The Argus Database

John Stanley

May 7, 2018

Contents

1 Preface

The collection, archiving and analysis methods of Argus or any other data-intensive program require and rely on a great of metadata and standardization to make the tools work. While easy to handle in ad-hoc ways when the data are limited to a single sensor and experiment, the need for structure and conventions increases exponentially as the number of stations or collections expand or if collaborators wish to share tools. The Argus methods described in this document represent almost 30 years of experience including evolving technologies and increasingly extensive global partnerships. Success is only possible with the discipline of the conventions and structures agreed to by the community and described herein.

This document has two major components and two complementary roles. The Argus database is our way to interact with all of the metadata needed to make Argus work. Our implementation is highly successful, but others may wish to use other specific database versions as long as they are consistent with the Argus protocols. More critically, the data structures that compose the database (table definitions in database-speak), must have standard structures if Argus analysis programs are to be compatible. These structures are the building blocks of Argus analysis program so must be adhered to. Tools to ensure proper structures are described herein to simplify this process.

This document serves the dual purpose of describing the workings of the database as well as defining and describing the various tables (data structures) that are needed by Argus. Even if you do not want to use the database, you need to understand and adhere to the data structures.

Section 2 includes introductory material, while section 3 discusses some principles and broad components of the database. Section 4 quickly covers installation and testing of the database. The most important section and the part that should be familiar to all is section 5 describing the various table/structures and the expected (required) fields in each table. This is the language of Argus needed for compatible programming. Finally, section 6 describe database access tools and calls.

2 Introduction

This document describes the CIL implementation of the Argus database and the philosophy behind its implementation. It is specific for the MySQL version of this database and code in the CIL argusDB2 and argusDB.mysql3 paths, with code updates through December of 2017. It is anticipated that as the legacy CIL code is updated to the structures descibed in this manual, the "argusDB2" path will be changed to "argusDB", which implements the previous version.

The table definitions in Section 5were established for database purposes, but are the accepted way of defining Argus metadata for any Argus programming.

2.1 Goals

The primary goals for using a database are:

1. Uniformity and regularity of data types. By using a database which will enforce typing and formatting on the data entries, the software will not be required to deal with arbitrary data.

2. Centralized datakeeping. A central database server can allow many people all working at the same time access to the same data, and to changes made by others when they are made.

3. Centralized maintenance. Bad data can be removed with one operation, or fixes made the same way.

There are certain principles that should be followed when designing a good database. One of those is that if there is an indeterminate amount of data that is available related to one entry in a table, then there are multiple entries in a table designed to hold that data and they point TO the related table. In other words, if you calculate a geometry for an image using an arbitrary number of ground control points, then there is a table containing one entry for each of the ground control points you used ('usedGCP' for Argus) and each of them points to the geometry that they were used with. It is ungood design to have the geometry point to each of the many used GCPs.

A second design goal is to reduce the amount of redundant information. This usually manifests itself in the creation of a table containing entries that are pointed to by multiple other entries in other tables. Instead of recording all of the information about the specific camera being used, a 'cameraModel' table is created with information common to all instances of that camera model, and the 'camera' table has a pointer to the entry in the 'cameraModel' table.

2.2 History

The beginnings of Argus predate usuable versions of Matlab, but at that time there wasn't much station information, and the first automated station had not yet been activated. Camera and geometry data were stored in simple text files. As the CIL moved into the use of Matlab and began creating more meta-information about Argus systems, it became clear that a relational database would be the appropriate solution. Unfortunately, Matlab was slow to develop a database toolbox, and for a long time the toolbox they did have worked only on Windows systems with an expensive commercial database system called "Oracle". There was one main free and open source database system called 'mSQL', and the first Argus database system was written using that, in the mid 1990's. This interface required extensive programming in C and connection to Matlab via the "mex" libraries. The mSQL database system was relatively simple but lacked significant functionality.

In the early 2000's there were two non-CIL versions of the database. In an attempt to bypass the need for a database server, a version written entirely in Matlab using '.mat' data files was written and used at NRL. The interface was significantly different, which created issues with code sharing. Also at NRL, a version based on a Java database called "McKoi" was written. Neither are currently active.

The current database system is based on 'MySQL', which is currently managed and maintained by ... Oracle. This change to MySQL took place around 2006. There is a "community" version of the commercial product, and that is what we are using today. The first versions written for MySQL also used the C/mex interface, but has now been migrated to Java. There are two remaining C/mex functions necessary. One converts a Matlab array into a "binary large object" (blob) for storage into the database (DBToBlob), and the other returns the array from the database (DBFromBlob). Compiling this remnant of code is significantly easier than managing code with a full MySQL library.

To determine which version of the Argus database code you are running, look for the directory "argusDB.xxx" on your path. The 'xxx' will be one of the following:

Name	Version
argusDB.msql	mSQL, very obsolete
argusDB.mysql	first MySQL version, now obsolete
argusDB.mysql2	second MySQL version, still in use
argusDB.mysql3	latest version, with changes for the 2017 database
argusDB.matlab	Matlab version, obsolete
argusDB.mckoi	McKoi database, obsolete

Table 1: Directories for database versions.

2.3 Design

The Matlab application program interface (API) was designed to allow

The database is arranged as a set of tables (similar to arrays of structures in Matlab), with each table containing a certain kind of data. For example, one table contains information about the Argus site, while others contain information about the station(s) at a site or the cameras at a station. The primary key for each table is generally the "id" field, which allows interconnection between the tables. Some tables do not have an "id", but instead use a system-generated sequence number. The latter tables tend to be those tables where the data are generated by the computer and not the database administrator. Examples of such tables are "geometry" and "usedGCP".

The Matlab "toolbox" for the database consists of two directories. The main directory is "argusDB", and it contains the application program interface (API) user-level functions. These are the functions that the user calls when accessing the Argus database. A secondary directory contains the database-specific (e.g. MySQL vs. mSQL vs. postgres etc.) functions. The secondary functions almost all have the suffix 'Raw' to differentiate them from the user interface. E.g. "DBConnect" is the function to connect and select what database server and database is used, which calls DBConnectRaw to perform the specific actions necessary to accomplish the task. You should avoid calling the "Raw" functions directly since they may change at any time.

The current secondary directory is "argusDB.mysql3", for version 3 of the MySQL interface. Both "argusDB" and "argusDB.mysql3" must be on your Matlab path.

Users who wish to, for whatever reason, use a database system other than the already written MySQL/Java should do so by duplicating the function found in the argusDB.mysql3 as closely as possible, so that code written to the "argusDB" standard may be used with little or no changes. Please. While there are few, if any, uses of "views" or other relational database-specific features, it is unlikely that non-relational databases (such as NoSQL) will be usable.

3 The Argus Database

The core Argus database consists of several tables, defining and describing the Argus station and its environment, and the parts that make up those stations. Figure 1 shows the basic set of tables and their connections.

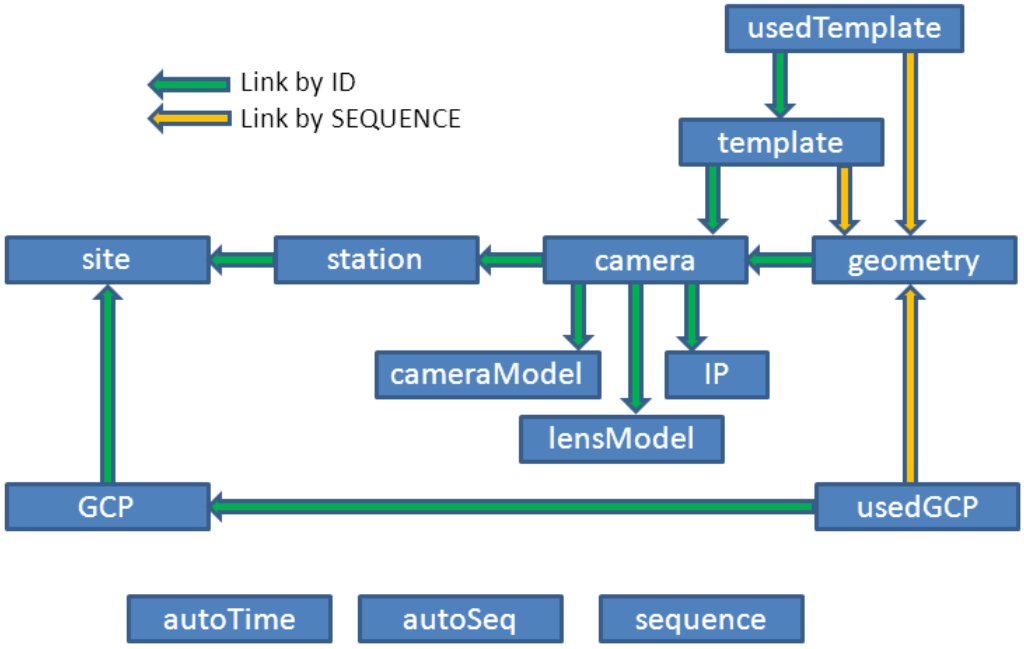

Figure 1: Core database tables and what they point to. The presence of an arrow indicates that one table contains a pointer to an entry in the other table. The bottom three tables are used by stored procedures in the database server to control automatic insertion of sequence numbers and/or update time into the other tables.

3.1 Database "pointers"

Pointers, in the CIL Argus database, are based on one of two table fields. The most common field used as a pointer is the "id" field of the destination table, with 'table-nameID' in the other table pointing to that destination. E.g. the "site" table contains an "id"; the "station" table contains a "siteID" field that will be set to the value of the "id" of the site to which the station belongs. Examples of this will be provided in sections describing the tables.

3.1.1 ID

By convention, "id" values are created by using the same first four characters for everything that belongs to a site, followed by three to five additional characters in a specified format. Early versions of the database used two characters unique to a site, but with the generation of many sites worldwide this was later extended to four. Legacy sites using two letters had 'XX' appended to the initial two characters.

For the Yaquina Head site, the original id was "YAXXX". When the id was extended to 7 characters, it was changed to "YAXXXXX". Any table entry describing data belonging to the Yaquina Head site, by convention, has an id that begins with "YAXX". The original North Head Lighthouse site id is 'NHXXXXX"; the new site is "NH2XXXX". It is not a requirement that sites end with "XXX".

Why is this convention used? While it is possible to write more complex database queries to gather information about a site and associated data by using the id information as a pointer, it is much more convenient to be able to use the first four characters of the id field by themselves. I.e., instead of a query that needs conditions like "where station.siteID=site.id ...", it is much faster to use a simple condition like "where id like 'YAXX%"' to retrieve Yaquina Head data from any table with an id field. (The '%' is a wild card in SQL.) As much as possible, the CIL argusDB.mysql3 and argusDB code does not make use of this shortcut, but users can take advantage of this convention to make simple queries to retrieve specific subsets of the database.

