

Paper SD02

The Twain Shall Meet: Facilitating Data Exchange between SAS® and Matlab

Dimitri Shvorob, Vanderbilt University, Nashville, TN

Abstract

Intended for the audience of SAS programmers familiar with Matlab, this report outlines an attractive method of exchanging data between the two applications, employing a MySQL database as conduit.

1. Introduction

The author's experience suggests that many SAS programmers know of and use Mathworks Inc.'s Matlab software. Such 'bilingual' programmers are able to assess each package's features relevant to the task on hand, and pick the tool offering greater convenience. Occasionally, a project includes a component easily accomplished in SAS, and another that is more amenable to Matlab. The programmer is tempted to adopt a 'mix-and-match' tactic, but has to consider the overhead of passing data from one application to the other, and possibly back.

In absence of suitable conversion software, such as Stat/Transfer of Circle Systems Inc., one has to rely on SAS's and Matlab's own export/import capabilities to accomplish data exchange. Direct transfer is ruled out, as SAS cannot read data stored in Matlab's mat format, nor can Matlab read a sas7bdat dataset created by SAS. It is possible, however, to pass data through a temporary file of a third-party format, which can be read from, and written to, by both SAS and Matlab. In practice, one has a choice between a text file and an Excel spreadsheet. Both can be handled by SAS with PROC EXPORT and PROC IMPORT, usually without problems. Outside of SAS, things get more complicated.

The Excel way, accommodated by Matlab functions xlsread and xlswrite, is normally the clear choice. Unfortunately, Excel's involvement imposes limits on the size of a SAS dataset (or Matlab array) that can be moved in a single pass. Size of an Excel 2002 spreadsheet, for example, is limited to 65,536 rows and 256 columns. To transfer a larger block of data, one needs to break it up into segments of admissible size, and re-assemble them at destination.

Transfer through a text file - in SAS, one can write a text file with EXPORT procedure or in DATA step with put, and read from a text file with PROC IMPORT or input - allows larger file sizes, but is quite cumbersome. Unless the data being transferred are purely numeric, one has to use Matlab's low-level read/write functions textscan and fprintf to access the 'pass-through' text file, in the process

setting up format strings similar to those of SAS's put and input statements. Getting missing values written to a text file correctly remains a challenge; ultimately, one may have to temporarily replace missing values with a numeric code, and perform a reverse recode later. Notably, whereas xlsread lets one retrieve selected rows or columns from a spreadsheet - e.g. read cell range 'A1:A10000' - textscan insists on reading all of file's rows, which can be a problem if the file is very large.

This paper suggests a third way: transfer through a MySQL database. Easy to set up, MySQL-mediated data exchange has four advantages.

a) Convenience

Connection between SAS and a MySQL database is established with a simple libname statement, setting up the database as an 'external' SAS library. To transfer a SAS dataset to a MySQL database (or vice versa), one can use PROC COPY, or open SAS Explorer window and drag-and-drop the icon associated with the dataset from one library to the other. Once in MySQL, the data are accessible to Matlab and can be fetched into its workspace with an SQL query. Individual variables can be selected, and individual rows retrieved with where filter - a fully flexible way to extract data, unavailable with either textscan or xlsread. Whereas textscan and xlsread place retrieved data into a single cell array, so that one has to break up its columns into smaller arrays, corresponding to Matlab variables (name = X{1}, age = X{2}, etc.), and cast cell arrays with numeric data to a numeric type (e.g., age = cell2mat(age)), MySQL makes this unnecessary. Likewise, one needs not cast numeric arrays to cell, and merge many cell arrays into one, when taking data out of Matlab. Finally, column names are easily retrieved and preserved through data transfer, in contrast to the text-file and spreadsheet alternatives.

b) High capacity

A full-fledged database management system, MySQL is designed to store and manipulate large volumes of information, and can easily handle any amount of data generated by either SAS or Matlab.

c) Robustness

Though generally effective, text-file and spreadsheet methods are, in the author's experience, not 100% reliable, and one is advised to inspect their output for possible errors, such as corrupted column types and values. Testing of MySQL-aided data transfer has not encountered similar problems.

d) Expanded functionality

The most impressive feature of the MySQL conduit is the ability to manipulate data that reside in a database from either SAS or Matlab. (Access is especially easy in SAS: it is generally true that one can set up a DATA step involving a MySQL table, or supply a MySQL table to a procedure, using syntax identical to that which would be required if a native SAS dataset were involved. In Matlab, one manipulates a MySQL table by passing commands to MySQL, as if working with MySQL Command Line Client). What this means for data transfer is that in some cases, it can be reduced by a step or avoided altogether, by placing data into MySQL and leaving them there, for SAS and Matlab to use.

