

BACC Developer's Manual for Windows

Bill McCausland and John J. Stevens

August 2, 2001

1 Intended Audience

The intended audience for the PC Developer version of BACC is those who

- have used a precompiled version of BACC
- need to add a new model to BACC
- have programming experience in C

2 Objective

The objective of this manual is to provide an introduction to developing with BACC on Windows PC's. The first few sections are descriptive, providing details about the BACC source code files and the Microsoft Visual C++ workspace used by BACC. The latter sections are instructive, providing details about developing new models and routines for use within BACC.

Before attempting to develop using BACC, it is recommended that you make certain that your needed model specification and/or computational routines are not already available within the BACC system. If BACC does contain the needed model and/or computational routines, then it is recommended that you download a pre-compiled version of BACC for Console (MS-DOS), Matlab, Gauss, or Splus, as well as the corresponding user's manual.

Please send any comments regarding this manual, including information that would be helpful for other developers, via electronic mail to `bacc@econ.umn.edu`.

3 Getting Started

3.1 Requirements

- PC running Microsoft Visual C++ Version 6.0
- BACC zip file, `baccWinDevelop.zip` available at <http://www.econ.umn.edu/~bacc>

- Solid knowledge of C programming techniques including the use of structures, pointers, enumerated types, etc....

3.2 Installing BACC

Unzip `baccWinDevelop.zip` to a directory on your hard drive. The main directory is called `bacc`. The main subdirectories, discussed in the next section, contain the BACC source code and the Microsoft Visual Studio workspace files needed to compile the application independent BACC static libraries, `baccClosedLibrary.lib` and `baccLibrary.lib` (used by all application interfaces), the (DOS) Console executable, the dynamically linked library (DLL) for Gauss, the DLL for use within Matlab 5 Release 11 or higher, and the DLL for use with Splus.

4 Detailed BACC Directory Structure

This section provides a detailed accounting of the BACC directory structure. If you first want to compile BACC without attempting any modifications or additions, then you may want to skip to Section 5 and return to this section later.

The three sub-directories of `bacc` are

- `Build`
- `Compiled`
- `Source`

4.1 Subdirectories of Build

This directory contains the subdirectories `Linux`, `Unix`, and `Windows`, but we will only focus on the latter. `bacc\Build\Windows` contains several Microsoft Visual C++ project files beginning with `PCbacc` as well as a number of subdirectories. These are:

- `baccClosedLibrary`
This folder contains the Microsoft Visual C++ project settings file for the static BACC closed library, `baccClosedLibrary.lib`. “Closed” refers to the fact that these libraries were not programmed by the BACC team.
- `baccLibrary`
This folder contains the Microsoft Visual C++ project settings file for the static BACC library, `baccLibrary.lib`.
- `BUILD BACC *`
These folders contain the Microsoft Visual C++ project settings files for building the application `*`, where `*` indicates either `Console`, `Gauss`, `Matlab`, or `Splus`.

- **Debug, Matlab, and Release**
After BACC has been compiled, one or more of these subdirectories will exist depending upon which versions were compiled. These folders contain intermediate files (mostly compiled object code).

4.2 Subdirectories of Compiled

This directory contains the subdirectories **Linux**, **Unix**, and **Windows**, but we will only focus on the latter. There is a separate subdirectory of `bacc\Compiled\Windows` corresponding to each application. The application folders contain the application-specific files necessary to communicate with the compiled BACC code, a test directory, and (after compiling) the compiled executable or DLL

- **Console**
This folder also contains a batch file called `copyCallBacc.bat` that copies the executable `callBACC.exe` once for each BACC command. A sub-directory, `test`, contains test files. After compiling the Console version there will also be a file called `consoleBACC.exe`.
- **Gauss**
This folder also contains the Gauss `.src` files and the Gauss `.lcg` file. A sub-directory, `test`, contains test files. After compiling the Gauss version there will also be a file called `libBACC.dll`.
- **Matlab**
This folder contains the Matlab m-files for the BACC routines. A sub-directory, `test`, contains test files. After compiling the Matlab version there will also be a file called `bacc.dll`.
- **Splus**
This folder contains the script files Splus needs for the BACC routines. A sub-directory, `test`, contains test files. After compiling the Splus version there will also be a file called `baccSplus.dll`.

