable.

`datasource` The data source indicates which type of data to query, and can be specified by one of the following strings (the default `datasource` value is 'file'):

‘file’	Metadata about files.
‘datagroup’	Metadata about datagroups.
‘monitor’	Metadata about monitorable datagroups.
‘varmeta’	Metadata about Matlab variables.
‘var’	Matlab variables.

A datagroup that was created with the ‘monitor’ flag can be queried as an ordinary datagroup, or as a collection of data about a computational job, by setting `datasource` to ‘monitor’. This provides a quick and easy query mechanism for finding a user’s most recent job, or the latest job meeting certain other metadata criteria. It is provided for convenience so that the user does not have to remember any particular field names, values, or what time the datagroup was created. In addition to `standard.ID`, `standard.userID` and user defined metadata, the following standard metadata can be used together with ‘monitor’ to query a job monitoring datagroup.

`standard.jobIndex`	Job index. Special query syntax `jobIndex = max` gets the highest index (most recent job).
`standard.jobName`	Name of job (same as datagroupname).
`standard.startDate`	Start date of job (when the datagroup was created).

Examples

Query file metadata to find files archived on or after 1st September 2004 where iterations = 9000. A datasource argument is not required because 'file' is the default.

```
q = 'standard.archiveDate>=2004-09-01 & iterations=9000';
qresults = gd_query(q)
```

```
qresults =
    [1x1 struct]    [1x1 struct]
```

```
disp(qresults{1});
```

```
       standard: [1x1 struct]
          model: [1x1 struct]
         params: [1 4.7000 5.3000]
     iterations: 9000
```

```
disp(qresults{1}.standard.archiveDate);
```

```
2004-09-03 15:25:45
```

See `gd_display` for an example of displaying the full contents of query results.

Query to find files which have a name field equal to 'test_design' in their metadata and only return the fields standard.ID and params.

```
q = '*.name = test_design';
qresults = gd_query(q,'file','standard.ID, params');

disp(qresults{1})
```

```
     standard: [1x1 struct]
       params: [1 4.7000 5.3000]
```

Query to find datagroups with comments containing the text 'experiment'.

```
q = 'standard.comment like %experiment%';
gd_query(q,'datagroup');
```

Query variable metadata to find the metadata for all variables that are in a particular datagroup.

```
q = 'standard.datagroups.datagroupID= dg_ce868f40-8ds0-455...';
gd_query(q,'varmeta');
```

Query variables to find structures where field width is between 9 and 14 inclusive.

```
gd_query('width >= 9 & width <= 14', 'var');
```

Find files that have a comment in their metadata, using "" (two double quotes) to indicate an empty value.

```
gd_query('standard.comment != ""');
```

Find the latest job monitoring datagroup then find the latest job monitoring datagroup which matches some other criteria.

```
m.modelver = 0.6; m2.modelver = 0.71;
gd_datagroup('design model job xyz',m,'monitor');
gd_datagroup('design model job abc',m,'monitor');
gd_datagroup('design model job 999',m2,'monitor');

r1 = gd_query('standard.jobIndex = max','monitor');
r1{1}.standard.jobName
```

```
ans =
design model job 999
```

```
r2 = gd_query('standard.jobIndex = max & modelver <= 0.6',
'monitor');
r2{1}.standard.jobName
```

```
ans =
design model job abc
```

Notes

When querying standard date information (`archiveDate` or `createDate`), specify the date/time using the International Standard Date and Time Notation (ISO 8601) which is: "YYYY-MM-DD hh:mm:ss" (hh:mm:ss is optional).

Only results for data you are authorised to access will be returned. Function `gd_addusers` can be used to grant access to others.

A valid proxy certificate is required to query the database (see `gd_createproxy` from the Geodise Compute Toolbox).

Your certificate subject must have been added to the authorisation database.

See also

`gd_display`, `gd_createproxy`, `gd_archive`, `gd_retrieve`, `gd_datagroup`, `gd_datagroupadd`, `gd_addusers`

gd_querydeleted

Performs queries over metadata or Matlab structures marked for deletion in the archive.

