

SBW MATLAB User's Guide

Cameron Wellock
cwellock@kji.edu

Keck Graduate Institute

February 18, 2004

1. Introduction.....	1
1.1. Requirements	1
1.2. Installing on Windows systems	1
2. About SBW.....	2
2.1. The SBW Broker and Making Calls	2
2.2. Modules, Services, and Methods	3
2.3. Data Types in SBW	4
3. Accessing SBW through MATLAB.....	6
3.1. Connecting to SBW	6
3.2. Finding out what's available through SBW.....	6
3.3. Calling SBW services from inside MATLAB.....	7
4. Making SBW Modules from MATLAB Scripts.....	11
4.1. Adding SBW Directives to Your Scripts.....	11
4.2. Compiling your Module.....	14
4.3. Running using the Chaperone.....	14
4.4. Troubleshooting.....	15
5. Appendix A – New MATLAB Commands	16
6. Appendix B –SBW Directives for .m Files	17
7. Appendix C – Code for Calling an Existing Service	18
8. Appendix D – Code for Creating a New Service.....	19

1. Introduction

One of the most popular tools for scientific computing today is MATLAB. While C, FORTRAN, and Perl have their advantages, few tools can turn high-level mathematical thought into high-speed computation as well as MATLAB. MATLAB has a few weaknesses however: chief among these is the difficulty in integrating MATLAB programs with other software. This is particularly chafing in a field like systems biology, where the ability to write interchangeable “software components”—simulators, optimizers, and the like—is sorely needed.

The tools presented in this guide help to solve this problem. SBW is a robust and well-established framework for building systems biology software components; now MATLAB users can use and create these components themselves. Existing SBW modules can be accessed, and new ones can be created entirely inside of MATLAB. Better yet, modules created in MATLAB can be distributed freely, even to users who do not have MATLAB themselves.

1.1. Requirements

On Windows systems, you need *at minimum* a working SBW installation (see <http://sbw.sourceforge.net> to get a recent copy). If you do not have MATLAB, you will be able to run other people’s SBW modules, but not to create new ones. To use SBW inside of your MATLAB scripts, you will need MATLAB 6.5 or better. If you want to create SBW modules using MATLAB, you’ll also need the MATLAB compiler.

1.2. Installing on Windows systems

To install on a Windows system, you must download and run the installer (sbwmatin.exe). The installer will work whether or not you have MATLAB installed. If MATLAB is not installed, it will install the necessary runtime files only.

2. About SBW

The Systems Biology Workbench, or SBW, is a messaging system programs can use to exchange data and to accomplish tasks. SBW-enabled programs provide “services” to other SBW client applications; a SBML parser program for example might provide services to extract parts of SBML documents. Another program such as a simulator could use these services to load a model, without having to implement its own parser.

SBW is designed to be simple to use, portable, and fast. SBW is supported by a number of programming languages and tools, including: Java, C/C++, Perl, Python, Delphi, and now MATLAB. It works on Windows, Linux, and FreeBSD.

A number of SBW-enabled modules have been written, which provide services such as: interactive model editing, simulation, optimization, and SBML reading and writing. Through SBW, your applications can make use of these abilities quickly and easily.

2.1. The SBW Broker and Making Calls

At the heart of SBW is the Broker. The Broker is a program that runs in the background, out of sight and invisible to users. The Broker has two responsibilities: helping clients find available services, and forwarding call requests (and their results) from one program to another. Nothing in SBW happens without the assistance of the Broker. A good analogy is to think of SBW as a telephone service for programs; the Broker is the old-fashioned switchboard operator.