4.3 Subdirectories of Source

Model developers should only need to create files in the **Models** and **Matrx** sub-directories of `bacc\Source`. The **Lapack**, **Blas**, **Cdflib**, and **Ranlib** sub-directories provide useful routines for those wishing to expand the **Matrx** library. It is not recommended that any modifications be made outside of the **Models** and **Matrx** sub-directories. The following sub-directories are in `bacc\Source`:

- **Auxil**
This folder contains two files. The first, `allocate.c`, handles memory allocation and the second, `error.c`, handles BACC errors. Each of these files has its own header file located in `bacc\Source\Includes`.

- **Blas**
This directory, together with its two subdirectories, `libF77` and `libI77`, constitutes the BLAS (Basic Linear Algebra Subprograms). This library is distributed with BACC, but it is also available from the Netlib Repository.¹
- **Cdflib**
This folder contains routines for evaluating cumulative distribution functions as well as their inverses and parameters. It is also available from Netlib.
- **Console**
This folder contains the source code files needed to compile executables that run on the system console (i.e., at a DOS, Unix, or Linux command line prompt).
- **Core**
This folder contains application independent BACC files. The names of most of these files end in “Core”. These files are the computational heart of BACC. It is not recommended that any changes be made to these files. If you suspect that there is a bug in one of these files, please send a detailed email to BACC indicating why you suspect there is a bug and, if possible, where the bug is.
- **Gauss**
This folder contains the source files needed to compile BACC into a dynamically linked library that can be used by Gauss.
- **Includes**
This folder contains the major C header files. Details on specific routines are found in some of these. For example, `bacc_core.h` and `bacc_matrx.h` contain extensive comments. Some secondary header files are located within the various source file directories. When a non-system header file is included in a C file, Visual C++ searches the directory with the source file and then any paths that have been added to the preprocessor settings. The Microsoft Visual C workspace for BACC contains the path for the `Includes` sub-directory, therefore it is not necessary to explicitly add header files to the BACC workspace.
- **Lapack**
This folder contains LAPACK (Linear Algebra PACKage). This library is distributed with BACC, but it is also available from Netlib.
- **Matlab**
This folder contains the source files needed to compile BACC into a dynamically linked library that can be used by Matlab.

¹The Netlib Repository can be found on the web at <http://www.netlib.org>.

- **Matrx**
This folder contains numerous routines designed to operate on the BACC matrix structure named **Matrx** (Yes, there is no *i* in **Matrx**). Be sure to examine the header file `bacc\Includes\bacc_matrx.h` to learn about the **Matrx** structure. Model developers will work primarily in this directory and in `bacc\Source\Models`.
- **Models**
This folder contains the model specification files. To add a model specification to BACC you need to write a model specification file. Be sure to look at the header file `bacc\Includes\bacc_model.h`. Model developers will work primarily in this directory and in `bacc\Source\Matrx`.
- **Ranlib**
This folder contains files which have random number generating routines. This library is distributed with BACC, but it is also available from Netlib.
- **Splus**
This folder contains the source files needed to compile BACC into a dynamically linked library that can be used by Splus.

5 Compiling BACC

The following subsections describe how to compile the existing BACC files. Compiling BACC amounts to compiling the static BACC libraries (`baccClosedLibrary.lib` and `baccLibrary.lib`) which are application independent and then compiling an application specific interface for DOS, Matlab, Gauss, or Splus.

5.1 Loading the BACC Workspace

After unzipping `baccWinDevelop.zip`, run Microsoft Visual C++ and click on the **File** menu. Then click **Open Workspace**. Locate the file `bacc\PCbacc.dsw` and click **Open**.

5.2 Setting the Active Configuration

Some projects may be compiled according to various configurations. The Console BACC executable may be compiled using either the “Debug” or the “Release” configuration. The Debug configuration produces a larger executable and cannot be optimized for speed. The advantage of the Debug version is that it allows the use of the powerful Microsoft Visual C debugger, a topic that is beyond the scope of this manual. The Gauss DLL can also be compiled as a “Debug” or “Release” version, but it is generally recommended that only the “Release” version be used, leaving the debugging process to the Console version. Currently, the Matlab DLL may be compiled only as a “Matlab” version which contains no debugging information and is optimized for speed.

To set the active configuration, click on **Build** and then select **Set Active Configuration**. From the list of projects and configurations, select the project and configuration you wish to compile and then click on **OK**.