Syntax

```
qresults = gd_querydeleted(query)
qresults = gd_querydeleted(query,datasource)
qresults = gd_querydeleted(query,datasource,resultfields)
```

Description

`qresults = gd_query(query,datasource,resultfields)` has exactly the same functionality as `gd_query` but performs the queries over data that has been marked for deletion using the `gd_markfordeletion` function. `gd_querydeleted` returns a cell array of metadata structures, one for each matching result. The `standard.ID` field in each result can be passed to the `gd_unmarkfordeletion` function, which can recover data that was mistakenly marked for deletion. This function will only return results for data that has not already been permanently deleted from the archive by an administrator.

A brief description of the input arguments is given here, see `gd_query` for further information. To view the query results, use function `gd_display`.

Input Arguments

`query`	A query takes the form `'field = value'` where `field` is the name of a field in the archived metadata/variable structure, for example `iterations` or `standard.ID` (dot notation is used to access the subfields of a structure). The `value` is an alphanumeric value the field should contain. The operator `&` (meaning 'and') can be used to specify more than one search condition.
`datasource`	The data source indicates which type of data to query, and can be specified by one of the following strings (the default `datasource` value is 'file'):

'file'	Metadata about files.
'datagroup'	Metadata about datagroups.

‘monitor’	Metadata about monitorable datagroups.
‘varmeta’	Metadata about Matlab variables.
‘var’	Matlab variables.

`resultfields` The resultfields string is a comma separated list indicating which fields should be returned for each result, for example just the `standard.ID` fields. The default, `*`, returns all fields.

Examples

Query variable metadata that has been marked for deletion, and then unmark the corresponding variables so that they are no longer eligible for deletion from the archive.

```
q = 'standard.archiveDate > 2004-12-01 & a.b < -500';
qresults = gd_querydeleted(q, 'varmeta');
for i=1:size(qresults,2)
    IDs{i} = qresults{i}.standard.ID;
end
unmarktotal = gd_unmarkfordeletion(IDs)
```

```
unmarktotal =
     5
```

Notes

When querying standard date information (`archiveDate` or `createDate`), specify the date/time using the International Standard Date and Time Notation (ISO 8601) which is: "YYYY-MM-DD hh:mm:ss" (hh:mm:ss is optional).

Only results for data marked for deletion and owned by the user (i.e. data the user archived/created) will be returned.

If the marked data has been permanently deleted from the archive by an administrator it cannot be queried.

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

Your certificate subject must have been added to the authorisation database.

See also

`gd_unmarkfordeletion`, `gd_markfordeletion`, `gd_display`, `gd_query`, `gd_createproxy`

gd_retrieve

Retrieves a file, variable or metadata from the archive to the local machine.

Syntax

```
filename = gd_retrieve(ID,filename)
filename = gd_retrieve(ID,directory)
filename = gd_retrieve(ID,filename,[],'overwrite')
filename = gd_retrieve(ID,directory,[],'overwrite')
v = gd_retrieve(ID)
metadata = gd_retrieve(ID,[],'metadata')
```

Description

The `ID` needed to retrieve some data can be found in its metadata as `standard.ID`, and is also returned by `gd_archive`.

`filename = gd_retrieve(ID,filename)` retrieves a file from the archive based on its unique identifier (`ID`) and saves it to a local file specified by the `filename` string. The function returns the retrieved file's new location as a string, which is equal to the `filename` argument in this case. If a file exists with the same name a prompt will appear asking whether to overwrite it.

`filename = gd_retrieve(ID,directory)` retrieves a file from the archive based on its unique identifier (`ID`) and saves it to a local directory specified by the directory string. The original name of the file will be used, which is determined by the `standard.localName` property in the file's metadata, see `gd_archive` and `gd_query`. The function returns the retrieved file's new location as a string.

`filename = gd_retrieve(ID,filename,[],'overwrite')` retrieves a file and saves it to the local file system. If a file exists with the same name it will be overwritten without prompting. This is also the case when a `directory` is given as the second argument.

`v = gd_retrieve(ID)` returns a variable from the archive to the Matlab workspace based on its unique identifier (`ID`).

`metadata = gd_retrieve (ID,[],'metadata')` will return a metadata structure which corresponds to the file, variable or datagroup identified by `ID`. This is a shortcut, as the same result can be achieved using gd_query.