We walk through the setup procedure in Section 2 (see Appendix I for download links), and in Section 3 illustrate the proposed method with a simple exercise.

2. Setup

MySQL-mediated data transfer can be accomplished with either of the following three components ('interfaces') of SAS/ACCESS: MySQL, ODBC, or OLE DB. In what follows, we will concentrate on the first two.

Before going ahead with installation, it makes sense to check whether a suitable SAS/ACCESS component is, in fact, present. PROC SETINIT may not be of help, as it displays SAS modules that are licensed, rather than installed. Instead, submit

```
libname x mysql database = y;  
libname x odbc dsn = y;
```

to SAS, and inspect the log for 'Engine cannot be found' error messages.

2.1. Installing MySQL

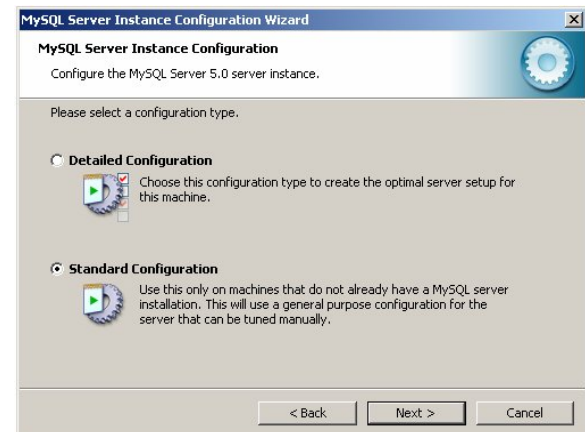
Download and run the installer module of MySQL Server 5.0 (Windows Essentials package), selecting 'Typical Install' in 'Setup type' screen,



skipping sign-up in 'MySQL.com Sign Up' screen,



and marking checkbox 'Configure the MySQL Server now' in 'Wizard completed' screen. Select 'Standard configuration' in 'Configuration type' screen,



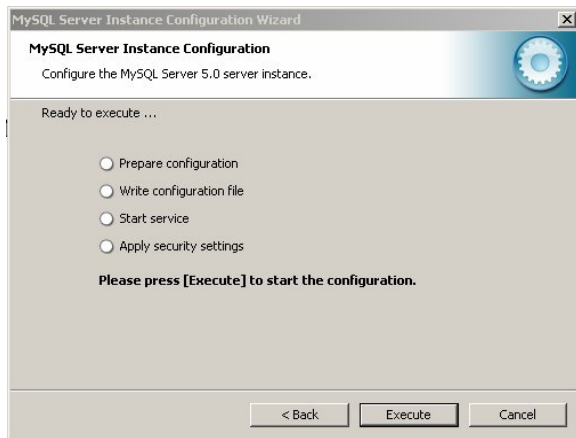
and mark 'Install as a Windows service' checkbox in 'Windows options' screen. 'Include Bin Directory in Windows PATH' does not need to be marked.



It is up to you whether to password-protect data stored in MySQL, or 'Create an Anonymous Account' instead, in 'Security options' screen. We choose to establish a password, and select 'ake la'.



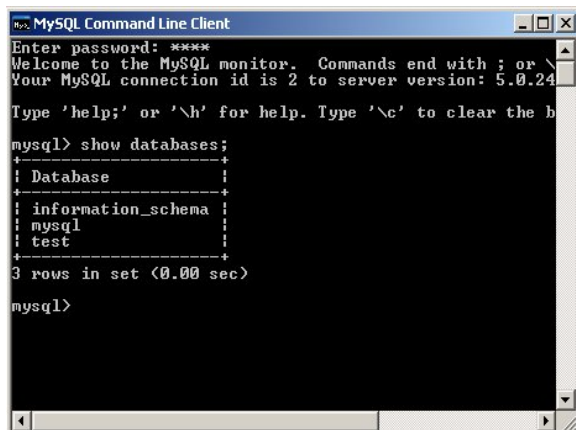
Complete the installation by pressing 'Execute' button in 'Execute configuration' screen.