- For a DOS build this would mean selecting either
BUILD BACC CONSOLE - Win32 Release
or
BUILD BACC CONSOLE - Win32 Debug
- For a Gauss build this would mean selecting either
BUILD BACC GAUSS - Win32 Release
or
BUILD BACC GAUSS - Win32 Debug
- For a Matlab build this would mean selecting
BUILD BACC MATLAB - Win32 Matlab
- For an Splus build this would mean selecting either
BUILD BACC SPLUS - Win32 Release
or
BUILD BACC SPLUS - Win32 Debug

5.3 Compiling the (MS-DOS) Console Executable

To build the Console executable, from the **Project** menu, select **Set Active Project**. Click on **BUILD BACC CONSOLE**. Then from the **Build** menu, select **Build**. After compiling, the Console executable and the executables call it will be located in

```
bacc\Compiled\Windows\Console
```

5.4 Compiling the Gauss DLL

A single dynamically linked library contains all of the BACC routines for use in Gauss. From the **Project** menu, select **Set Active Project**. Click on **BUILD BACC GAUSS**. Then from the **Build** menu, select **Build**. This creates a dynamically linked library named `libBACC.dll` that is located in

```
bacc\Compiled\Windows\Gauss
```

5.5 Compiling the Matlab DLL

5.5.1 The first time

Before compiling the Matlab version of BACC for the first time, it is necessary to set an environment variable called **MATLABLOCATION** that provides the compiler the path containing the Matlab `Bin` subdirectory. In Windows 95/98 you do this by adding a **SET** command to the `autoexec.bat` file located in your root directory. For example, if the `Bin` subdirectory is in `C:\Matlab`, you would add the following line to `autoexec.bat`:

```
SET MATLABLOCATION=C:\Matlab
```

In Windows NT you add the environment variable by right clicking on **My Computer**, selecting the **Environment** tab, and then typing the word

```
MATLABLOCATION
```

in the **Variable** field and the Matlab path, i.e.,

```
C:\Matlab
```

in the **Value** field.

It is necessary to restart Windows in order for the new environment variable to take effect. This must be done prior to compiling the Matlab version of BACC.

5.5.2 Each time

To build the Matlab DLL, from the **Project** menu, select **Set Active Project**. Click on **BUILD BACC MATLAB**. Then from the **Build** menu, select **Build**. This will build the Matlab DLL and place it in

```
bacc\Compiled\Windows\Matlab
```

5.6 Compiling the Splus DLL

A single dynamically linked library contains all of the BACC routines for use in Splus. From the **Project** menu, select **Set Active Project**. Click on **BUILD BACC SPLUS**. Then from the **Build** menu, select **Build**. This creates a dynamically linked library named `baccSplus.dll` that is located in

```
bacc\Compiled\Windows\Splus
```

5.7 An Aside: Dependencies

All of the BACC projects depend upon the static libraries `baccLibrary.lib` and `baccClosedLibrary.lib`. These dependencies are made explicit in the workspace settings. This means that changes to a file in `baccLibrary`, such as a model specification or matrix routine file, will cause the library to be rebuilt automatically if any BACC project is built.

5.8 An Aside: Warning Messages

Depending upon the sensitivity at which you set the compiler warnings, during the first build of BACC you will see a number of warnings flashing across the display window (over 1100 warnings in fact). Most of these are LAPACK warnings and are not of any concern. If you get any errors rather than warnings, the build should fail. The default settings in BACC are not to show warnings for `baccClosedLibrary`.

6 After Compiling BACC

This section discusses what needs to be done (if anything) after compiling BACC for a particular application. It also discusses how to access the BACC commands from each application.

6.1 DOS Files

The compiled Console executables are located in

`bacc\Compiled\Windows\Console`

There are three options for calling these executables:

1. Copy the executables to the directory in which you are working and then call them using only the command name, i.e.,
`miLoadAscii mi mi.txt`
2. Leave the executables where they are and call them using the full path, i.e.,
`c:\bacc\Compiled\Windows\Console\miLoadAscii mi mi.txt.`
3. In Windows 95/98/NT you can add the path of the executables to the `autoexec.bat` file. For example, to append the path,
`c:\bacc\Compiled\Windows\Console`
to the existing path, add the line,
`path=%path%;c:\bacc\Compiled\Windows\Console`
After rebooting Windows the executables can then be called as in 1 above, regardless of what directory you are in.

6.2 Gauss Files

There are three copying steps for Gauss.