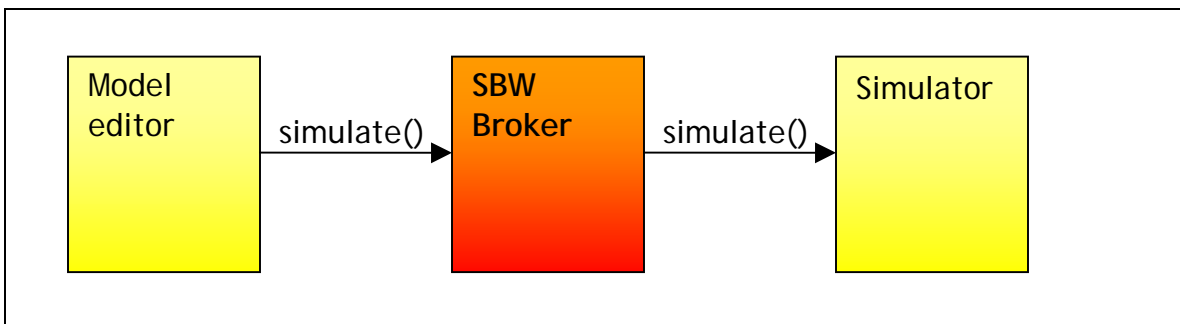


Figure 1. Example of a call through the Broker.

In the above example, a client application (a model editor of some kind) wishes to use a simulator. The simulator has a method called “simulate” which the model editor calls. The call passes first to the Broker, who in turn forwards it on to the simulator. All modules communicate through the Broker.

2.2. Modules, Services, and Methods

When a program wishes to expose its functionality to SBW clients, it must organize this functionality into a coherent framework. SBW requires that programs present themselves in a module-service-method hierarchy.

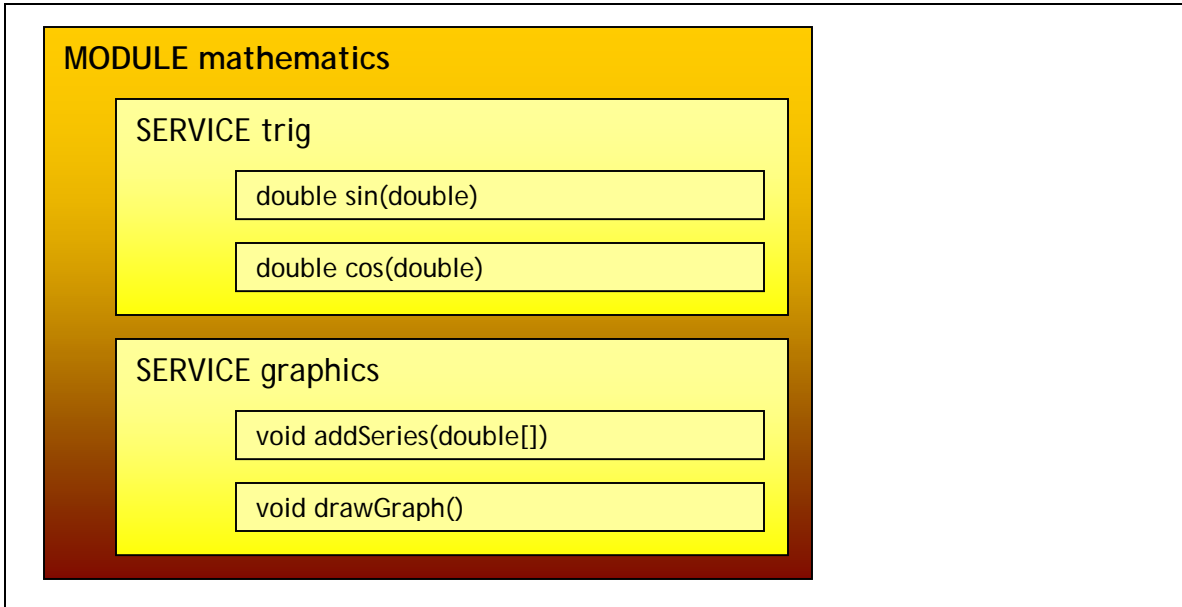


Figure 2. Module-Service-Method structure of a simple SBW module.