3.1.2 Seq

The "seq" field in a table is short for "sequence number". Most tables have this entry because it is used to help ensure that each entry in the database is unique. That is, it is part of the "unique key" defined for each table. This helps protect the user because an attempt at re-inserting the same data will fail with "duplicate key".

For tables where the generation of an "id" field would be difficult, the sole key is a computer-generated sequence number. It is this "seq" key that is used as the pointer value. For example, the "geometry" table is created automatically in most cases, so it is assigned a "seq" key when it is inserted into the database. The table of "usedGCP" values, which lists which ground control points (GCP) wer used in the geometry solution, contains a field "geometrySequence" that points to the geometry.

There are two methods of creating this sequence number in the CIL argusDB code. The first is through a special table in the database containing the "last used" sequence number (table "sequence"), and a special call that will return the next value while incrementing the remembered last value (see section 6.10). The use of this method is controlled by a flag in the table "autoSeq". The "autoSeq" table contains two columns. The first is the name of a table in the database, and the second is a "1" if automatic sequence numbers are to be used, "0" otherwise.

The second method is to use the Unix epochtime[1] value, assuming that one is not creating table entries faster than once per second.[2] Because if this issue, the Argus code is being converted to use the first version of sequential sequence numbers.

Some tables do not need a sequence for any other purpose than a unique key to identify the table entry, and thus the user really never needs to know this value. HOWEVER, the geometry table is different. The sequence number for the geometry is used to link to the "used" tables that identify GCPs or templates. Because of this requirement, the sequence numbers for a geometry must NOT be created using the trigger system discussed in section 6.7. The practical result of this requirement is that the "autoSeq" table must always have '0' for the "geometry" field value; this can be ensured by using DBDisableSeqTrigger('geometry').

3.1.3 Built-in Pointers

In some cases, the user will want to retrieve the linked data at the same time as the base table. The function "DBGetImageInfo", for example, takes an image name as input and returns all of the available information about it. This includes the site, station, camera, geometry, and other data. Because this extra information takes additional database calls to retrieve, and it is not necessary for all retrievals of data about the image, this multi-step retrieval has been written into the DBGetImageData function.

However, there are some cases where one table is so intimately linked to another that the retrieval should be automatic. The image sensor sizes for a camera are one such example.

To account for such use, the basic "DBGetTableEntryRaw" function has been written to look for two specially formatted field names in every table. Any field name that begins with "l?_" (for "link") will be treated as a link to another table.

The field name indicates what kind of link it is. If the "?" is "i" it will be a link to another table's "id" field. If "?" is "s", it is a link to a sequence in another table. The name of the other table follows the underscore.

Currently, the only such link field in the base ArgusDB is the "li_IP" field in the camera table. This field contains the "id" of the associated entry in the IP table. The IP table contains basic information about the "image processor" or image sensor used in the camera (or in the station for older systems). When "DBGetTableEntry" sees this field name, is automatically peforms a second database retrieval from the "IP" table for the entry with the "id" contained in the camera record. That data is inserted into the returned "camera" struct as a new field named "IP".

You might think that this should also be applied to the cameraModel and lensModel data, but those tables are not as critical to the CIL/CIRN Argus code so the overhead of retrieving data from these tables is avoided by using a non-linked field name.

[1] The Unix 'epoch', or epoch time, is the number of seconds since Jan. 1, 1970, 0000 GMT. In comparison, a Matlab "datenum" is a floating point number of days, with datenum 1.0 being Jan. 1, 0000. Datenums are not tied to a time zone; Unix epoch time is always referenced to GMT.

[2] This assumption is most often broken when inserting multiple geometry solutions into the database after generating them all. A work-around is to delay each insert command by one or two seconds so the Unix epochtime will have rolled over to the next second.

3.2 Database "time"

Many database tables contains the fields "timeIN" and "timeOUT". These are Unix epochtime values that identify a time range for which the table entry is valid. For example, a specific station may have been installed on Mon Feb 9 18:50:34 2009 (epoch time 1234234234) and removed on Sat Aug 18 20:02:25 2012 (time 1345345345). To make sure that all data requests for this station would locate this specific station, is it normal to create the timeIN value as slightly less than the exact station installation time, and the timeOUT value as either '0' (for a station that is still operational), or a few minutes after it is actually decomissioned. For example, this station could have a timeIN value of 1234234000 (a convenient round number, 234 seconds before it actually went in) and a timeOUT of 1345346000 (another convenient round number, about 10 minutes after it was actually removed.) These time values are not critical, but must properly define which station at a site (if there has been more than one) was in existence at any specific time.[3]

Much of the Argus code has been written assuming that the database is consistent. That is, only one station with a certain name exists at any one time, or one camera with a specific camera number at a station at one time. The symptom of incorrect time control will be a fatal error in Matlab where it expects one item returned from the database query and there are two.

For convenience, a value of '0' for "timeIN" means that the entry is valid from "the beginning of time"[4]**. More convenient is that a 'timeOUT' value of '0' means "still there". This avoids all overflow types of problems where a default value of, say, 999999999, was used to mean "still there", which would become obsolete on Sat Sep 8 18:46:39 2001 when that was the actual epoch time.**

When replacing a camera, for example, it is critical to update the database entry for the camera you have just removed so that it has a true "timeOUT" value, and the new camera must have a "timeIN" that is later than the "timeOUT" of the previous camera.

Immediately upon writing Argus database access functions, it was noted that a common request was determining what entry was valid at a specific time. With a "timeIN" and "timeOUT" entry, it is easy to write an SQL conditional statement to accomplish this, but it would add complexity. Because it was required for so many functions (current camera, geometry, etc), and because the normal value comparision is "equal to", the special field name 'time' was created. See Section 6.2.1for more explanation and an example. The important point to note here is that "time" is a reserved field name for Argus database tables, and will have special processing applied when used in a query. A recommended alternative for "time" as a database field name is "epoch".

3.3 Timestamp

Each database table contains a field called "timestamp". This field is intended to record the last update or creation time for that entry. In the mSQL version of the database, this was managed by the database itself internally. Under MySQL, this field is set by something called a "trigger", or "stored procedure" in general database terms. The act of inserting or updating an entry

[3] This requirement is why two stations at one site at the same time MUST have different shortName and id fields. For example, argus02 and argus02a both existed during the SuperDuck field experiment and thus the timeIN/timeOUT values overlapped. Two station names were necessary.

[4] Technically, epoch time 0 is January 1, 1970 00:00 GMT. Since there were no Argus stations at that time, it is effectively "the beginning of time".

in the table calls a "stored function" which assigns the current time to the "timestamp" field, which is then stored with your data. The intended use of this field is to allow mult-database updates, selecting the more recent of two differing entries when they exist.

If you are creating your own table with a 'timestamp' field and want it set when an entry is made to that table, then you must add the appropriate "triggers". The function DBCreateTriggers will create the appropriate triggers for any table that needs them. The trigger code also uses an entry in the table "autoTime" to enable and disable the automated setting of the timestamp. You will need to alter the table to add a field with the same name as your new table, and a value of 1 to enable automatic timestamp updates, or 0 to disable them. The SQL commands:

```
alter table autoTime add column myNewTableName int(1);
update autoTime set myNewTableName=1;
```

will update the autoTime table to include your new table, and set the flag so that the update trigger will insert the current epoch time every time you update a row in that table.

4 Installation

Installation of the Argus database system requires four steps. First, obtain the argusDB and argusDB.mysql3 directories. Second, compile the remaining two MEX functions. Next, adjust the startup pathing for Matlab to include both argusDB source directores. And finally, test it.

A fifth step is to actually set up the MySQL database, which will be discussed later.

4.1 Matlab Code

The Argus database toolbox consists of the directories argusDB and argusDB.mysql3. You can obtain these from the CIL using subversion with the following two commands:

```
svn co svn://argus-home.coas.oregonstate.edu/CIL/argusDB2
svn co svn://argus-home.coas.oregonstate.edu/CIL/argusDB.mysql3
```

or by using the "extract" option of Windows-based TortoiseSVN or similar program. It is expected that there will be a Github version of the code in the near future.

4.2 Compiling MEX code

Run Matlab, and cd into the directory argusDB.mysql3. The subversion download contains pre-compiled versions of the two MEX programs, so you may not need to compile anything. To test this, try executing the command:

```
DBFromBlob(DBToBlob(12))
```

from within Matlab. If you receive an "ans" that looks like:

```
ans =
    12
```

then you need to do nothing more. This has been tested using Matlab 2017a and the subversion functions on a CentOS 7 system. If you see anything else, you will need to compile both functions.

Compiling the two necessary MEX functions may be as simple as entering the following commands:

```
mex DBToBlob.c
mex DBFromBlob.c
```

Depending on your current version of Matlab, you may see some warnings about "incompatible type conversion", but if the program compiles it should be fine.

You may also need to configure MEX, and worst case, install a compiler. Unix systems should already have a C compiler, but on Windows you may need to locate a Microsoft compatible compiler. The web page at:

```
https://www.mathworks.com/support/compilers.html
```

contains information about compilers for Matlab. After you have compiled both functions, test them as above.

4.2.1 Important note concerning Matlab versions and MEX functions. "Vany"

It is possible that changes in Matlab versions can introduce changes in MEX function calls that "break the code". This was last observed in the Argus database code when Matlab changed from version 6.x to version 7.x (which happened around 2003). Versions since that time have been compatible. When this happened, the Argus database code began selecting Matlab-specific versions of the MEX functions. These are located under the "argusDB.mysql?" directory, and are of the form "Vxx", where "xx" is based on the Matlab version number. For example, Matlab 2017a is version 9.2, and the associated directory would be V92.

HOWEVER, since the MEX code is currently compatible for any recent Matlab, "argusDB.mysql3" has only a "Vany" directory, which is empty. All MEX functions are located in the argusDB.mysql3 directory. At some future time, if a MEX issue is discovered, then the specific version directories will be once again necessary. The correct version directory is identified in the DBConnectRaw function.

4.3 Matlab Setup for argusDB

You must add the argusDB commands to your "matlabpath" before you can run the Argus database toolbox. These commands are normally added in your startup.m, such as:

```
addpath /home/ruby/matlab/CIL/argusDB
addpath /home/ruby/matlab/CIL/argusDB.mysql3
```

4.4 Testing

If you are going to use the Argus database, you should connect to it as early in your Matlab session as possible. Doing this is also how you can test the installation. The command to connect to an Argus MySQL database is:

```
DBConnect server user password database
```

The "server" is the hostname of an Argus database server. The CIL server is located at blossom.coas.oregonstate.edu. If you have not yet installed your own database server, please contact the CIL and you can get access to ours.