1. Copy the DLL, `libBACC.dll`, from
`bacc\Compiled\Windows\Gauss`
to the `dlib` folder for your Gauss software, typically `Gauss\dlib`.
2. Copy
`bacc\Compiled\Windows\Gauss\libBACC.lcg`
to the `lib` folder for your Gauss software, typically `Gauss\lib`. Note: If Gauss is not installed to `C:\Gauss`, then you will need to edit the absolute paths given in the `libBACC.lcg` file.
3. Copy the `*.src` files from
`bacc\Compiled\Windows\Gauss`
to the `src` folder for your Gauss software, typically `Gauss\src`.

Note: `libBACC.dll` may not be visible if your view settings in Windows Explorer are set to hide DLL files. This can be changed by going to the **View** menu and then to **Options** while in Windows Explorer.

6.3 Matlab Files

As discussed in Section 5.5, the compiled DLL for Matlab is located in

`bacc\Compiled\Windows\Matlab`

Also located in this folder are the `*.m` files required by BACC. To be make the BACC commands available within Matlab, this folder must be added to Matlab's search path. To add the path of your folder to Matlab's search path, select **Set Path** from Matlab's **File** menu. This will open a new window. Select **Add to Path** in the **Path** menu of this new window. Either type in the path of your folder or browse for it by clicking on the button with the ellipsis mark.

Note: The `*.dll` files may not be visible if your view settings in Windows Explorer are set to hide DLL files. If you wish, this can be changed by going to the **View** menu and then to **Options** while in Windows Explorer.

6.4 Splus Files

As discussed in Section 5.6, the compiled DLL for Splus is located in

`bacc\Compiled\Windows\Splus`

The script files needed by Splus to call the DLL are also be in this directory.

Note: The `*.dll` files may not be visible if your view settings in Windows Explorer are set to hide DLL files. If you wish, this can be changed by going to the **View** menu and then to **Options** while in Windows Explorer.

7 Model Development

7.1 What the Model Developer Does

The model developer defines in a highly structured manner, a statistical model describing unknown and known quantities, and a Monte Carlo Markov Chain algorithm for simulating the unknown quantities, conditional on the known quantities. We describe the model and algorithm together as a model specification.

7.1.1 The Elements of a Model Specification

The elements of a model specification are dimension parameters, known quantities, unknown quantities, auxiliary quantities, and Gibbs blocks. Dimension parameters are quantities that determine the dimensions of certain matrices. Examples include the number of observations in a data set, the number of individuals in a cross section, the number of equations in a system of equations, and the number of states in a Markov chain. Known quantities may be fixed prior parameters or data. To the model developer, there is no functional distinction between the two. Both need to be fully specified by the user to generate a model instance, and both remain constant during simulation. Unknown quantities may be model parameters, latent variables, or missing data. Similarly, there is no

functional distinction among these categories. All are simulated from their joint posterior distribution.

Auxiliary quantities are any other quantities used in calculations. They may be sufficient statistics that are calculated only once, or they may be constantly changing intermediate results such as the current value of regression residuals. Gibbs blocks define distributions to draw subset of parameters from. There may be a one-to-one correspondence between blocks and parameters, a single block to draw all parameters, or anything else.

7.1.2 Defining a Model Specification

The model developer provides the numbers of dimension parameters, known quantities, unknown quantities, and Gibbs blocks.

For each known quantity, she may provide a routine that establishes whether or not a specific value of the quantity is valid or not. A model specification may require, for example, that a particular quantity be positive definite, and this is how such a requirement is imposed. If no routine is provided, it implies that there are no restrictions on the value of the known quantity.

For each unknown quantity, she may provide routines to evaluate the log prior density at a given value of the quantity, and a routine to perform the transformation of the quantity. For efficiency, she provides two functions that sum to the log prior density. One may depend on the parameter itself, but the other cannot. In this way, she can provide a function to evaluate the integration constant of the density, which is only called once, and another to evaluate the kernel of the density. If either function is not provided, the result is as if a function identically zero were provided. The sum of the log densities over all unknown quantities must equal the log density for the joint distribution of all unknown parameters. If the unknown quantities are not a-priori independent, then some of the densities must be conditional. To allow this, the log prior evaluation routines have access to the values of the unknown parameters. The routine to perform the transformation of a quantity is optional. Such a routine is useful when the value of an unknown quantity is restricted to some subset of Euclidean space, in which case a routine to transform the parameter to a (possibly lower dimensional) Euclidean space is useful. The routine must return the log determinant of the transformation, evaluated at the value transformed.