You can check that MySQL was installed and its instance is running on your PC, by locating mysqld-nt.exe in the 'Processes' list of Windows Task Manager, or by navigating Windows taskbar: Programs > MySQL > MySQL Server 5.0 > MySQL Command Line Client. After keying in your password, or hitting 'Enter' if none was selected, you can type

`show databases;`

to display available databases.



Databases `information_schema` and `mysql` store system data and are best left alone; in Section 3, we will use the empty starter database `test`.

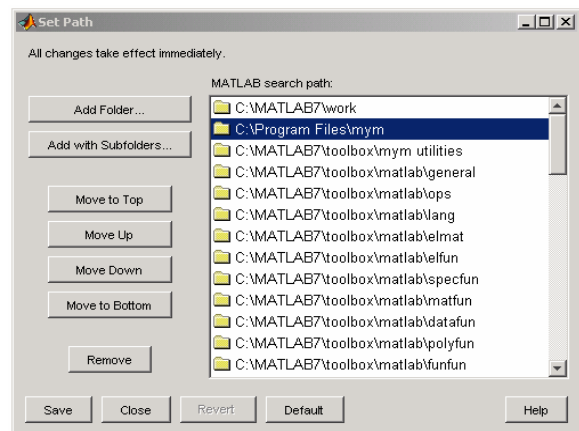
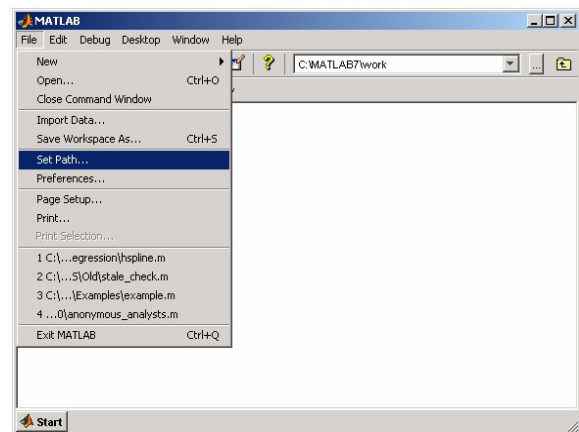
2.2. Connecting MySQL to Matlab

Matlab functions reading from and writing to MySQL include `mym.m` by Yannick Maret, and a set of utilities based on `mym.m`, written by the author.

Download and run `mym.m` installer.

Download `mym.m` utilities.

Add locations of downloaded m-files, including `mym.m`, to Matlab's working path, as shown below.



We can test the link between Matlab and SAS by submitting

`myopen('localhost','root','akela')`

Matlab will attempt to connect to the running MySQL instance and, if successful, display a message from `mym.m`.

mYm v1.0.8, Copyright (C) 2006, Swiss Federal Institute of technology, Lausanne, CH

mYm comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are
welcome to redistribute it under
certain conditions.

We can once again inspect the list of available
databases, this time from Matlab.

```
dblist

ans =

    'information_schema'
    'mysql'
    'test'
```

2.3. Connecting MySQL to SAS

a) with SAS/ACCESS Interface to MySQL

In SAS, submit

```
libname dbtest mysql database = test
user = root password = akela;
```

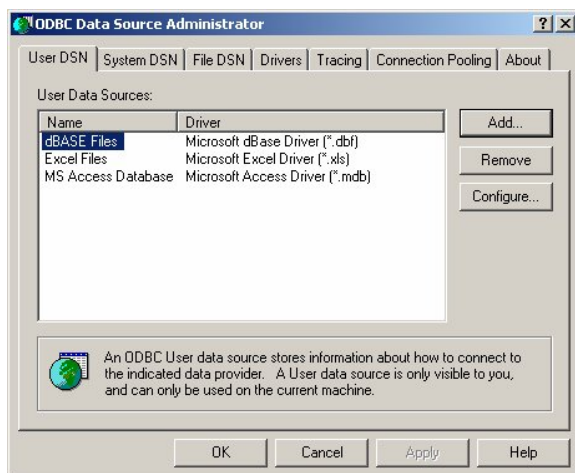
(Omit password option if working with an anonymous
account).