In the above example, we have a module named “mathematics”, which contains two services: “trig” and “graphics”. Each of these services has two methods that a client could call.

2.2.1. Modules

A module is a running instance of a program. Modules can provide zero or more services. Some modules are single-instance, but some can have multiple running copies, typically one for each client.

2.2.2. Services

A service is a logical grouping of related functions. For example, a module might have a “trig” service which contained functions for sin, cos, and tan. A service must contain one or more methods—no empty services are allowed. Services have optional category specifiers; you can define your own service categories as you wish. Typically, services in a category would all have the same methods. This way, clients can identify an available service by category and be assured it will have the methods they need, even if the implementations differ. For example, we could implement a “simulator” category and have two services implementing it, one using stochastic simulation and the other using deterministic methods.

2.2.3. Methods

A method is the lowest level in the SBW interface hierarchy. Methods are specific functions that can be called by clients. A method may take zero or more parameters, and may return zero or one results. SBW methods are modeled after traditional imperative programming constructs, and are strongly-typed functions or procedures (in computer science parlance). The name, input types, and return type of a function define a method's *signature*. For example, here is the signature for a hypothetical sin function:

double sin(double)

Signatures become important when accessing SBW methods using the `sbwgetmethod` function (see §3.3.3).

SBW differs significantly from MATLAB in how it handles function arguments. MATLAB makes no effort to enforce data types: a vector, a matrix, and a cell array are all valid arguments to any function parameter. Furthermore, MATLAB functions may return multiple results. These differences can be dealt with, however you should be aware of them as they may affect you in peculiar ways (see §3.3.6).

2.3. Data Types in SBW

SBW Supports a number of data types; these should largely be familiar to people familiar with C, Pascal, or other traditional programming languages. If MATLAB is the extent of your programming experience, however, these may seem a bit unusual.

2.3.1. Scalar Types

SBW Supports several distinct scalar data types. Scalar data types hold only a single value. SBW includes the following scalars:

SBW Name	Description
int	Integer values—whole numbers only. Values must be in the range of -2 147 483 648 to 2 147 483 647.
double	Floating-point value, including decimals. Generally accurate to ten significant digits, although caution is always advised.
boolean	Simple true/false value.
byte	Small integer value. Values must be in the range of -128 to 127. Can be more efficient than an int as it uses only ¼ the memory.
string	A text message, i.e. “this is a string.”

Table 1. SBW scalar data types.

2.3.2. Composite Types

In addition to the scalars, SBW also has support for two composite data types: arrays and lists.

Arrays are one-dimensional vectors or two-dimensional matrices of some scalar type. SBW denotes an array with a “[]” symbol. The type “int[]” represents a one-dimensional array of ints; the type “int[][]” represents a two-dimensional array of ints.

Lists are arbitrary sequences of other values. A list can have any size, and can contain items of any type or even many different types. Lists can contain scalars, arrays, and even other lists. A list is denoted in SBW with a “{ }” symbol.

Some lists have parameters, which indicate that the list takes a specific set of values. For example, a list denoted “{int, double, int[] }” would have to contain an integer, a double, and a single-dimensional array of integers, in that order. An error could potentially arise if it did not.

SBW methods can take any number of parameters, and each must have a type. Methods can also return a single value, although more could be “returned” if the return type is a list or array. Methods can also elect not to return any value, in which case their return type is the special “void” type.

3. Accessing SBW through MATLAB

The following paragraphs outline how to connect and use SBW services from inside of MATLAB. These instructions should work from the MATLAB console or from within a .m file. For a more concrete example of using a SBW service, see Appendix C – Code for Calling an Existing Service.

3.1. Connecting to SBW

Before you can make use of SBW, you must “connect” to it. In technical terms, this means that your script must first contact the SBW Broker (See §2.1). To open the connection, use the `sbwconnect` command:

```
>> sbwconnect  
  
ans =  
  
    1
```

`sbwconnect` will return a 1 if it succeeds, and 0 otherwise.