The "user" is the assigned username[5]. By convention, if you need only to read from the database (e.g. you are not entering geometries or other data) the user is "default". The "password" is the password assigned to the user, which is '' (two single quotes for an empty Matlab string) for the user "default.

The "database" is the database on the server. Each server can have many different databases depending on what other data is served from it. The Argus database is called, by convention, 'argus'.

Therefore, if you are testing your installation by using the CIL server at Oregon State, asking only for read-access, the command to test would be:

```
DBConnect blossom.coas.oregonstate.edu default '' argus
```

If you are successful, you will get an "ans" that looks like this:

```
  server: 'blossom.coas.oregonstate.edu'
    user: 'default'
password: ''
database: 'argus'
    port: 3306
```

The DBConnect function is described in more detail in section 6.3, "DBConnect". If you get a "Communications Link Failure" error, see Section 7.2 for a possible solution.

4.5 Using the database

Once you have connected to a database using DBConnect, the system will remember that connection and the credentials until you clear global variables. A "clear all" will remove the status information about the connection but not the connection credentials and server information. If you "clear all", the next time you try to access the database for any reason, you will see the message:

```
DB connection went away, connecting again
Reinit JDBC connector
```

This is normal. If you leave your session inactive, or do not access the database for a long period of time, you may see a warning that your connection timed out, and the same "Reinit" notice. You may also see a lot of red error text. This is, unfortunately, a known issue that has not yet been resolved. You will find that even after the error has caused your database access to fail, you will now be connected to the database and the same command should now succeed.

If there is any doubt about your connection status, you may always use a bare "DBConnect" command and your connection will be re-established if it has been lost, using the same credentials you used the last time you connected. If you wish to change your user status (for example, from "default" read-only to a user with write permission) you may simply reissue the DBConnect command with the new parameters.

[5] User management is covered in section 6.6.

A fuller description of Argus database commands will be described following the description of the tables.

5 Core table descriptions

The following table descriptions show the "official" data declaration as entered in the MySQL database server, and information about the contents of each field with the given name. The "type" of a field can be either an integer ("int"), a floating point number ("double"), a character string ("char"), or a special purpose binary field ("blob", used to encode binary data). The length of the field is the number of characters or digits used for that field. For most Argus database integers, the length is 11 digits ('int(11)')[6]. Character strings are limited to the number of bytes defined in the table, which means that a field defined as 'char(7)' may be up to seven characters long.

The definition for each table is shown as the result of a MySQL "show create table xxx" command, where 'xxx' is the name of the table. The definition starts with the name of the table in the "CREATE TABLE" line. Following that are the fields, first with the name ('seq', for example), then the type (int(11)), then whether the value in the database can be "null" (empty). The "UNIQUE KEY" lines identify one or more keys for that table. The field name in the parentheses is the field that must be unique in the table data. For example, the "site" table has unique keys on both the 'id' and 'seq' field – you may not have duplicate entries with the same ID or sequence number[7]. The "ENGINE=" line defines the kind of storage used for the data and is included only for completeness. You can ignore it.

Following the table definition will be a description of the fields in the table. The string '(M)' indicates mandatory entries. These are entries that are used by the Argus code, while others are mainly included for bookkeeping (for example, "zDatumNote" in 'site').

And finally, an example row from each table will be shown. Please refer to Figure 1to visualize the tables and their relation to each other.

5.1 Site

The "site" table contains information specific to each site. In this usage, a site is a geographic location, such as "Yaquina Head", which may contain one or more stations. This table defines the mapping of the local Argus coordinate system to the real world, and all stations at that site will use the same conversion.[8]

```
| site  | CREATE TABLE `site` (
  `seq` int(11) NOT NULL,
  `id` char(7) NOT NULL,
  `siteID` char(8) NOT NULL,
  `name` char(64) NOT NULL,
```

[6] Why 11? The earliest versions of the database and Argus structures used "double" for almost all numbers. The primary use of integers was for the epoch time. When we started, the epoch time was 9 characters long, which quickly changed to 10 (at Sat Sep 8 18:46:40 2001). Anticipating Nov 20 9:46:40 2286 (when epoch time becomes 11 digits), most ints were created as int(11). As we began getting some large arrays of double values that stressed the RAM under Matlab, we began using smaller integers (uint8, uint16, etc) and the database was changed to reflect that.

[7] A requirement for good database programming is that there is always a unique identifier of some kind for each row in a table. If you want to remove two duplicate entries in a table and do not have a unique key, then you cannot select just one to remove, you must remove both and then put one back. "seq" serves this purpose for all Argus DB tables.

[8] CIL routines in the UTM toolbox include 'argus2ll' to convert from local Argus coordinates to lat/lon, and 'll2Argus' to go the other way.

```
  `lat` double NOT NULL,
  `lon` double NOT NULL,
  `elev` double NOT NULL,
  `zDatumNote` char(32) NOT NULL,
  `WGS84Height` double NOT NULL,
  `scaleCorrE` double NOT NULL,
  `scaleCorrN` double NOT NULL,
  `TZoffset` int(11) NOT NULL,
  `tideSource` char(64) NOT NULL,
  `waveSource` char(255) NOT NULL,
  `degFromN` double NOT NULL,
  `owner` char(20) NOT NULL,
  `TZName` char(8) NOT NULL,
  `useLocalNames` int(11) NOT NULL,
  `sortLocalTime` int(11) NOT NULL,
  `timestamp` int(11) default NULL,
  UNIQUE KEY `siteidx` (`id`),
  UNIQUE KEY `siteseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

The fields in the site table mean the following:

seq - (M) an up to 11 digit sequence number unique to this entry

id – (M) a 7 character identifier for the site. It consists of four letters followed by 'XXX'. For example, 'YAXXXXX' is the site id for Yaquina Head.

siteID – (M) an 8 character "short name" for the site. Deprecated, but still used in some functions. The siteID for Yaquina Head is YAQUINA.

name – a long name, up to 64 characters. This describes the site for human viewers. "Yaquina Head, OR"

lat – (M) the latitude of the reference point for the site, as a double precision number in decimal degrees. South is negative. This, and the following 'lon' entry, define the location of the 0,0 coordinate in the Argus local coordinate system. A convenient value would be a known benchmark, tying the Argus coordinate origin to that benchmark location[9].

lon – (M) the longitude of the site, as a double precision number in decimal degrees. West is negative.

elev – (M) the elevation of the site in meters. Or metres, for non-US locations. This is an approximate value; entries in other tables that require elevation information, such as camera or GCP, will be accurate relative to the z datum being used.

zDatumNote - any comment regarding the elevation datum used in 'elev' or other elevation fields in other tables.

WGS84Height - ellipsoidal height of the elevation. Currently unused. If unknown, use 99999.

scaleCorrE - (M) the scale correction for conversion from lat/lon to and from UTM and then into Argus local. This is usually a number very close to 1. If you do not know the true value, use 1.

scaleCorrN - (M) same kind of data as scaleCorrE except applied to northing. Same restrictions as scaleCorrE

[9]For both "lat" and "lon", make sure to use enough significant digits to achieve the level of precision you want for your 0,0 reference point. You may use "45.0" for site latitude, if you wish, but you probably want to tie 0,0 to something a bit more specific, such as a benchmark or significant fixed feature.

TZoffset – the difference from GMT in minutes for the site's standard time. E.g., Oregon (Pacific Standard Time) is -480 minutes.

tideSource – a space delimited list of the ids of tide sources for this site. It is intended that these be the ids of various tide gauges for which data is available, and that these ids can be used to select that gauge in calls to retrieve tide data.

waveSource – a space delimited list of the ids of wave data sources, similar to the tideSource field.

degFromN – (M) the direction of the positive Y axis, clockwise from true north, in degrees. This supplies the rotation parameter to convert from lat/lon through UTM to Argus local; offset is provided by the lat/lon values.

owner – an 8 character string identifying the owner of the site and its data. This field is used when distributing site data to restrict access to authorized users. When empty, the data are not restricted.

TZName – a string of up to 8 characters containing the timezone name associated with the TZoffset indicated above. This string is used when creating Argus filenames in site-local time.

useLocalNames – a flag (0/1) that indicates whether archived images will use GMT-based times or local times based on TZoffset and TZname.

sortLocalTime – a flag (0/1) that indicates whether the archived image files are stored in days based on GMT or site local time.

An example, in Matlab "struct" format, of a site table and its contents is:

```
              seq: 55
               id: 'YAXXXXX'
           siteID: 'YAQUINA'
             name: 'Yaquina Head, OR'
              lat: 44.658458283120801
              lon: -1.240563776909660e+02
             elev: 400
       zDatumNote: ''
      WGS84Height: 99999
       scaleCorrE: 1
       scaleCorrN: 1
         TZoffset: -480
       tideSource: '9435380'
       waveSource: ''
         degFromN: 180
            owner: ''
           TZName: 'PST'
    useLocalNames: 0
    sortLocalTime: 0
        timestamp: 1.318375115000000e+09
```

5.2 Station

The station table describes a station. Each site (geographical location) may have one or more stations associated with it. In addition, as stations are updated (from PC to SGI and then back to PC, for example) each version is recorded as a seperate station. The definition for the station table is:

```
station | CREATE TABLE ‘station‘ (
  ‘seq‘ int(11) NOT NULL,
  ‘id‘ char(7) NOT NULL,
  ‘shortName‘ char(9) default NULL,
  ‘name‘ char(64) NOT NULL,
  ‘siteID‘ char(7) NOT NULL,
  ‘timeIN‘ int(11) NOT NULL,
  ‘timeOUT‘ int(11) NOT NULL,
  ‘timestamp‘ int(11) default NULL,
  UNIQUE KEY ‘stationidx‘ (‘id‘),
  UNIQUE KEY ‘stationseq‘ (‘seq‘)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

The fields in the station table have the following definitions:

id – (M) an up-to 7 character id for the station. The id consists of four letters (matching the first four letters of the site, for convenience), two digits, and the letter "S" to identify a station id. E.g., "YAXX01S".

shortName – (M) an up-to 9 character UUCP name[10] for the station. This name is also the one that appears in the filenames in the data archive. 'argus00', e.g.

name – a string of up to 64 characters that contains a long, human meaningful name for the station.

siteID – (M) the id of the site where this station exists. This must be the id from an existing site.

timeIN – (M) the Unix epoch time that the station came into existance. If this is the first station at a site, it may have "0" here.

timeOUT – (M) the Unix epoch time when the station went out of service. A special case is a timeOUT of "0" which means that the station is still operating. The timeIN and timeOUT values allow the system to pick which "argus00" (shortName) was in operation when a specific image was collected, for example.