For each Gibbs block, she provides four routines to draw values of unknown quantities from certain distributions. The first two routines are for simulating values from the posterior distribution of the unknown quantities. One is for drawing the first sample, if no others are available, and the other is for drawing subsequent samples. This supports various MCMC algorithms, which require an initial distribution and a transition kernel. The other two routines are for simulating values from the prior distribution of the unknown quantities, and the two routines perform the analogous functions. The routines do not have to be all different. She can write a routine to draw a block directly from the prior, and another routine to draw the block from a transition kernel which preserves the posterior distribution. The former routine can do triple duty as the first,

third and fourth routine described above.

The model developer can provide two initialization routines. One is a model specification initialization routine. It is called once the first time a model specification is accessed. Typically, this routine is not needed, but it may be useful for model developers who wish to use assignment statements rather than declaration initializers to fill in the fields of the model specification structure. The other initialization routine is more useful, and is called once each time a model instance is created. This routine is used to perform calculations that are done only once before simulation. The calculation of sufficient statistics is an example. Note that checking the dimensions and validity of known quantity values is done in other routines, not in this initialization routine. The initialization routine is called after these other two routines, so the model developer can be confident that if the initialization routine is called the known quantity values are reliable.

The model developer also supplies a routine to check the dimensions of the values of the known quantities provided by the user, and to set the dimensions of the auxiliary quantities so that they agree with the dimensions of the known quantities. This routine is obligatory.

She provides two functions to evaluate the log likelihood at given values of the unknown parameters. The two functions must add to the properly normalized log likelihood. One cannot depend on the value of the unknown quantities. This function is useful for evaluating the log integration constant of the likelihood. For efficiency's sake, it is only called once.

If there is a closed form expression for the marginal likelihood, in terms of the known quantities, then the model developer can supply a routine to evaluate the marginal likelihood. Typically, there is no such expression, and no routine is supplied.

Finally, the model developer may supply a function that indicates whether or not a particular value of the unknown quantities is in the support of their posterior distribution. The function has access to the values of the unknown parameters in the case where there is no transformation, and access to the transformed values, where there is a transformation.

7.2 Things the Model Developer Does Not Do

The model developer never evaluates the functions, or calls the routines that she provides when defining a model. She does not have to read in known quantities from files or memory, or elicit anything from the user. She does not have to reserve memory directly for any variables, or store any simulated samples. All this is done by the core routines. Model development has more in common with filling in a form than with programming.

Useful functions are provided for accessing matrices, checking and setting the dimensions of matrices, and performing operations on matrices.

7.3 Using Matrices and the Matrix Library

A special matrix data structure is used for all matrix manipulation. All known quantities, unknown quantities, and auxiliary quantities are stored in this format. The matrix structure (`struct Matrix`) is defined and documented in detail in the file `Includes/bacc_matrx.h` in the BACC distribution. It contains the dimensions of the matrix, the elements of the matrix, and other information.

The matrix structure allows the model developer to treat a matrix as a unit, removing many opportunities for error, and simplifying the calling sequence of matrix functions. Array dimension checking is done automatically and invisibly.

A library of matrix manipulation routines is provided in the `Matrix` directory. These routines are documented in the above-mentioned `bacc_matrx.h`. Routines include low level matrix routines such as multiplications, inversion, and Cholesky decomposition, as well as high level routines for evaluating multivariate densities and drawing from multivariate distributions.

It is highly recommended that the model developer not directly access individual matrix elements in the routines they provide as part of a model specification. The BACC system is designed to facilitate modular programming. The appropriate level for performing operations on matrix elements is in the matrix library. Routines provided by the model developer should only refer to matrices, and not to their elements.

If a model developer wishes to perform some matrix operation as part of the computation in any routine, she should first check to see if such an operation is supported in the library already. If not, she should split the computation into two levels. She writes a low level routine for the matrix library, which takes matrices as arguments, and performs the required matrix operation. She also writes a routine which passes matrix arguments to the low level routine.

7.4 Supplying a Model Specification to the BACC System

The file `Includes/bacc_model.h` describes the `ModelSpec` structure used to contain the routines and functions described above, and the calling sequence of the routines and functions themselves. This information is sufficient for the model developer, but we suggest that she follow along with a previously written model specification to better understand the model specification task. A full annotation of an example is given in the next section.

7.5 An Example: the Normal Seemingly Unrelated Regressions Model

This is an annotation of the model specification file `Models/nlm.c`. We assume the reader is following along with the text of this file. Another useful reference for this annotation is the file `Includes/bacc_model.h`, which defines the structures that a model developer is filling in by writing a model specification file.