3.2. Finding out what's available through SBW

3.2.1. Finding Modules

To find out what modules are available to you, use the `sbwgetmodulelist` command:

```
>> sbwgetmodulelist  
  
ans =  
  
    'DataGenerator'  
    'twocalls'  
    'edu.caltech.NOM'  
    'edu.caltech.gibson'  
    'edu.caltech.NOMClipboard'  
    'edu.caltech.MatlabTranslator'  
    'JDesigner'  
    'edu.caltech.plot'  
    'edu.caltech.gibson.gui'  
    'Jarnac'  
    'fftModule'  
    'sbwoptimize_SAsimplex'  
    'optController'  
    'sbwoptimize_simplex'
```

```
'BROKER'  
'anonymous-1'
```

The function returns a cell array of module names. You can use these names with other functions, for example `sbwgetservicelist` or `sbwgetmoduleinstance`.

3.2.2. Finding Services

There are two ways to find services: by module, or by category. To find services by module, use the `sbwgetservicelist` command:

```
>> sbwgetservicelist('edu.caltech.MatlabTranslator')  
  
ans =  
  
    'MatlabTranslator'  
    'SaveModelAsMatlabSimulinkFunction'  
    'SaveModelAsMatlabODEFunction'
```

To find services by category, use the `sbwfindservices` command:

```
>> sbwfindservices('BioSPICE/Analyzer')  
  
ans =  
  
    'JDesigner'      'Jarnac'  
    'biospiceAnalyze'  'biospiceAnalyzeSim'  
    'Jarnac'         'fftModule'  
    'biospiceAnalyzeSS' 'biospiceAnalyze'
```

3.3. Calling SBW services from inside MATLAB

In order to call the methods of an SBW service, you must do three things: get a module handle, possibly starting the module in the process; get a service handle; and finally get one or more method handles for the methods you intend to use.

3.3.1. Getting a Module Handle

A module handle is a unique identifier for a running module. Because more than one copy of a module can exist, the name of the module alone is not sufficient to ensure uniqueness. Instead a number is assigned to each running module; you will need this number (the “handle”) in order to call methods on the module.

To get a module handle, use the `sbwgetmoduleinstance` function:

```
jarnac = sbwgetmoduleinstance('Jarnac')
```

```
jarnac =
```

```
3
```

`sbwgetmoduleinstance` takes the name of the module you want as its only argument. A return value of -1 indicates failure; any other return value is a valid handle.

3.3.2. Getting a Service Handle

Just as for modules, you must get a handle to a service before you can use it. To do so, use the `sbwgetservice` function:

```
>> sim = sbwgetservice(jarnac, 'sim')
```

```
sim =
```

```
2
```

`sbwgetservice` requires two arguments: the module handle for the module the service is on, and the name of the service. A return value of -1 indicates failure; any other return value is a valid handle.

3.3.3. Getting a Method Handle

Finally, one must get a method handle. To get a method handle, you will need the full signature of the method, not just its name (see §2.2.3). If you don't know the signature, try looking it up using the SBW Browser or the SBW Inspector. Use the `sbwgetmethod` function to get the method handle:

```
>> simulate = sbwgetmethod(jarnac, sim, 'double[][] simulate()')
```

```
simulate =
```

```
262218784
```

```
>> snp = sbwgetmethod(jarnac, sim, 'void setNumPoints(int)')
```

```
snp =
```

```
262219360
```

`sbwgetmethod` requires three arguments: the module handle, the service handle, and the full method signature. It returns an arbitrary integer (the handle) if successful, or -1 otherwise.