Examples

Retrieve a file and save it with a specific file name.

```
fileID = gd_archive('C:\file.dat');
gd_retrieve(fileID,'C:\filesdir\myfile.dat')
```

```
ans =
C:\filesdir\myfile.dat
```

Retrieve a file to a directory and use its original name.

```
gd_retrieve(fileID,'C:\filesdir')
```

```
ans =
C:\filesdir\file.dat
```

Retrieve a variable to the Matlab workspace.

```
v.width = 12;
v.height = 6;
varID = gd_archive(v);
x = gd_retrieve(varID)
```

```
x =
     width: 12
    height: 6
```

Retrieve some metadata about a file.

```
m = gd_retrieve(fileID,[],'metadata');
```

Notes

You can only retrieve data that you archived or that someone else has given you permission to access.

A valid proxy certificate is required to retrieve a file, variable or metadata (see `gd_createproxy` from the Geodise Compute Toolbox).

You must have access to the host machine the files will be retrieved from. Your certificate subject must be added to the gridmap file on the host and to the authorisation database.

See also

`gd_archive`, `gd_datagroup`, `gd_datagroupadd`, `gd_query`, `gd_createproxy`

gd_unmarkfordeletion

Recovers data marked for deletion, if it has not been permanently deleted by an administrator.

Syntax

```
unmarktotal = gd_unmarkfordeletion(ID)
unmarktotal = gd_unmarkfordeletion(IDs)
```

Description

`unmarktotal = gd_unmarkfordeletion(ID)` takes an ID string and unmarks the corresponding file, variable or datagroup so it is no longer marked for deletion from the archive. This is a safety measure to recover data that was mistakenly marked for deletion. This function is only applicable for data that has not already been permanently deleted from the archive by an administrator. The function returns 1 if successful or 0 if failed, in which case the reason is displayed in a warning message (for example the ID does not exist). If data is successfully unmarked it is visible again to `gd_query`, `gd_retrieve` and other Database Toolbox functions.

`unmarktotal = gd_unmarkfordeletion(IDs)` is similar but takes a cell of ID strings and unmarks the corresponding files, variables and datagroups so they are no longer marked for deletion from the archive. The function returns `unmarktotal`, the total number of IDs successfully unmarked for deletion, and displays warning messages for those that were unsuccessful.

Examples

Unmark a single file so that it is no longer eligible for deletion from the archive.

```
ID = gd_archive('C:\file.dat');
gd_markfordeletion(ID);
unmarktotal = gd_unmarkfordeletion(ID)
```

```
unmarktotal =
     1
```

Query variable metadata that has been marked for deletion, and then unmark the corresponding variables so that they are no longer eligible for deletion from the archive.

```
q = 'standard.archiveDate > 2004-12-01 & a.b < -500';
qresults = gd_querydeleted(q, 'varmeta');
for i=1:size(qresults,2)
    IDs{i} = qresults{i}.standard.ID;
end
unmarktotal = gd_unmarkfordeletion(IDs)
```

```
unmarktotal =
     5
```

Notes

Only the owner of the data (the person who archived it) can unmark it for deletion.

If the marked data has been permanently deleted from the archive by an administrator it cannot be recovered.

A valid proxy certificate is required (see `gd_createproxy` from the Geodise Compute Toolbox).

See also

`gd_markfordeletion`, `gd_querydeleted`, `gd_createproxy`

XML Toolbox

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). 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 6 XML Toolbox functions

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="3.1" 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 (i.e. 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="3.1" 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. 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

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) 2002-2004
  Author: Marc Molinari <m.molinari@soton.ac.uk>
  $Revision: 1.5 $ $Date: 2004/03/31 15:51:04 $
```

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 state.

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 `idx`, `size`, `type` (default), 'off' doesn't store XML type attributes.

Examples

This example saves a Matlab structure as XML in a file at a given location.

```
v.name = 'Google'
v.url = 'http://www.google.com'
v.rating = 5
v.description = 'Great search functionality for the web'
xml_save('c:/data/myfavourite.xml', v)
```

See also

xml_formatany, xml_format, xml_parse, xml_load, xml_help