7.5.1 Code following `/* Include files */`

Two `#include` statements make visible within this model specification file the definitions of the `Matrx` structure in `bacc_matrx.h` file and the `ModelSpec` structure in the `bacc_model.h` file. The others define the calling conventions of standard C library function and mathematical library functions.

7.5.2 Code following `/* The following functions ...`

This is a list of all the functions supplied by the model developer. The names of the functions are chosen by the model developer. It is the placement of these functions in the `ModelSpec` structure than tells the BACC function which are which. The `static` keyword ensures that the names of the functions are only visible inside this file. This allows more than one model specification file to have a function with the same name, without ambiguity.

7.5.3 Code following `/* Define the known quantities ...`

The enumeration statement assigns the constant values 0, 1, 2, 3, 4, and 5 to the tokens `beta_`, `H_`, `nu_`, `S_`, `X` and `Y`, thus identifying the the known quantities. The token `nKnowns` gets the value 6, which is the number of know quantities.

The `KnownSpec` structure is a substructure of the `ModelSpec` structure. The array `knowns` of type `KnownSpec` and length `nKnowns` contains, for each known, a 0/1 indicator for whether or not the known is integer-valued and a function which indicates whether or not the value of a known quantity is valid. Here, none of the knowns are restricted to be integer-valued. The values of `beta_`, `X`, and `Y` are completely unrestricted as indicated by the keyword `NULL`, and the values of `H`, `nu_`, and `S_` must be positive definite.

According to `bacc_model.h`, the functions which test the validity of the knowns must always take as arguments a pointer to a `Matrx` structure and a pointer to a `ModelInst` (model instance) structure. The second pointer allows access to other information in a model instance. This would be necessary, for example, if the validity of one known quantity depended on the values of other known quantities.

7.5.4 Code following `/* Define the unknown quantities ...`

The enumeration statement identifies the unknown quantities in a similar way. The `UnknownSpec` structure, a substructure of the `ModelSpec` structure, contains, for an unknown quantity, the functions which evaluate the log prior normalization constant and the log prior kernel, the routine which transforms the parameter and returns the log Jacobian, and the routine which reverse transforms the parameter and returns the log Jacobian of the reverse transformation. The array `unknowns` of length `nUnknowns` contains all this information for both the unknown quantities. The order of the routines and their calling conventions are defined in `bacc_model.h` where `UnknownSpec` is defined.

7.5.5 Code following `/* Define the auxiliary ...`

The enumeration statement identifies the auxiliary parameters in a similar way. The two `NULL` statements correspond to two vectors indicating which auxiliary and unknown quantities each auxiliary parameter depends upon. These vectors will be filled in later.

7.5.6 Code following `/* Define the Gibbs blocks ...`

The enumeration statement identifies the blocks in a similar way. The `BlockSpec` structure, also a substructure of the `ModelSpec` structure, contains, for a block, a routine to draw blocks of parameters, a routine to calculate the Hastings ratio, and a vector indicating which unknowns are modified by the block. Two `BlockSpec` structures are created, one for the prior blocks and one for the posterior blocks.

Here the arrays `priorBlocks` and `posteriorKernelBlocks` of type `BlockSpec` contain this information for the two blocks. The `NULL` second element of each array indicates that routines to calculate the Hastings ratio are not provided. The third array element is also `NULL`, but this will be filled in later using the `initSpec` routine.

7.5.7 Code following `/* Define the dimension ...`

The enumeration statement identifies the dimension parameters in a similar way.

7.5.8 Code following `/* Define the model ...`

This simply fills in the model structure with all of the information that is needed to completely specify the model.

7.5.9 Code following `/* Function definitions follow ...`

The remainder of the model specification file is a series of routines that handle the tasks mentioned in the specification of the various structures. There are routines to check parameters for proper form (eg. `positiveDefinite`), check the dimensions of the knowns and create the unknown and auxiliary parameters (eg. `dimensions`), evaluate the log constant, log kernel, and transformations of each prior distribution (eg. `betaConst` and `betaKernel`), handle the prior and posterior draws (eg. `betaPriorDraw` and `betaDraw`), compute the auxiliary quantities (eg. `uConstruct`), initialize the model structure by filling in the auxiliary parameter structure and the Gibbs block structures (eg. `initSpec`), and finally, routines to compute the log likelihood constant and log likelihood kernel (eg. `logLikeConstant` and `logLikeKernel`).