3.3.4. Making the Call

It's finally time to make a call to SBW. To do this, we need to make use of the `sbwcall` function. `sbwcall` is used as follows:

```
>> sbwcall(snp, toint(100))

ans =

     []

>> r = sbwcall(simulate)

r =

     0         0.1000         0.1000         0.1000
    0.1010        0.1363        0.3758        0.0930
    0.2020        0.2536        0.8163        0.0949
    ...
```

In the above example we use the method handles to `setNumPoints` and `simulate` for our simulator. `sbwcall` takes a variable number of arguments: the first is always the method handle; the rest are the arguments to the function (i.e. 100, the number of time points to simulate). `sbwcall` returns whatever the function would return, or an empty list if the function's return type is `void`.

3.3.5. Converting Integers to Floating-Point Values

A common problem when dealing with SBW is that many SBW functions take or return integer values. MATLAB supports integers, but generally does not use them: any number you enter in MATLAB is assumed to be a floating-point value, even if it has no decimal. To help smooth over this difference, two additional commands are provided: **`toint`** and **`fromint`**. `toint` takes one or more floating-point numbers and converts them to integers; `fromint` takes one or more integers and converts them back to floating-points.

3.3.6. How MATLAB Translates Parameters

MATLAB and SBW have very different ideas about what constitutes "data." While SBW supports all the data types outlined in Table 1, as well as arrays and lists. MATLAB works primarily in matrices, but also uses character and cell arrays. The following table illustrates how different data types are converted from SBW types to native MATLAB data types.

SBW Type	MATLAB Equivalent
int	1x1 int32 matrix
double	1x1 double matrix
boolean	NOT SUPPORTED
byte	NOT SUPPORTED
string	1xN char array
[] (1-dimensional array)	1xN matrix
[][] (2-dimensional array)	NxM matrix
{ } (list)	1xN cell array

Table 2. SBW to MATLAB type conversions.

The following table illustrates how different data types are converted from native MATLAB data types to SBW types:

MATLAB Type	SBW Equivalent
double 1x1 matrix	double
double 1xN or Nx1 matrix	double[] (row- or column-orientation lost)
double NxM matrix	double[][]
int32 1x1 matrix	int
int32 1xN or Nx1 matrix	int[] (row- or column-orientation lost)
int32 NxM matrix	int[][]
char 1xN matrix	string
char NxM matrix	string[]
cell 1xN matrix	{ }
cell NxM matrix	{ } (cell matrix will be unraveled)

Table 3. MATLAB to SBW type conversions.

There are some special considerations to be aware of: first off, not all SBW data types are currently supported; avoid methods that use them. A more complicated situation arises when MATLAB functions return SBW lists: if the lists have no extra type data (i.e. “{ }”) then the converter system must make its best guess as to what the contents of the list are supposed to be. If the cell array contains a 1x1 matrix, the converter will assume this is supposed to be a scalar—there is no simple way to tell the difference between a scalar and a 1x1 matrix in MATLAB.

4. Making SBW Modules from MATLAB Scripts

We have shown how to access SBW modules from inside your MATLAB scripts. Just as excitingly, we can also write scripts in MATLAB and then export them as SBW modules. Modules created in this way can be distributed easily, as they do not require MATLAB to run. You can compile as many MATLAB functions as you want into a single SBW module.

To create a SBW module from an existing MATLAB script, we need to follow a few simple steps:

- 1) Add some extra comments to the .m file. These tell SBW how to present your script, what data types are for the arguments, etc.
- 2) Compile the script into a DLL.
- 3) Run the Chaperone program with your DLL.

The following sections will explain in detail how this can be done. For a complete example of how to create a module in MATLAB, see Appendix D – Code for Creating a New Service.