b) with SAS/ACCESS Interface to ODBC

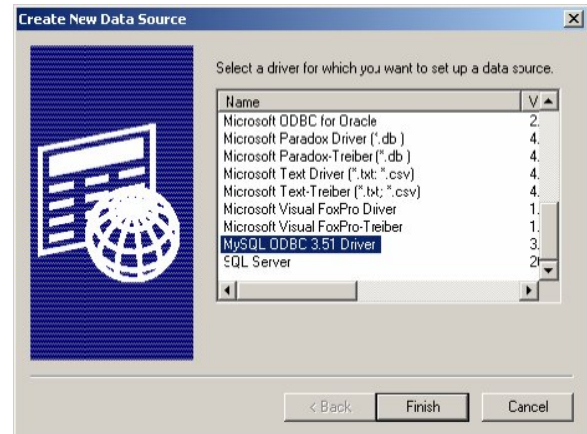
Here, we need to install the ODBC driver for MySQL,
and set up the selected MySQL database, test, as
an ODBC 'data source'.

Download and run the installer module of MySQL
ODBC driver, selecting 'Typical Install' in 'Setup Type'
screen. (No further configuration is needed).

Open Windows Control panel, navigate to
'Administrative Tools' section, and click on 'Data
Sources (ODBC)' icon. 'ODBC Data Source
Administrator' window appears, tab 'User DSN' active.



Press 'Add', select 'MySQL ODBC 3.51 Driver' from
the list of available data sources, and click 'Finish'.

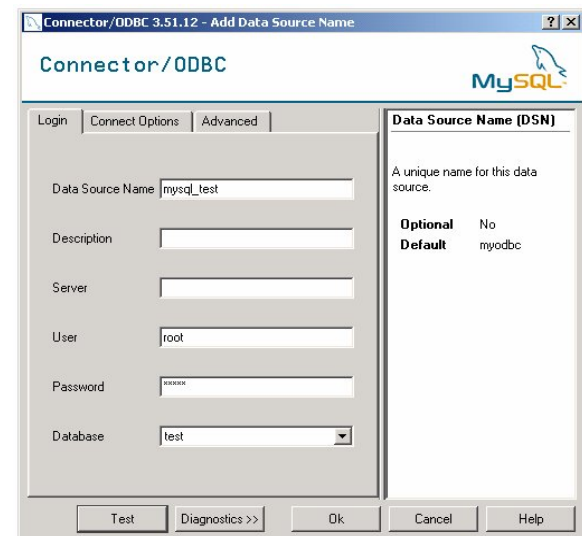


In MySQL Connector/ODBC configuration screen,
with 'Login' tab active,

Enter 'root' in field 'User', and 'akela' in field
'Password'. (Skip this step if MySQL is not password-
protected).

Select 'test' from the drop-down list in field
'Database'.

Assign a name (e.g., mysql_test) to the data
source, by entering it in field 'Data Source Name'.



Finally, make test accessible to SAS with

```
libname dbtest odbc dsn = mysql_test
user = root password = akela;
```

3. Test drive

To see MySQL-mediated data transfer in action, we
take up an exercise from the realm of finance.

A European call option, written on a stock, grants its
holder the right to buy the stock at a fixed price ('strike

price') on given future date ('expiration date'). In the early 1970s, Fisher Black and Myron Scholes showed how to value an option if - one of many assumptions! - the stock price follows a simple stochastic process, geometric Brownian motion.

$$dS/S = \mu dt + \sigma dW$$

The famous Black-Scholes formula gives the option price as a function of current stock price, option's strike price and time to expiration, risk-free interest rate, and return volatility σ . By far the most important input, volatility is unobserved. Trader looking for the 'right' value of σ to plug in can estimate it from past stock prices ('historic volatility'), or infer the value *implied* by current option prices ('implied volatility'), assuming that those are derived with Black-Scholes. Notably, if the assumption is correct, implied volatilities backed out from multiple quotes have to be the same, bar some random noise. Is it something that the trader would actually find?