A sample station table entry is:

```
       seq: 1
        id: 'YAXX00S'
 shortName: 'argus00'
      name: 'Yaquina Head, OR first ever'
    siteID: 'YAXXXXX'
    timeIN: 7.0764e+08
   timeOUT: 8.69178e+08
 timestamp: 1329052020
```

5.3 cameraModel

Each Argus station consists of one or more cameras. A station may have more than one model of camera, and several stations may all have the same model. To prevent duplication of the camera model specific data for each instance of an Argus camera, that data is kept in the "cameraModel" table (with a similar "lensModel" for the lens, and "image processor" (sensor

[10]The standard communication protocol for returning data from an Argus station is the Unix-Unix Copy Program, or UUCP for short. This is an up-to 8 character name that is used for authentication and identification of the UUCP station. One existing station that did not use UUCP was given a 9 character name, so this field was extended by one to accomodate that.

in digital cameras, or the "chip")), and the "camera" table will contain the IDs that correspond to the model of each.

```
| cameraModel | CREATE TABLE `cameraModel` (
  `seq` int(11) NOT NULL,
  `id` char(5) NOT NULL,
  `make` char(32) NOT NULL,
  `model` char(32) NOT NULL,
  `size` double NOT NULL,
  `color` int(11) NOT NULL,
  `timestamp` int(11) default NULL,
  UNIQUE KEY `cameraModelidx` (`id`),
  UNIQUE KEY `cameraModelseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

And has fields with the following definitions:

id – (M) an up to 5 character string identifying this entry in the database. It must be unique, but otherwise can be anything. This id is how the camera table refers to the specific camera model, and will be entered in the camera "modelID" field.

make – an up to 32 character string identifying the maker of the camera.

model – an up to 32 character string identifying the model of the camera.

size – a floating point number containing the nominal image sensor size. This is the 1/2, 1/3, 1/4 inch value reported in the camera manual. The specific sensor can be recorded in the "IP" table.

color – a flag (0/1) indicating whether this is a non-color or color camera.

Most of this table is for documenting the station equipement; however the image sensor size can be used by tools that calculate real fields of view based on lens focal length and camera details. A sample cameraModel entry is:

```
   seq: 4
    id: 'DC50'
  make: 'Sony'
 model: 'SSC-DC50'
  size: 0.5
 color: 1
```

5.4 lensModel

The "lensModel" table contains information about the various models of lenses used at Argus sites. It is declared thusly:

```
| lensModel | CREATE TABLE `lensModel` (
  `seq` int(11) NOT NULL,
  `id` char(5) NOT NULL,
  `make` char(32) NOT NULL,
  `model` char(32) NOT NULL,
  `f` double NOT NULL,
  `aperture` double NOT NULL,
  `autoIris` int(11) NOT NULL,
  `timestamp` int(11) default NULL,
```

```
  UNIQUE KEY `lensModelidx` (`id`),
  UNIQUE KEY `lensModelseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

with fields that have the following definitions:

id – (M) an up to 5 character id uniquely identifying this lens model. This id is how other tables (such as camera) refer to this entry. (It is found in the "lensModelID" field.) It must be unique. It has typically been constructed by using the first letter of the make, two numbers for the focal length, and then MI for manual and AI for auto iris. E.g., a Fujinon 9 mm manual iris lens would have an id of "F09MI".

make – an up to 32 character string indentifying the lens maker.

model – an up to 32 character string identifying the specific lens model.

f – (M) a floating point number representing the lens' focal length.

aperture – a floating point number representing the lens' maximum aperture.

autoIris – a flag (0/1) set to 1 if this lens is an auto-iris lens and 0 otherwise.

A typical lensModel entry looks like this:

```
           seq: 3
            id: 'F16AI'
          make: 'Fujinon'
         model: 'HF16B-SND4-1'
             f: 16
      aperture: 1.8
      autoIris: 1
```

5.5 IP

The "IP" table records important data about the image processors used in Argus stations. It may not seem obvious why this table exists. Modern cameras and Argus stations do not have an "image processor" as they used to. The camera itself produces the digital image. However, the sensor ("chip") in the camera performs the same function as the old image processors – converting the analog image to a digital format. If you think about the sensor as the basic "image processor", then this table may make more sense, and, in fact, the "chip" has many of the same parameters as the old image processors. Some of this information has moved to the "camera" table, but the basic info in this table has not. It would be a duplication of this data to include it in every camera entry, and make it more complicated to correct any errors or updates as they occur.

The table is declared as:

```
| IP    | CREATE TABLE `IP` (
  `seq` int(11) NOT NULL,
  `id` char(12) NOT NULL,
  `make` char(32) NOT NULL,
  `model` char(32) NOT NULL,
  `name` char(128) NOT NULL,
```

```
  `width` int(11) NOT NULL,
  `height` int(11) NOT NULL,
  `pixelWidth` double default NULL,
  `pixelHeight` double default NULL,
  `timestamp` int(11) default NULL,
  UNIQUE KEY `IPidx` (`id`),
  UNIQUE KEY `IPseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

and fields with the following meanings:

id – (M) an up to 8 character string identifying this image processor entry. This id is used by the camera table to record which IP is digitizing the image in each camera. It is in the "IPID" and li_IP fields (currently redundant) in the camera table.

make – an up to 32 character string identifying the maker of this IP.

model – an up to 32 character string identifying the specific model of this IP.

name – an up to 64 character "name" for the IP, which is intended for human readers.

width – (M) the width of the image (in pixels) produced by this IP.

height – (M) the height of the image (in pixels) produced by this IP.

pixelWidth - the physical width of one pixel on the sensor in micrometers. From this and the width value it should be possible to calculate the physical width of the imagine area of the sensor, and thus the field of view given a valid focal length.

pixelHeight - the physical height of one pixel on the sensor. See pixelWidh.

An example IP table entry looks like this:

```
               seq: 27
                id: 'BFLY50H'
              make: 'Sharp'
             model: 'RJ32S3AA0DT'
              name: 'CCD, 2/3 color'
             width: 2448
            height: 2048
        pixelWidth: 3.4500
       pixelHeight: 3.4500
```

This is the IP entry for the Point Grey Blackfly BFLY-PGE-50H5C that has a Sharp 5MP 2/3" image sensor.

5.5.1 The INDEF IP

As you use the Argus database, you may come across the image processor with the ID "INDEF". This IP entry is a special case, based on an attempt at using historical photographs taken at a site as Argus images. This ID triggers special code that calculates the theoretical parameter values based on the actual image size. The last word on this was that it isn't working too well. There is nothing to see here. Move along.

5.6 camera

The "camera" table contains information about the cameras in use at a particular station. It is perhaps one of the most complicated tables, simply because it must serve data for cameras that range from "photographs" (IPID "INDEF") through the entire series of analog, and now into modern digital GigE. It also contains data using four or five[11] different systems of distortion calibrations. One magnificent example of legacy information this table contains is the "kramerButton" used with the SGI Argus station to control the external video switcher. We have tried to place these legacy entries later in the table.

The 'camera' table is declared as follows:

```
| camera | CREATE TABLE `camera` (
  `seq` int(11) NOT NULL,
  `id` char(7) NOT NULL,
  `stationID` char(7) NOT NULL,
  `modelID` char(5) NOT NULL,
  `syncsToID` char(7) NOT NULL,
  `lensModelID` char(5) NOT NULL,
  `lensSN` char(16) NOT NULL,
  `cameraSN` char(16) NOT NULL,
  `polarizerFlag` int(1) default NULL,
  `polAngle` double default NULL,
  `filters` char(32) NOT NULL,
  `orientation` char(6) NOT NULL,
  `cameraNumber` int(11) NOT NULL,
  `timeIN` int(11) NOT NULL,
  `timeOUT` int(11) NOT NULL,
  `x` double NOT NULL,
  `y` double NOT NULL,
  `z` double NOT NULL,
  `K` tinyblob NOT NULL,
  `kc` tinyblob,
  `li_IP` char(11) NOT NULL,
  `IPID` char(12) NOT NULL,
  `D1` double default NULL,
  `D2` double default NULL,
  `D_U0` double default NULL,
  `D_V0` double default NULL,
  `Destimated` int(1) default NULL,
  `Drad` double default NULL,
  `Dtan` double default NULL,
  `kramerButton` int(11) default NULL,
 `timestamp` int(11) default NULL,
  UNIQUE KEY `cameraidx` (`id`),
  UNIQUE KEY `cameraseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

with the following field definitions:

[11] Ok, maybe only three. But "several".

id – (M) an up to 7 character string uniquely identifying this camera. It is created from the first four letters of the station id, two digits, and the letter 'C' for camera. Any change to the physical camera requires a new camera entry in this table, with a new camera ID.

stationID – (M) the ID of the station this camera is being used at. If one camera is being used at more than one station, there must be separate camera table entries for each one.

modelID – (M) the ID from the cameraModel table entry that describes the camera model.

syncsToID – the ID of the camera which provides the synchronization for the video from this camera. This was especially significant in the analog video systems since that determined which camera was the sync source. In Firewire systems, all cameras typically synced to the bus clock. Under GigE, an external trigger source is commonly used. This field can be any arbitrary string that indicates the correct situation; for analog systems it would be the camera ID.

lensModelID – (M) the ID of the corresponding lensModel table entry for the lens on this camera.

lensSN – an up-to 16 character representation of the serial number for the lens on this camera.

cameraSN – an up-to-16 character representation of the serial number for this camera.

polarizerFlag – a flag (0/1) indicating the presence of a polarizing filter on the lens of this camera.

polAngle – a double containing the rotation angle of the polarizer in degrees, counterclockwise from horizontal, in the look direction. If this value is unknown, or unused, set it to 999.

filters – up to 32 characters describing any filters on the lens of this camera.

orientation – up to 6 characters describing the orientation of this camera relative to true north. I.e., "sse", "nw".

cameraNumber – (M) the integer "number" for this camera. At most sites, cameras range from 1 up; at old sites, for historical reasons, camera numbers sometimes start at 0.

timeIN – (M) the Unix epoch time when this camera was installed or images were first collected.

timeOUT – (M) the Unix epoch time when this camera was removed, or '0' to indicate a camera that is still being used.

x – (M) the floating point value representing the X location of this camera, in the local Argus station coordinate system, in meters.

y – (M) the floating point value representing the Y location of the camera, ditto.

z – (M) the floating point value representing the camera elevation, ditto again.

K - the 3x3 K matrix for this camera used when computing the photogrammetric projection. Note that for historical reasons, the vertical field of view (K(2,2)) is negative. (We count V from the top down.)

kc - the kc matrix (5x1) from the Caltech camera calibration toolbox, which includes radial and tangential parameters.

li_IP – a link to the IP table through the IP 'id' field.