4.1. Adding SBW Directives to Your Scripts

4.1.1. The Comments Section

```
%SBW module mathematics help="a module for math"

%SBW service trig help="a trig module"
%SBW method sin sig="double sig(double)" service="trig" help="sine"
%SBW method cos sig="double cos(double)" service="trig" help="cosine"

%SBW service graphics help="to do basic plotting"
%SBW method addSeries sig="void addSeries(double[])" service="graphics"
%SBW method plotData sig="void plotData()" service="graphics"
```

Looking at the above lines, you might notice that each line is a proper MATLAB comment: these lines have no effect on MATLAB at all. The only time they're used is when the **sbwbuild** command is run: it reads these lines and uses them to construct your SBW module. Every SBW module you build in MATLAB must have comments like these.

In this case, the comments specify a module named "mathematics" which would implement the service shown in Figure 2. Although we recommend you put them near the top of your .m file, they can appear anywhere--at the bottom of the file, spread throughout the file, or even split up among multiple files.

Each SBW comment line must start with a "%SBW" symbol, followed by one of the following: "module", "service", or "method". After this should come the name of the module, service, or method you are specifying. The rest of the comment line should consist of one or more name-value pairs of the form "name=value", i.e. "service=trig". Double quotes can be put around the value if necessary, but are not required. You can have multiple lines to define a single item, for example

```
%SBW module myModule desc="a module" management="unique"
```

Is equivalent to

```
%SBW module myModule desc="a module"  
%SBW module myModule management="unique"
```

4.1.2. What's Required

To create a module, you will need *at minimum*:

- a line specifying the module name (a %SBW module line)
- a line defining a service (a %SBW service line)
- a line defining a method (a %SBW method line)

You can have more than one method in a service, and more than one service in a module. Additional methods and services simply require extra lines.

4.1.3. Specifying a Module

To specify a module, use the "%SBW module" line. It takes only a few options, outlined below:

- desc: a brief description of the module.
- management: can be either "unique" or "self-managed". Unique modules start up a new running instance every time a client calls `getmoduleinstance`. Self-managed modules start only once and are shared between all clients. The default value is "unique".
- help: a short text string indicating how to use the module.

4.1.4. Specifying a Service

To specify a service, use the "%SBW service" line. It takes the following options:

- desc: a brief description of the service.
- category: an optional category to put the service into.
- help: a short text string indicating how to use the service.

4.1.5. Specifying a Method

To specify a method, use the “%SBW method” line. One important point to note is that the name of the method specified should match *exactly* the name of the .m file which contains the function implementation. For example, if the comment line was

```
%SBW method createIdentityMatrix sig="double[][] createIdentityMatrix(int)"
```

Then the implementation of createIdentityMatrix should be in createIdentityMatrix.m.

A “%SBW method” line takes the following options:

- sig: defines a method signature for the function; this tells SBW what the argument types are and how to convert them. You **must** specify a signature.
- service: specifies what service the method should appear in. If you do not specify a service, the method will be accessible only to the other functions inside of your module.
- help: a short help string indicating how to use the method.

To create a method signature, you will have to decide what SBW data types your function will take and return. See Table 3 to decide what types your arguments should be. You should have one SBW type for each argument your function takes. Remember that SBW methods can only have a single return value; if you want to return more you should return them in a list (“{ }”) and be sure to modify your function to return a cell array of values.

After you’ve decided what the arguments should be, you can create the full signature. Start with the return type (remember that if your function does not return anything, the return type is “void”) and then add the function name. In parentheses, add all of the argument types, separated by commas. That’s it!

Let’s look at an example to help clarify. Suppose our MATLAB function looked like this:

```
function c = add2(a, b)
    c = a + b;
```

Sometimes it’s easy to decide what data types your arguments should be: if you know you’re expecting a matrix for example. In this case, the function could add scalars, vectors, or matrices. Let’s assume we really want it to be able to add matrices. We look in Table 3 and see that a MATLAB matrix is equivalent to a double[][] type in SBW. Both arguments, as well as the return value, will need to be double[][] then. With this, we can construct the SBW signature:

```
double[][] add2(double[][], double[][])
```

and finally add the %SBW method line to our .m file:

```
function c = add2(a, b)
    c = a + b
%SBW method add2 sig="double[][] add2(double[][], double[][])"
%SBW method add2 service="math"
```