Armed with a SAS dataset of option prices and characteristics, pertaining to a single stock, having the same time to expiration, and collected on a single day - with this, what's left to vary if the strike price - we are ready to put ourselves in his shoes.

At this point, we realize that SAS does not have a function to compute the implied volatility, nor, indeed, the Black-Scholes formula itself. A closed-form expression for the implied volatility is not available, making it necessary to set up and solve the non-linear equation defining σ_{IMP} - with PROC MODEL, for example. Alternatively, we can call on `blsimpv` function of Matlab's Financial Toolbox.

Variable	Type	Format	Label
CRDATE	Num	YYMMDDN8.	Current date
EXDATE	Num	YYMMDDN8.	Expiration date
OPRICE	Num		Option price
RATE	Num		Interest rate
SPRICE	Num		Current stock price
STRIKE	Num		Strike price

Database test was made accessible to SAS earlier; opening SAS Explorer window, you can find `dbtest` among the session's libraries. (Double-clicking on `dbtest` reveals that the library, i.e. the database, is empty). We transfer dataset `example.sas7bdat` to test with

```
proc copy
  in = sas
  out = dbtest;
  select example;
run;
```

Although Matlab has established a connection with the MySQL instance, no database was selected for use at the time, which can be confirmed by entering

```
dbcurre
```

```
ans =
```

```
Empty string: 1-by-0
```

We open database test with

```
dbopen('test')
```

and verify that table `example` is visible to Matlab, and has the expected structure.

```
tblist
```

```
ans =
```

```
'example'
```

```
[names,types] = tbattrib('example')
```

```
names =
```

```
'OPRICE'
'SPRICE'
'STRIKE'
'RATE'
'CRDATE'
'EXDATE'
```

```
types =
```

```
'double'
'double'
'double'
'double'
'date'
'date'
```

Retrieving contents of `example` to Matlab with

```
[oprice,sprice,strike,rate,crdate,
exdate] = mym('select * from example');
```

- note that Matlab variable names are case-sensitive, while MySQL column names are not - we confirm that columns of `double` type became Matlab arrays of `double` type, whereas date columns were retrieved as cell arrays.

```
class(oprice)
```

```
ans =
```

```
double
```

```
class(crdate)
```

```
ans =
```

```
cell
```

Since `blsimpv` needs time to expiration, expressed as fraction of the year, as an input, we compute it with

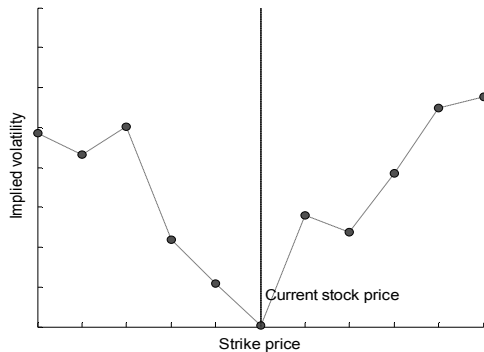
```
fmt = 'yyyy-m-dd';
```

```
exdate_num = datenum(char(exdate),fmt);
crdate_num = datenum(char(crdate),fmt);
time = (exdate_num - crdate_num)/365;
```

and invoke `blsimpv` with

```
impvol = blsimpv(sprice,strike,rate,
                time,oprice);
```

Drawing a plot of the implied volatility against the strike price, we find σ_{IMP} values to be lowest for strike prices close to the current stock price, and increase with the distance between the two. The pattern does not seem to be random, suggesting that sample option prices do *not* conform to Black-Scholes.



Implied-volatility patterns could be further explored with SAS. With this goal in mind, we take the original data back, accompanied by computed `impvol`.

Writing data from Matlab to MySQL takes two steps: creating an empty table with function `tbadd`, and filling it with `tbwrite`.