IPID - the IP id value for the image sensor or image processor

The following are legacy entries not normally used for current stations.

D1 – the third-order distortion coefficient for this camera/lens combination. A 0 here is a valid entry, but it means there will be effectively no distortion corrections for this camera.

D2 – the first-order distortion coefficient. 0 has the same meaning as for D1.

D_U0 - (M) the calibrated image center in U.

D_V0 - (M) the calibrated image center in V.

Destimated – a flag (0/1) indicating whether the D1/D2 parameters are estimates (from other similar camera/lens combinations, e.g.) or measured for this particular camera and lens.

Drad - a typically 1x4 array of data from an intermediate system of camera calibrations.

Dtan - unused but created to hold the tangential coefficients from the Drad system.

kramerButton – (M) the integer "number" of the button on the Kramer video switch that this camera's video signal appears on. Buttons are numbered from 1 through 6 for the Kramer 601 and from 1 through 8 for the 801.

An example camera table entry is:

```
              seq: 410
               id: 'NH2X01C'
        stationID: 'NH2X00S'
          modelID: 'PGE50'
        syncsToID: ''
      lensModelID: 'F50SA'
           lensSN: ''
         cameraSN: '15103318'
    polarizerFlag: 0
         polAngle: 999
          filters: ''
      orientation: 'S'
     cameraNumber: 1
           timeIN: 1.4900e+09
          timeOUT: 0
                x: 43.2834
                y: 5.0898e+03
                z: 64.1721
                K: [3×3 double]
               kc: [5×1 double]
            li_IP: 'BFLY50H'
             IPID: 'BFLY50H'
               D1: 0
               D2: 0
             D_U0: 1.2245e+03
             D_V0: 1.0245e+03
      Destimated: 0
            Drad: 0
            Dtan: 0
     kramerButton: 0
        timestamp: 1.5110e+09
               IP: [1×1 struct]
```

You may notice that there is a field in this structure called "IP". This is the information from the IP table that the field "li_IP" earlier in the camera table referred to. Most notably, this is where you can find the IP or sensor chip image width and height (camera.IP.width/camera.IP.height). See section 2.1.3 for a fuller description.

5.6.1 Notes on distortion in the 'camera' table

Over the two-plus decades we have been doing this, we have gone through several systems of calibrating lenses. This has resulted in a table with multiple entries to record what should be the same thing. Unfortunately, the values depend on the system being used, and conversion is not always trivial.

The CIL 'distort' and 'undistort' functions which take a 'camera' struct as input have been designed to based their calculations on what fields are present and non-empty in that camera struct. For the example camera above, because there is a non-empty 5x1 array of calibration coefficients in 'kc', distort and undistort will use the Caltech distort and undistort routines ('distortCT' and 'undistortCT'). The USER does not need to worry about which will be used, since the functions 'distort' and 'undistort' will select the proper one given the information at hand.

Therefore, there is NO mandatory distortion coefficient field. If all fields related to distortion are empty, D1 and D2 will be '0' and no distortion corrections will be applied (actually a zero coefficient radial distortion is used.) If K and Drad are present, an old version that is similar to the ancient D1/D2 system will be applied. The presence of kc uses the Caltech system.

It would be simply marvelous if we could collapse the table and lose legacy distortion values, relying only in kc, but unfortunately, there is no simple way to convert old coefficients into the kc format, and we must still be able to process images from systems that don't have known kc values.

5.7 gcp

The 'gcp' table contains data about the ground control points existing at a site. Notice that GCPs are considered to exist at the geographical location (site) and not the logical (station) location. This allows GCPs to be shared between stations at the same site. The declaration of the gcp table is as follows:

```
| gcp   | CREATE TABLE `gcp` (
  `seq` int(11) NOT NULL,
  `id` char(8) NOT NULL,
  `name` char(64) NOT NULL,
  `siteID` char(7) NOT NULL,
  `timeIN` int(11) NOT NULL,
  `timeOUT` int(11) NOT NULL,
  `x` double NOT NULL,
  `y` double NOT NULL,
  `z` double NOT NULL,
  `xDim` double NOT NULL,
  `yDim` double NOT NULL,
  `intensity` int(11) NOT NULL,
  `eccentricity` double NOT NULL,
  `PPPable` int(11) NOT NULL,
```

```
  `timestamp` int(11) default NULL,
  UNIQUE KEY `gcpidx` (`id`),
  UNIQUE KEY `gcpseq` (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

and has the following field definitions:

id – (M) an up to 8 character id for this GCP. It is created from the first four characters of the site ID and four digits. This is how every piece of software that deals with GCPs will refer to them.

name – (M) an up to 32 character string giving a human-relevant identification of the GCP. E.g., "center disk". This is most likely how YOU will refer to GCPs.

siteID – (M) the ID of the site where the GCP exists.

timeIN – (M) the Unix epoch time that the GCP came into existance.

timeOUT – (M) the Unix epoch time that the GCP went away, or '0' if it hasn't yet.

x – (M) the floating point representation of the X coordinate of the GCP in the local Argus coordinate system. For non-point GCPs, this should be the location of the center of the target.

y – (M) ditto the Y coordinate.

z – (M) ditto the Z (elevation).

xDim – the dimension of the GCP in the x direction. This was intended for automated GCP detection, to assist in determining whether the automation did it correctly. Currently unused.

yDim – ditto the y dimension.

intensity – a flag (0/1) indicating whether this GCP is light colored (1) or dark (0). Also unused by anything currently.

eccentricity – the floating point eccentricty value for the non-point GCP. A perfect circle has an eccentricity of 2. Used the same way xDim, yDim and intensity are used.

PPPable – (M) a flag (0/1) which indicates whether this GCP can be identified using the "Pie Pan Picking" routine of geomtool6[12], or other similar routines in other geometry software. In other words, is this GCP visibly large enough to allow some sort of center of mass calculation for sub-pixel location. If you don't know the right answer for this flag, enter 1, since you don't HAVE to use PPP on it just because this flag is set.

An example gcp table entry looks like:

```
    seq: 215
     id: 'DRXX0002'
   name: 'Fence Corner'
 siteID: 'DRXXX'
 timeIN: 0
```

[12] "Pie Pan Pick" is an image processing function that was intended to identify the round, pizza pie pans that were used for some of the early GCPs. These were cheap, round targets available at the local grocery store. It started at the assumed center of the pan and extracted a rectangular region around that point. It then determined a contrast and threshold to "label" (bwlabel in Matlab) the pixels, and identify a connected region based on the initial center. The center of mass was then calculated, and that was the U/V for the GCP in the geometry solution. The "intensity" field was used to identify black circles on a light background, and the "eccentricity" was used to compare the identified "pie pan" to true round (a circle has an eccentricity of 2.0). This was the very first "autogeom" system.

```
            timeOUT: 0
                  x: 553.971
                  y: 99.667
                  z: 29.02
               xDim: 1
               yDim: 1
          intensity: 0
       eccentricity: 0
            PPPable: 0
```

5.8 geometry

The "geometry" table contains the results of geometry calculations, either from geomtool or some other geometry routine. This entry is declared as:

```
| geometry | CREATE TABLE `geometry` (
  `seq` int(11) NOT NULL,
  `cameraID` char(7) NOT NULL,
  `solvedVars` char(8) NOT NULL,
  `isAuto` int(1) NOT NULL,
  `autoMethod` char(10) default NULL,
  `similarityMethod` char(10) default NULL,
  `valid` int(11) NOT NULL,
  `m` tinyblob NOT NULL,
  `tilt` double NOT NULL,
  `azimuth` double NOT NULL,
  `roll` double default NULL,
  `fov` double NOT NULL,
  `imagePath` char(64) NOT NULL,
  `referenceImagePath` char(64) default NULL,
  `referenceGeomSeq` int(11) default NULL,
  `seedGeomSeq` int(11) default NULL,
  `meanSimilarity` double default NULL,
  `whenDone` int(11) NOT NULL,
  `whenValid` int(11) NOT NULL,
  `user` char(12) NOT NULL,
  `iterations` int(11) NOT NULL,
  `version` double NOT NULL,
  `tiltCI` double default NULL,
  `azimuthCI` double default NULL,
  `rollCI` double default NULL,
  `err` double default NULL,
  `timestamp` int(11) default NULL,
  KEY `geomseq` USING BTREE (`seq`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

and has field definitions as follows:

seq – (M) the sequence number for this geometry. Unlike all other tables with an 'seq' field, this table requires one. It is created by the geometry software to identify this geometry solution uniquely. It is, in short, the only way of automatically identifying and connecting other data to this solution.

cameraID – (M) the ID of the camera table entry for the camera involved in this solution.

solvedVars – (M) a list of the variables which were solved in the solution. See Section 5.8.1.

isAuto – (M) a flag indicating that this was an automatically (1) created geometry (as opposed to manual, 0).

autoMethod – a string reporting the automated method used, whether "auto" or "cross". Empty otherwise.

similarityMethod – a string identifying the kind of "cost function" used in the solution. (MI, cov, etc.) Empty for non-auto or other methods.[13]

valid – (M) a flag (0/1) indicating if this geometry solution is valid. Previously unused (always 1), but will be used to allow removal of bad solutions during external database merges. 0 means this solution is invalid, and it will eventually be removed. (This is John's Stupid Idea.)

m – (M) the "m" vector encoded as a "blob". Decoded automatically upon retrieval by DBFromBlob, encoded on insert by DBToBlob.

tilt – (M) the angular tilt of the camera, in radians.

azimuth – (M) the angular azimuth of the camera relative to the local coordinate system, in radians.

roll – (M) the angular roll of the camera, in radians.

fov – (M) the field of view of the lens and camera, in radians.

imagePath – an up-to-64 character string containing the file name of the main image used in this solution. This is the file name, excluding any path elements.

referenceImagePath – up to 64 character string with the file name of the reference image for crossgeom.

referenceGeomSeq – the sequence number of the geometry used with the reference image for crossgeom results.

seedGeomSeq – the sequence number of the geometry used as a starting point for the minimization for an autogeom or crossgeom.

meanSimilarity – a quality metric for automated geometries, the mean of the individual template similarity metrics.

whenDone – (M) the Unix epoch time recording the time the solution was calculated. This value is used in breaking "ties" when selecting the geometry solution for a particular image. If two or more solutions have the same whenValid entry, the one with the latest whenValid is chosen as the correct solution.

whenValid – (M) the Unix epoch time that this solution should start being considered as the current solution. It defaults to the epoch time in the imagePath name. The geometry solution with a whenValid most recently prior to the requested time will be returned as the "current" one, with ties being broken by whenDone, as mentioned above.[14]

user – an up to 12 character string recording the user who created this solution. Typically, the Unix username. It will be obtained from the "USER" system environment variable. It is not the user name that the user connected to the database as.