4.2. Compiling your Module

You compile your module into a DLL using the **sbwbuild** command. The usage is simple:

```
sbwbuild [mfile1 mfile2 ...] [mexfile1 mexfile2 ...]
```

Use the **sbwbuild** command from the MATLAB prompt. Specify as options all the .m files you want to include in your module, even those that are only used internally. Specify also any extra MATLAB commands you use. **sbwbuild** will create a DLL containing your functions; the name of the DLL will be the same as the name of the first .m file specified. For example, if your module had re-implemented `sin` and `cos`, and `cos` used the `matlab tan` function, your **sbwbuild** command would look like

```
sbwbuild sin.m cos.m tan
```

And would create a DLL named `sin.dll`.

4.3. Running using the Chaperone

To use your module, you need the help of the Chaperone. The Chaperone is a small program that loads your module and connects it to SBW for you. There are two versions of the Chaperone: a Windows version (`sbwmatch.exe`), and a console version (`sbwmatcc.exe`), which is handy when you're trying to find problems—it can print out messages you've embedded in your .m file using the “display” command.

Before clients can use your module, you'll need to make sure your module is *registered*. You'll need to register your module on every computer it will be used on. To register a module, use the following command:

```
sbwmatch <dllname> -sbwregister
```

If we were trying to register the fictional `sin.dll` we build in §0, we would use the following command:

```
sbwmatch sin.dll -sbwregister
```

Note that you can use `sbwmatch` or `sbwmatcc` interchangeably.

After your module is registered, you can run it. When running, its services will be available to any clients that want to make use of it. Run the module with the following command:

```
sbwmatch <dllname> -sbwmodule
```

4.4. Troubleshooting

These hints provide some general advice on solving problems with the MATLAB-SBW interface.

4.4.1. Trouble accessing SBW from inside of MATLAB

None of the new commands work. This is probably a path-related problem. Check your MATLAB search path (using the “path” command) and make sure that the SBW bridge directory is included; this directory is typically “C:\Program Files\SBWMATCH” but could be something else.

I can't connect to a particular module. Chances are the module isn't registered properly. Re-register the module and try again.

I get “Function ‘xxx’ is not defined for values of class ‘int32’” errors. You are trying to use an integer value with a function that takes only floating-point values. The integer was probably the result of an SBW function. Use the `fromint()` function to convert the value to a floating-point.

4.4.2. Trouble compiling SBW modules from MATLAB

I get some kind of parsing error when using `sbwbuild()`. Check that your SBW comments are all syntactically correct, and complete (you've got module, service, and method lines).

I get linking errors mentioning some other function. Chances are the other function is a MEX-file that your program depends on. You need to add the name of this function to the `sbwbuild()` command line as well. You should not need to modify your source code however.

4.4.3. Trouble using SBW modules that were built in MATLAB

I get “DLL not found” errors. Make sure that the SBW-MATLAB interface installation directory is on the current search path.

5. Appendix A – New MATLAB Commands

Name	Purpose
<code>fromint</code>	Convert integers to doubles.
<code>sbwcall</code>	Call a SBW method. Must have a method handle from <code>sbwgetmethod</code> first.
<code>sbwconnect</code>	Connect to the SBW broker.
<code>sbwconnectonhost</code>	Connect to a SBW broker on a remote machine.
<code>sbwdisconnect</code>	Disconnect from the SBW broker. Call this when you are done with SBW to free used resources.
<code>sbwfindservices</code>	Find all available services in a specific category.
<code>sbwgetmethod</code>	Get a method handle. See §3.3.3 for details.
<code>sbwgetmethodlist</code>	Get a list of all the methods on a service.
<code>sbwgetmoduleinstance</code>	Get a module instance. See §3.3.1 for details.
<code>sbwgetmodulelist</code>	Get a list of all available modules.
<code>sbwgetservice</code>	Get a service handle. See §3.3.2 for details.
<code>sbwgetservicelist</code>	Get a list of available services on a module.
<code>sbwlink</code>	Link the local broker to a remote broker.
<code>sbwmoduleshutdown</code>	Shut down a running module. Use this after you have finished with a module, or the module will run indefinitely.
<code>toint</code>	Convert double values to integers.