Inputs to `tbadd` include the table's name, and names and (MySQL) types of its columns, each packed in a cell vector. The two vectors can be obtained by modifying the output of a previous `tbattr` call; in this case, we can 'recycle' the 'table definition' of example as follows.

```
names(end+1) = {'impvol'}
types(end+1) = {'double'}
```

```
names =
```

```
'OPRICE'
'SPRICE'
'STRIKE'
'RATE'
'CRDATE'
'EXDATE'
'impvol'
```

```
types =
```

```
'double'
'double'
'double'
```

```
'double'
'date'
'date'
'double'
```

```
tbadd('example2',names,types)
```

A list of table columns must be provided to `tbwrite` as well, along with a list of source Matlab arrays, numeric or cell vectors of common length.

```
tbwrite('example2',names,lower(names))
```

Once in MySQL, the data are passed to SAS with

```
proc copy
  in = dbtest
  out = sas;
  select example2;
run;
```

Variable	Type	Format	Label
CRDATE	Num	DATE9.	CRDATE
EXDATE	Num	DATE9.	EXDATE
IMPVOL	Num		IMPVOL
OPRICE	Num		OPRICE
RATE	Num		RATE
SPRICE	Num		SPRICE
STRIKE	Num		STRIKE

We see that `example2.sas7bdat` has the right column names and types, but notice that the format of `crdate` and `exdate` has changed from `YYMMDDN8` to `DATE9`. Variable labels in `example2` are also clearly not the same as in `example`. Appendix II provides two SAS macros addressing these 'wrinkles'.

References

Hull, John C. (2005), *Options, Futures and Other Derivatives* (6th ed.). Upper Saddle River, NJ: Prentice-Hall Inc.

SAS Institute Inc. (2004), *SAS/ACCESS 9.1.2, Supplement for MySQL (SAS/ACCESS for Relational Databases)*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), *SAS/ACCESS 9.1, Supplement for ODBC (SAS/ACCESS for Relational Databases)*. Cary, NC: SAS Institute Inc.

Acknowledgements

I am grateful to Michael Boldin of Wharton Research Data Services (WRDS) at the University of Pennsylvania for suggesting the core idea of this paper and for valuable feedback. Expert help from Damu Zhang and Mark Keintz is gratefully acknowledged. Finally, I thank Chris Shull for giving me the opportunity to work with the WRDS team.

Appendix I. Download links

MySQL Server 5.0

<http://dev.mysql.com/downloads/mysql/5.0.html>
(see 'Windows Essentials (x86)')

MySQL Connector/ODBC 3.51

<http://www.mysql.com/products/connector/odbc>
(see 'Windows Downloads, Driver Installer (MSI)')

mym

http://sourceforge.net/project/showfiles.php?group_id=200091

mym utilities

<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=11913&objectType=FILE>
(see 'Download now:')

Appendix II. Recovering SAS labels and formats

Column labels and formats are not supported by MySQL, and when a SAS dataset is placed into a database, its labels and formats are lost. It is a nuisance if we intend to get the data back to SAS later, or would like to use the labels in the Matlab session. The SAS macros below offer a remedy.

getLabelsAndFormats extracts a dataset's labels and formats, and saves them to another dataset. By directing the macro's output to a MySQL table, one places labels within Matlab's reach. Once the data (all or some of the original columns) get back to SAS, labels and formats are re-applied by setLabelsAndFormats.

```
/* Save variable labels and formats
   from dataset DATA to dataset INFO */
%macro getLabelsAndFormats2(data,info);
  %let dst = %scan(&data,-1,'. ');
  %let lib = %scan(work.&data,-2,'. ');
  proc sql;
    create table &info as
    select name, label, format
    from dictionary.columns
    where libname = upcase("&lib")
       and memname = upcase("&dst")
       and memtype = "DATA";
  quit;
%mend;
```

```
/* Apply variable labels and formats,
   saved by macro getLabelsAndormats
   in dataset INFO, to dataset DATA */
%macro setLabelsAndFormats(data,info);
  proc sql noprint;
    select count(*) into :n from &info;
    select cat('label ',strip(name),
              ' = "',strip(label),
              '"'; format ',strip(name),
```

```
              ' ',strip(format),';')
    into :l1 - %sysfunc(compress(:l&n))
    from &info;
  quit;
  data &data;
    set &data;
    %do i = 1 %to &n; &&l&i %end;
    run;
%mend;
```

Contact information

Dimitri Shvorob
Department of Economics
Vanderbilt University
Nashville, TN 37235

phone: 615-497-4968

e-mail: dimitri.shvorob@vanderbilt.edu



SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.