[13] Currently there are two valid entries. "MI" is "mutual information", "cov" is "covariance". If you create a new automated geometry solver, you can add to this.

[14] For the same reasons mentioned in Section 3.2 you may want to backdate this time. For example, if you perform a geometry on a time exposure image, you may want to enter a time a few seconds earlier so that the associated snap image is also covered by this geometry. If you are doing a geometry on a specific image because it wasn't foggy that day, you may want to backdate the "whenValid" time to cover previous days when the image is usable for other things and the geometry has not changed.

interations – an integer recording the number of iterations in the solution fit.

version – a floating point number indicating the version of the software used to calculate the solution. The specific version number will be defined in the code used to create the solution.

tiltCI – confidence interval on the tilt angle in radians.

azimuthCI – ditto azimuth.

rollCI – ditto roll.

err – the error of the fit

Notice that even though many of these fields are marked as mandatory, the user will almost never be responsible for entering anything into them. They will be created and managed by whatever geometry solution system is being used.

An example geometry table entry might look like this:

```
                  seq: 54
             cameraID: 'DRXX02C'
           solvedVars: ''
               isAuto: 0
           autoMethod: []
     similarityMethod: []
                valid: 1
                    m: [11×1 double]
                 tilt: 1.3375
              azimuth: 2.9758
                 roll: 0.0233
                  fov: 0.7503
            imagePath: '838984421.Fri.Aug.02_11:13:41.GMT.1996.droskyn.c2.snap.jpg'
    referenceImagePath: []
      referenceGeomSeq: []
           seedGeomSeq: -1
        meanSimilarity: -1
              whenDone: 900524657
             whenValid: 838984421
                  user: 'stanley'
            iterations: 4
               version: 1
                tiltCI: -1
             azimuthCI: -1
                rollCI: -1
                   err: 4.3642e-09
             timestamp: 1.5114e+09
```

Numerical entries that contain the value "-1" above indicate that this geometry was produced by a solution engine that did not provide the associated values, and are thus invalid. Version "1" dates from 1996, as indicated by the imagePath file name. If you write geometry solutions that do not provide this data, please use "-1" to indicate that it is not available.

5.8.1 solvedVars/fixedVar

The "solvedVars" field is intended to record which of the possible variable (azimuth, roll, tilt, fov, x, y, or z) were solved in the geometry solution engine. The information will be recorded as a set of single characters (the first character of each possible variable). Preferred order is "tarfxyz", but software that uses this entry should be lenient in accepting the characters in any order.

This is the same coding that should be followed in the "fixedVar" field in the "template" and other tables, with the assumption that the fixed variable will be one of 'x', 'y', or 'z'.

5.9 usedGCP

The table called 'usedGCP' records the GCPs used during the creation of a geometry solution. We do not record all the details of the GCP in this table, but refer to the GCP by its ID. The additional fields contain data specific to this use of the GCP in the geometry solution. Notice that there will be more than one usedGCP table entry for each solution, since more than one GCP is required to calculate a solution.

The table is defined as follows:

```
| usedGCP | CREATE TABLE `usedGCP` (
  `seq` int(11) NOT NULL,
  `gcpID` char(8) NOT NULL,
  `geometrySequence` int(11) NOT NULL,
  `U` double NOT NULL,
  `V` double NOT NULL,
  `timestamp` int(11) default NULL,
  UNIQUE KEY `ugcpIdx` (`gcpID`,`geometrySequence`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

And has the following field definitions:

seq – (M) the unique identifier for this used GCP table entry

gcpID – (M) the ID of the gcp table entry corresponding to this usedGCP entry.

geometrySequence – (M) the sequence number of the geomety solution this GCP was used in.

U – (M) the floating point U coordinate for this GCP in the image.

V – (M) ditto V. The U and V may be used to display the GCP on any image where the solution is valid.

A typical usedGCP entry might look like this:

```
                   seq: 2
                 gcpID: 'DRXX0003'
       geometrySequence: 54[15]
                     U: 309.4656
                     V: 454.5635
             timestamp: 85932820
```

[15] Astute readers will notice that this usedGCP was involved in the solution of the geometry used as an example previously. Why they know this is left as an exercise, but the placement of this footnote marker should be a good hint.

5.10 template

The "template" table defines a preselected region of a reference image that will be used to calculate an autogeom. The minimum and maximum U and V values bound the region, and the "I" field records the greyscale image values from that image. This data can be used to identify an shift in the current image compared to the reference, and thus new angles for the geometry may be calculated. The "template" table is defined as follows:

```
| template | CREATE TABLE `template` (
  `seq` int(11) NOT NULL,
  `id` char(8) NOT NULL,
  `name` char(32) NOT NULL,
  `imagePath` char(64) NOT NULL,
  `cameraID` char(8) NOT NULL,
  `usedGeometrySeq` int(11) NOT NULL,
  `user` char(12) NOT NULL,
  `whenDone` int(11) NOT NULL,
  `fixedVar` char(1) default NULL,
  `fixedVal` double default NULL,
  `timeIN` int(11) NOT NULL,
  `timeOUT` int(11) NOT NULL,
  `UMin` int(6) NOT NULL,
  `UMax` int(6) NOT NULL,
  `VMin` int(6) NOT NULL,
  `VMax` int(6) NOT NULL,
  `I` longblob,
  `timestamp` int(11) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

And has the following field definitions:

seq – the sequence number for this entry

id – an up to 8 character id for this template

name – up to 32 characters naming this template

imagePath – up to 64 characters identifying the image from which this template was extracted

cameraID – the id of the camera producing the image used for this template

usedGeometrySeq – the sequence number of the geometry used when creating this template

user – up to 12 character user name of the template creator

whenDone – when this template was created

fixedVar – which variable is fixed when converting U and V into real space ('x', 'y', or 'z')

fixedVal – the value of the fixed variable

timeIN – the time when this template becomes valid

timeOUT – the expiration date for this template

UMin,UMax – the minimum and maximum U coordinate for the template

VMin,VMax – ditto V

I – the intensity data for the template.

An example template:

```
                  seq: 1003
                  id: 'NH2011T'
                name: 'soloBushByBeach'
           imagePath: '1499378401.Thu.Jul.06_22_00_01.GMT.2017.nrthhead.c1.timex.jpg'
            cameraID: 'NH2X01C'
      usedGeometrySeq: 10700
                user: 'Holman'
           whenDone: 1.5115e+09
           fixedVar: 'y'
           fixedVal: 1000
             timeIN: 1.4993e+09
            timeOUT: 0
               UMin: 1474
               UMax: 1554
               VMin: 246
               VMax: 324
                  I: [79×81 uint8]
          timestamp: 1.5115e+09
```

5.11 usedTemplate

The "usedTemplate" table records information about the templates used in autogeom calculations. There are currently two kinds of "auto" geometry calculations: autogeom and crossgeom. Autogeom solutions use the fixed, predefined templates contained in the "template" table, while a crossgeom generates templates on-the-fly based on the images being correlated. The assumption for the crossgeom is that there is no fixed template information (otherwise you'd do an autogeom!) and you are using an image from one camera that has a geometry solution to calculate the geometry for an adjoining image. Because the templates are transient in nature, the "usedTemplate" table records different, albeit similar, information about the templates.

The "usedTemplate" table is defined as:

```
| usedTemplate | CREATE TABLE `usedTemplate` (
  `seq` int(11) NOT NULL,                      (auto)
  `templateID` char(8) default NULL,           (A)
  `whenDone` int(11) NOT NULL,                 (AC)
  `geometrySequence` int(11) NOT NULL,         (auto)
  `similarity` double default NULL,            (AC)
  `contrast` double NOT NULL,                  (AC)
  `refUMin` int(6) default NULL,               (C)
  `refUMax` int(6) default NULL,               (C)
  `refVMin` int(6) default NULL,               (C)
  `refVMax` int(6) default NULL,               (C)
  `UMin` int(6) default NULL,                  (AC)
  `UMax` int(6) default NULL,                  (AC)
  `VMin` int(6) default NULL,                  (AC)
  `VMax` int(6) default NULL,                  (AC)
  `fixedVar` char(1) default NULL,             (C)
  `fixedVal` double default NULL,              (C)
  `timestamp` int(11) NOT NULL                 (auto)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 |
```

```
Note: The (X) entries at the end of the line describe which operation produces this data. 'A' means "aut
```

And has the following fields:

seq – the unique sequence number for this used template

templateID – the ID of the template used for an autogeom, empty for crossgeom

whenDone – when this template was used in the solution

geometrySequence – the sequence number of the geometry produced using this template.

similarity – the quality metric for the fit of this template to the image

contrast – the contrast of the template in the solution image – the standard deviation of the intensities.

refUMin,refUMax – the minimum and maximum U coordinate for the reference (other) image for crossgeoms. Empty for autogeoms.

refVMin,refVMax – ditto for V

UMin,UMax – the minimum and maximum U coordinate for the image being geometried.

VMin,VMax – ditto V

fixedVar – a one character flag indicating which variable ('x', 'y', or 'z') was fixed in the crossgeom solution

fixedVal – the value for the fixed variable

Because there are two uses for this table, there will be two examples shown. First, for an autogeom:

```
                seq: 193
         templateID: 'NH2X03T'
           whenDone: 1520034256
  geometrySequence: 83952
         similarity: 3.295
           contrast: 0.934
            refUMin: 0
            refUMax: 0
            refVMin: 0
            refVMax: 0
               UMin: 1235
               UMax: 1285
               VMin: 666
               VMax: 696
           fixedVar: ''
           fixedVal: 0
          timestamp: 1520034260
```

Note that the 'ref' coordinates are empty. These may be retrieved from the "template" table through the recorded templateID. Also, the fixedVar and fixedVal are contained in the template referenced by the templateID field.

For a crossgeom:

```
                seq: 199
         templateID: ''
           whenDone: 1520034856
 geometrySequence: 83952
         similarity: 0.895
           contrast: 3.934
            refUMin: 1230
            refUMax: 1280
            refVMin: 669
            refVMax: 699
               UMin: 235
               UMax: 285
               VMin: 666
               VMax: 696
           fixedVar: 'z'
           fixedVal: 0
          timestamp: 1520036260
```

Because the templateID is blank, and the ref coordinates are non-zero, we know this was a transient template created during the crossgeom calculation.

6 Database Access Functions – argusDB

We have now covered all of the original database tables as currently implemented. The code for accessing the database was written in a manner that it is simple to create new tables for other uses. The CIL actually has several other tables in use, such as predicted and measured tide tables for site with that data. Some functions are relatively specific in use, doing very limited, specific jobs. For example, "DBGetCurrentGeom" takes information about a station and camera and retrieves the geometry valid at that time.