Table 4. New MATLAB commands.

All of these commands have online documentation available; the online documentation is considered the definitive source of information. To get help for a command from the MATLAB command prompt, type

```
help <command>
```

For example,

```
>> help sbwfindservices
```

```
SBWFINDSERVICES Finds available SBW services according to category.
```

```
S = sbwfindservices(category)
```

`sbwfindservices` will locate all services in the specified category. The function returns a Mx2 cell matrix, where M is the number of services found. The first column of each row will be a string identifying the module name; the second column will be the service name itself.

Cameron Wellock (cwellock@kgi.edu)

November 19, 2003

6. Appendix B –SBW Directives for .m Files

Parameter	Value
<i>For modules (%SBW module)</i>	
desc	A text description of the module.
management	“unique” or “self-managed” (see §4.1.3).
help	Any additional text about using the module.
<i>For services (%SBW service)</i>	
desc	A text description of the service.
category	A short text name for the service category, if there is one (see §4.1.4).
help	Any additional text about using the service.
<i>For methods (%SBW method)</i>	
sig	The method signature.
service	The service to attach the method to, or none to keep the method private.
help	Any additional text about using the method.

Table 5. SBW directives for .m files.

7. Appendix C – Code for Calling an Existing Service

The following script gets the current model from the running instance of JDesigner and uses Jarnac to run a simulation, plotting the results. It requires both JDesigner and Jarnac to run.

```
% connect to SBW
sbwconnect;

% get JDesigner method to fetch SBML
jdesigner = sbwgetmoduleinstance('JDesigner');
model = sbwgetservice(jdesigner, 'model');
getSBML = sbwgetmethod(jdesigner, model, 'string getSBML()');

% get Jarnac simulation methods
jarnac = sbwgetmoduleinstance('Jarnac');
sim = sbwgetservice(jarnac, 'sim');
loadSBML = sbwgetmethod(jarnac, sim, 'void loadSBML(string)');
setTimeStart = sbwgetmethod(jarnac, sim, 'void setTimeStart(double)');
setTimeEnd = sbwgetmethod(jarnac, sim, 'void setTimeEnd(double)');
setNumPoints = sbwgetmethod(jarnac, sim, 'void setNumPoints(int)');
simulate = sbwgetmethod(jarnac, sim, 'double[][] simulate()');

% get the SBML file from JDesigner
sbml = sbwcall(getSBML);

% set simulation parameters
%sbwcall(setTimeStart, 0);
sbwcall(setTimeEnd, 1);
sbwcall(setNumPoints, toInt(50));

% run the simulation
r = sbwcall(simulate);

% plot the results
plot(r);

% disconnect from sbw
sbwmoduleshutdown(jarnac);
sbwmoduleshutdown(jdesigner);
sbwdisconnect;
```

8. Appendix D – Code for Creating a New Service

The following example creates a module with a single service and a single method. This method can be used to add two vectors together.

```
function c = add2(a, b)
    c = a + b;

%SBW module math desc="basic mathematics"
%SBW service add desc="addition operations"
%SBW method add2 service=add sig="double[] add2(double[], double[])"
```

The above script, stored in a file named add2.m, would be compiled with the following command inside MATLAB:

```
>> sbwbuild add2
```

And would create a DLL named add2.dll. To register the module with SBW, the following command would be used from within the Windows command prompt:

```
sbwmatch add2.dll -sbwregister
```

And the following command would be used to start the module for use by clients:

```
sbwmatch add2.dll -sbwmodule
```