To help prevent namespace collisions in Matlab and with Matlab, the CIL/CIRN database toolbox functions begin with the two uppercase characters "DB".

6.1 How Do I Know What A Table Needs To Contain?

There is a lot of information in Section 5. Look there.

As an aide in programming using this database code, a function that will return a default, empty structure that is appropriate for use for the table of interest has been written. If you are using the database, it will actually query the MySQL database itself to return the information. A non-database version will be written so non-database users can write compatible code. The function is:

```
emptyTable = DBCreateEmptyStruct('tableName');
```

Where "tableName" is the name of the table you want an empty struct for, and "emptyTable" is a Matlab structure with default field name/field value pairs. For example:

```
DBCreateEmptyStruct('station')
ans =
```

```
       seq: 0
        id: ''
 shortName: ''
      name: ''
    siteID: ''
    timeIN: 0
   timeOUT: 0
 timestamp: 0
```

Notice that the character entries in the "station" table are returned as empty strings while numeric values return a scalar '0'. Any entry that is expected to be a matrix (for example, the "K" field of the camera struct) will be returned as an empty matrix.

It is expected that you will be able to start with a call to DBCreateEmptyStruct, fill in the appropriate fields, and any subsequent insert into the database should be successful. Of course, if you put garbage into the struct, you may expect garbage out later.

6.2 DBGetTableEntry

The vast majority of users and uses of the database will involve retrieving data. Almost all data query functions in the Argus database toolbox are based on this one function. The query data may be massaged in various ways before this function is called, or there may be multiple queries made to trace down a relation[16].

The function of "DBGetTableEntry" is exactly as it sounds: retrieve one or more table entries from the database. It is limited to one table at a time, and the name of the desired table is the first parameter to this function. Any additional parameters are interpreted as search terms within that table, usually in pairs of "field name", "field value". In general, the function is defined:

```
output = DBGetTableEntry( tableName, fieldName1, fieldValue1, ...
                fieldNameN, fieldValueN [, optionalRawCondition] );
```

There can be zero or more[17] "field name/field value" pairs. For example:

```
geom = DBGetTableEntry('geometry','cameraID','NH2X01C');
```

This call will return all data from the table 'geometry' where the cameraID value is equal to 'NH2X01C'.

The default condition for field name/field value pairs is "equal" for numeric values, or "like" for character strings. "Like" in SQL is like "equal", except the query may contain wildcards ("%") within a string condition.

[16] For example, DBGetCameraByStation, queries the "station" table to locate the station "id" value, then uses this to query the camera table.

[17] The code has not been tested with ten thousand pairs, so you're on your own there.

6.2.1 Special field names

If you are using the DBGetTableEntry function, you will almost always be refering to an existing field name in the table structure. However, there is a special case field name 'time' which has been designed to perform some special processing. If you use the field name 'time', then the field value will be a Unix epochtime that refers to either the field "epoch" in the table (if there is one, which there is not in the basic tables already described), "timeIN/timeOUT" otherwise. In the latter case, the data returned will be what is valid at the time specified. For example:

```
temps = DBGetTableEntry('template','time', 1520000000);
```

will result in the full SQL query[18]:

```
select * from template where timeIN <= 1520000000
    and ((timeOUT=0) or (timeOUT>=1520000000));
```

6.2.2 Field Value types

Field values in a DBGetTableEntry call may have one of several types. The following describes the effects on the generated SQL query for each of the possibilities.

Scalar	If the field value is a single number (integer or floating point) the query will ask for all data where the field name is EQUAL to the specified field value.
1x2	When provided two numbers, the query will match any data where the named field is between the two values, endpoints included. I.e., less than or equal the maximum of the parameter and greater than or equal to the minimum.
1xN	With more than 2 values, the query becomes an IN condition. That is, the field value must match any of the values provided.
'abc'	A string will be matched exactly.
'abc%'	The % character is a wildcard in character matching, and any table entry with a field value that begins 'abc' will be matched.

6.2.3 Examples

```
DBGetTableEntry('camera','cameraID','NH2X01C')
```

Returns any entry in the "camera" table where the camera id is equal to the string 'NH2X01C'. This may result in more than one returned struct if there is more than one camera with that id.

```
DBGetTableEntry('camera','cameraNumber',3)
```

Returns data about every camera in the database that has been camera number 3. This will return all camera 3's from every station.

```
DBGetTableEntry('camera','cameraID','NH2X%','cameraNumber',3)
```

[18] If you are interested in such things, you can see the last database query in native SQL format by declaring the Matlab global variable 'DBlastquery'. This variable is set by the function that performs the final call to the jave database interface. However, some functions make additional calls after the initial query, and you will see only the last query.

Returns data about all cameras that have been camera number 3 and have the camera id beginning with "NH2X". Because of the id conventions discussed previously, these will all be from the current station at North Head (NH2XXXXX).

```
DBGetTableEntry('camera','cameraID','NH2X%','cameraNumber',[2 4])
```

Returns all camera information for cameras 2, 3 and 4 at North Head.

```
DBGetTableEntry('camera','cameraID','NH2X%','cameraNumber',[-1 2 4])
```

Returns data for cameras 2 and 4 from North Head. Because there are three elements in the field value parameter the query will be "IN". There are no camera number "minus 1", so that value in the parameter is effectively null, serving only to force an "IN" instead of a range comparision. This is a very handy trick if you have a two-element IN query you want to make.

6.2.4 Optional Raw Condition

Because the function is defined to take the table name as the first parameter, and then field name/field value pairs, it can expect an odd number of input arguments. If it gets an even number, then it assumes that the last parameter is a literal string that will be appended to the database query. This allows you, the user, to be quite creative in designing queries while remaining within the normal CIL/CIRN Argus format. (There are ways to be even more creative, covered later.)

```
DBGetTableEntry('site','where lat>45')
```

Returns information about every site where the latitude of the site is greater than 45 degrees. Note that because I am providing the raw query string here, I must include the "where" portion of the SQL query. DBGetTableEntry will not insert that for me.

```
DBGetTableEntry('site','id','%','and lat>45')
```

This results in the same answer, but note that I am now using 'and' to join my condition to the field name/field value condition of "id" equal to "%" (anything). The difference is that with no field name/field value pairs as in the previous example, DBGetTableEntry will not insert the "where" part of the SQL query for me. A query with no name/value pairs does not require a "where" clause. However, once I include a name/value pair, even a useless one like in this example, DBGetTableEntry will include the "where" portion of the query and I must provide a logical connection between the first condition ("id like '%"') and mine ("lat>45"). This is the "and" that is part of the raw condition.

```
DBGetTableEntry('site','lat',[45 999])
```

This query is almost the same as the last two examples. Can you figure out what the difference is?[19]

[19] The raw query used the '>' greater than condition. The two-element numeric value will result in the query "lat>=45 and lat<=999". Assuming that nobody would put anything greater than 90 degrees (the north pole) in the site "lat" value, then the only difference between the three examples is that the last one will return site data for a site that is at exactly 45 degrees north latitude while the first two will not.

6.3 DBConnect

DBConnect is the function you will use to connect to the Argus database server. You can provide many of the parameters such as server address, user name, database, and even internet port, or you can allow the function to use either the stored or default values. Default values can be set in system environment variables, but it is probably better to use specific values in your startup.m or other code. Once DBConnect has been successfully called, it will record the connection parameters and use them if you do not provide them on the next DBConnect call.

There are two forms of calling this function. The first makes use of the Matlab convention that a function called with string inputs and no output can be entered on the command line without parantheses or quotes.

```
DBConnect servername username password database [port]
```

Where "servername" is the host name of the database server, "username" is the assigned username, "password" is the assigned password[20], and "database" is the name of the database being used. The standard database name for CIL/CIRN Argus is "argus", although you can create your own databases and use the database toolbox by simple entering the name of your database at the DBConnect call. Also by default, the default username is "default", and has no password. This user should be granted read-only permission to your database (and has been for mine). You can add users with more permissions (such as write capability) using the DBCreateUser function.

The final parameter, port, is optional, and will default to 3306.

For example:

```
>> DBConnect blossom.coas.oregonstate.edu default '' argus

    ans =

        server: 'blossom.coas.oregonstate.edu'
        user: 'default'
        password: ''
        database: 'argus'
        port: 3306
```

The first four fields of that 'ans' struct are the same as the parameters I passed to the call. The last field is the internet port number for the MySQL server, which defaults to 3306. If you want to run isolated database servers on the same computer, or need to bypass various security protocols that block specific ports, you can specify the actual port number. To do that, you would use the second form of the call:

```
DBConnect( ans )
```

[20] A note about database passwords. This database is not considered a high-security activity, and user accounts are intended to prevent unintended consequences (i.e. accidental deletions or insertions), not outright vandalism. The database itself stores any user password as an encrypted string, but the Matlab code here does not. It is kept in the clear in a bit of global memory once it has been entered, and is shown in the clear as the result of a DBConnect or DBInfo call. You are strongly advised not to reuse a "real" password here.

Where 'ans' is the same struct that is returned by the DBConnect call with the appropriate values set. So, if I wanted to change databases on the same server, I could use the sequence of commands:

```
DBConnect blossom.coas.oregonstate.edu default '' argus
ans.database = 'new_argus';
DBConnect( ans )
```

You can also create your own database connection structure by setting the parameters directly. You must do this if the defaults will not allow a connection, since DBConnect will not return that structure when it fails to connect. I.e., if you are behind a firewall where port 3306 is blocked, you must either use the first command line version including the optional 'port' parameter, or you must create the structure. E.g.:

```
db.server = 'blossom.coas.oregonstate.edu';
db.user = 'default';
db.password = '';
db.database = 'argus';
db.port = 8080;
DBConnect( db );
```

NOTE: because DBConnect will simply reconnect to the last known server with the last known authorization details, it can be called at any time to ensure that the connection is still active. If you leave a session for any significant amount of time, it is best to DBConnect again before using the database. There is an obscure issue where an inactive session will time out and the normal error handling in the Argus database toolbox will not manage to get the connection restored the first time.

6.4 DBInfo

An adjunct to the DBConnect function is DBInfo, which simply returns the same connection information that DBConnect does, without changing anything. This function makes use of the global structure "DBinfo"[21]which is also used by DBConnect to record connection information. If you change fields in DBinfo and call the function DBConnect with no parameters, your changes will be used in reconnecting to the database server.

6.5 DBSelectDB

This call allows you to switch databases easily. The standard Argus database is "argus". You may create your own databases for data or testing, and this function will allow you to acces those in place of "argus". This function demonstrates the "change DBinfo and call DBConnect" method mentioned in section 6.4.

```
DBSelectDB('newArgusTest');
```

6.6 DBCreateUser

The initial creation of the database must also create a root or administrative user that has all database permissions. This is done while creating the database using the MySQL system. At the same time, this user can be assigned a password. Once that user has been created, then

[21]Note the non-standard case (i.e., not DBInfo). This is to differentiate the structure from the function, but to also prevent namespace collisions (duplicate names) when you write Matlab code for the database. This is also why the global "DBlastquery" has a lower case "l".

the Argus DB Matlab function DBCreateUser can be used to add other users. This function has the usage:

```
DBCreateUser( name, password, [level], [db] )
```

"Name" is the desired user name. "Password" is the password for that user. The optional "level" parameter defines the status of the user. When this parameter is omitted, or is 0, the user will have "read only" access. A level 1 user will have sufficient permission to insert data. The final optional parameter is "db", which is the name of the database the user access is being set up for. It will default to "argus".

It is highly recommended that the first user that is created is the user "default" with an empty password. This creates a global read-only user that can read anything from the database. Then specific logins for each user that you want to be able to insert data should be created.

It is also highly recommended that users operate as "default" until they actually need to insert data, then switch into read/write mode only long enough to do the insert, and then switch back to the default "read only" mode.

So, for example:

```
DBCreateUser( 'default', '' );
DBCreateUser( 'stanley', 'agoodpassword', 1 );
```

6.7 DBCreateTriggers

This is another adminstrative user function. It is used to create the "triggers", or functions that the MySQL database server runs, associated with a specific table.

There are two trigger functions used in the Argus database. The first is a "before insert" trigger (run prior to any insertion of data into the table) that is intended to create a unique sequence number and apply that to the "seq" field. The other is a "before insert" that updates the "timestamp" field with the current epoch time. Both triggers are controlled by entries in another table. The "autoSeq" table contains one flag for each table, as does the "autoTime" table. A "1" in the value for field named after the table means "do", a "0" means "do thou not".

If you create a custom table and want to take advantage of the automated inserts, use the following command:

```
DBCreateTriggers( 'tableName');
```

Where "tableName" is the name of your table. For example, "geometry".

There are four helper functions to manage trigger actions. DBEnableXXXTrigger and DBDisableXXXTrigger, where 'XXX' is either 'Seq' (for sequence generation) or 'Time' (for 'timestamp' generation). Using these functions requires level 1 (not default) database authorization.

Critical note: because the Matlab database code is not informed of the automatically generated values, and the sequence number for every geometry must be known immediately after insertion, the "geometry" table should never have the automated sequence number trigger enabled. The function DBInsertGeom manages the geometry table sequence numbers internally.

6.8 DBInsert

This function provides the basic database insert. It takes two parameters, the table name and the structure to be inserted. It returns a copy of the inserted structure. Any values updated by the database triggers will not be updated, but geometry table entries will return the new sequence number.

```
newCam = DBInsert( 'camera', camNew );
```

Will insert the camera information in camNew into the camera table. If you wish to see the values for "seq" (sequence) and "timestamp" you will need to do a retrieve on this row. Because the 'id' must be unique for each camera, you can use that value to retrieve the data.

```
newCam = DBGetTableEntry('camera','id',camNew.id);
```

6.9 DBInsertGeom

This is the function you will use to insert a geometry into the database. It will create a unique sequence number automatically, and return the geometry structure with this new sequence number. It will also optionally insert the usedGCP or usedTemplate strutures associated with that geometry.

```
newGeom = DBInsertGeom( newGeom, usedThings );
```

"newGeom" must be a geometry structure, and "usedThings" must be an array of either usedGCP or usedTemplate structures, depending on the value of the "isAuto" field in the geometry struct. (0 requires usedGCP, 1 requires usedTemplate. There are helper functions named "DBInsertUsedGCP" and "DBInsertUsedTemplate" which this function calls, or you may use those functions yourself if you desire. Letting DBInsertGeom do it is easier.

6.10 DBGetSequence

Most functions that require a new sequence number already ask for one. When you write a new database function and need to create a unique sequence number for a table, this function will access the database to create one. The database manages a table of sequence numbers and uses an internal function that returns the next sequence number and increments the stored value in one step[22].

```
newseq = DBGetSequence('camera');
Warning: UPDATE command denied to user 'default'@'gudea' for table 'sequence'
```

Oops. Because this function requires write access to the database, it will not work properly for "default" or "0 level" (see section 6.6) users. Since it is used to create a sequence number prior to insertion of data into the database, this is not a significant issue. Simply connect as a "write" or "1 level" user before creating the sequence and then insert the data.

[22] This is called an "atomic operation" in computer science terminology. That is, it is a single, indivisible operation based on the ancient idea that an "atom" was indivisible. When this function is used nothing else can happen between retrieving the sequence number and incrementing the stored value. If you were to do this as multiple database queries ("get number", "add one", "update number") it is possible another database user could sneak in a "get number" query after you did but before you "update number", and thus you would both have the same "unique" sequence number. Us gurus are paid to think of such stuff for you.

6.11 DBGet–Something

There are a significant number of functions whose names begin with "DBGet" that have not been discussed yet. Each of these functions has been written to make getting specific kinds of information from the database easier for the user. For example, "DBGetCameraByID" takes a camera ID and returns the database entry for that camera. It also adds the camera model, lens model, and station information to that camera struct. "DBGetCameraByImage" takes a standard Argus file name, e.g.:

```
809020845.Mon.Aug.21_16_00_45.GMT.1995.argus00.c0.snap.jpg
```

splits it into pieces (using "parseFilename"), and looks up the camera at that station at that time with the correct camera number.

Perhaps the most useful function in this category is "DBGetCurrentGeom". It takes the camera ID and the desired epoch time as input and returns the geometry that is valid for that camera at that time. E.g.:

```
g = DBGetCurrentGeom('NH2X01C', 1545893213);
```

6.12 DBToBlob/DBFromBlob

These two functions are used by the database to manage the storage of indeterminate-sized Matlab matrices. Standard database entries consist of a scalar numeric value or a character string[23]. Open-ended storage of a matrix of numbers is not one of the native types.

To deal with arbitrary data of indeterminate size, databases have a type called "blob", for "binary large object". MySQL stores whatever you want there. Creating blobs is up to the user; in the Argus database they are created by the function DBToBlob, and decoded by DBFromBlob when retrieved by the database code. Use of these functions is not determined by the user, but based on the table definition. For example, if you refer to the camera table (section 5.6) you will see that the fields "K" and "kc" are both defined as "tinyblob". This is a "large object" that can be up to 256 bytes long. This is more than adequate to contain the 3x3 double "K" or the 1x5 double "kc".

Both blob functions are written in C using the MEX interface to Matlab, and convert the raw binary matrices into a binary form suitable for insertion into a blob, or decode the binary from the database back into a Matlab matrix. This includes the dimension (3x3, 1x5, etc.) and data type information (double, uint8, etc). When the blob value is inserted into the database, it is prefixed with a specific byte value in the first location that signals that this is DBToBlob-format data. Because of this, what comes out of a blob from the database should be identical to what went in. Unfortunately these functions cannot handle struct data, so structs cannot be stored in that format in the database.

These are currently the only two C/MEX functions in the Argus database code. Future versions will convert arbitrary data into XML[24]and back removing this complication. While the current blob encoding is excellent for Matlab use, it is not a standard encoding and thus languages such as Python cannot access the blob data without further effort.

[23] MySQL has several other data types native to the database, but none of them deal with matrices.

[24] eXtensible Markup Language, which is commonly used to store hierarchical data. It is pure text, and there is code that will create text from numeric matrices as well as structs, and convert that text back into something similar to the original. Careful design will be applied so that "similar" is "good enough".

DBToBlob returns a 1xN uint8 matrix. DBFromBlob consumes a 1xN uint8 matrix and returns the original data. for example:

```
>> blob = DBToBlob('a string')
  blob =
     1×32 uint8 row vector
  Columns 1 through 18
   0   0   0   4   0   0   0   2   0   0   0   1   0   0   0   8  97  0
  Columns 19 through 32
  32   0 115   0 116   0 114   0 105   0 110   0 103   0
>> DBFromBlob(blob)
  ans =
     'a string'
```

7 Common Errors

There are several common errors you may come accross while using the Argus database toolbox. Almost all of them are fatal, since almost everything that can be handled by the code is. The most common errors, in no particular order follow.

7.1 Access Denied

You are not authorized to access that database using that username or password.

```
>> DBConnect blossom god '' argus
Error using DBConnectRaw (line 89)
Connect failed.
Error was: Access denied for user 'god'@'gudea' (using password: NO)
```

Note that the user name used to connect to the database consists of the username as entered to the DBConnect command with the local host name appended ("gudea" is the host running Matlab.) When DBCreateUser creates a user, it authorizes "user@%", where the "%" indicates that the user may come from any host. If you create users manually, remember which system you wish them to be able to access the database from, or use "%" for a wildcard.

If the user enters the wrong password, the same error will be returned, with the difference that it reports "using password: YES". If the database does not exist, this is the error that is returned.

7.2 Communications Link Failure

If the database server is not running on the host you specify, or there is a firewall blocking your access to that server, you will get this error.

```
>> DBConnect blossom default '' argus
Error using DBConnectRaw (line 89)
Connect failed.
Error was: Communications link failure
```

It is VERY common for some service providers, especially those for domains that end in .MIL or .GOV, to block access to almost everything. This includes MySQL. There are two possible solutions. First, as a .MIL or .GOV user, you can file the extensive paperwork and wait a long time for your firewall administrator to deny your request to open your access to port 3306. (This is the default port for MySQL.)

Or, much simpler, communicate with the CIL database server using the alternate port 8080. The administrator of that server has created this alternate port for just such problems. To use port 8080 instead of the default, either add '8080' to the end of the DBConnect command (e.g. "DBConnect blossom default " argus 8080"), or use the function version of DBConnect and pass it a struct with the appropriate settings. See Section 6.3 for more information.

7.3 No Operations Allowed After Statement Closed

What an arcane error message. It is part of a half a screen of dangerous red text. What it means is that your connection to the database has timed out. The Argus database code has tried to reopen the connection, because that was the first error that occurred but was caught by the database code. After reconnecting, the query is retried but, unfortunately, the "statement is closed" and cannot succeed.

The current workaround is to simply redo the query. The connection will have been re-established and the query will reopen the statement. This is an open issue.

```
>> DBGetTableEntry('site')
Error using DBQueryRaw (line 52)
Java exception occurred:
java.sql.SQLException: No operations allowed after statement closed.
```

Index