BuGS 1.0 User Guide

Bruce P. Ayati

Abstract

This is the user guide for Bruce's General Solver (BuGS). BuGS is a toolkit for solving single space dimensional, nonlinear systems of partial differential equations which are at most order one in time and order two in space. The user defines the spatial discretization of the equations by writing a residual function based on first order backward differences in time. BuGS then uses a second order in time implicit finite difference scheme based on backward differences. BuGS features step-doubling, step control for the convergence of Newton's method, and automatic approximation of the Jacobi matrix.

Key words and phrases: second order nonlinear partial differential equations, single space dimension, implicit finite difference scheme.

AMS Subject Classifications(1991): 35G30, 65N06.

Contents

1 Introduction **3**
1.1 Disclaimer . . . 3
1.2 Second Order Nonlinear PDE's . . . 3
1.3 How to Acquire BuGS . . . 4
1.4 Acknowledgments . . . 4

2 Numerics of the Shell **4**
2.1 Data Structures and Representation of the Solution . . . 4
2.2 Algorithm . . . 5
2.3 Approximate Jacobian . . . 6

3 Classes and Prototypes of the Code **6**
3.1 Classes . . . 7
3.2 The Library . . . 9
3.3 Non-Library Routines . . . 10
3.4 Problem Dependent Routines . . . 11
3.5 Naming Conventions . . . 11

4 Example **11**
4.1 Problem . . . 11
4.2 Results . . . 12

5 How to Build a BuGS Application **13**
5.1 Initializing the System . . . 14
5.1.1 Example of setup.c . . . 15
5.2 Discretizing the system of PDE's . . . 17
5.2.1 Example of resid.c . . . 18
5.3 Generating Output . . . 20
5.3.1 Example of dataOutput.c . . . 21
5.4 Setting Parameters . . . 23
5.4.1 Example of parameters.h . . . 23
5.5 Additional Modifications and Compilation . . . 24

1 Introduction

This is the user guide for Bruce's General Solver (BuGS). BuGS approximates solutions to single space dimensional, nonlinear systems of partial differential equations (PDE's) that are at most order one in time and order two in space. BuGS is written in C++. It was originally developed to test models of the growth of bacteria colonies.

The actual definition of the partial differential equation and boundary conditions is done by the user providing a function $F(U, U_t)$. This function defines a two-level backward difference scheme in which the difference equations at each point depend on the solution only at the point in question and its neighbors.

1.1 Disclaimer

BuGS is a research tool, not a commercial product. It is made available to other researchers subject to the following restrictions and disclaimers. It is requested that use of BuGS cite this manual.

> Copyright © 1996 by Bruce Pirooz Ayati. Any program source code made available is for non-commercial use only. The author makes no representations or warranties about the correctness of any program code or documentation in this or any other document or program file, nor about the correctness of the executable program or its suitability for any purpose. The author is in no way liable for any damages resulting from the use or misuse of this program.

1.2 Second Order Nonlinear PDE's

Let $I = (x_0, x_{N-1})$ and $J = (T_0, T_f)$, where $x_0 < x_{N-1}$ and $T_0 < T_f$. BuGS approximates solutions to systems of the form

$$F(U(x,t), U_t(x,t)) = 0, \tag{1}$$

$x \in I$, $t \in J$, where F is a possibly nonlinear differential operator of at most order two in x and $U : \bar{I} \times \bar{J} \to \mathbb{R}^n$. Suitable boundary, regularity, and initial conditions must apply.

Two examples of such F and U are as follows:
For $n = 1$:

$$F(U, U_t) = U_t - (UU_x)_x \,. \tag{2}$$

For $n = 2$:

$$\begin{aligned} U(x,t) &= (\beta(x,t), \rho(x,t)), \\ F(U, U_t) &= \begin{bmatrix} \beta_t - (\beta\beta_x)_x - \beta\beta_x\rho_x \\ \rho_t - \beta \end{bmatrix}. \end{aligned} \tag{3}$$

Note: BuGS can solve higher order systems of partial differential equations if they are rewritten as second order systems in the form of equation (1).

1.3 How to Acquire BuGS

BuGS is currently available via the World Wide Web at URL:

http://www.cs.uchicago.edu/~bruce/software/

1.4 Acknowledgments

I thank my advisor, Professor Todd F. Dupont, for his assistance with BuGS and this manual. This work made use of MRSEC Shared Facilities supported by the National Science Foundation under Award Number DMR-9400379.

2 Numerics of the Shell

2.1 Data Structures and Representation of the Solution

BuGS represents vectors as objects of the class **interwoven** or as arrays of double precision numbers. The class **interwoven** is used for arrays that are a combination of several other arrays, such as the residual array **r** which is a combination of the residuals of the dependent variables. Arrays in this class can be referenced by parenthesis, not braces as is normally done with C++ arrays. **array**(i, j) refers to the value of the jth vector interwoven into **array** at the ith spatial node. For non-interwoven arrays, such as the spatial mesh, BuGS uses arrays of of double precision numbers, indexed with braces as usual.

The array **variables** stores the solution at a given time step. **variables** has size $n \cdot N$, where n is the number of dependent variables and N is the number of nodes in the mesh. **variables** stores the solution in an interwoven manner. The first n components correspond to the solution of the system at the first node, the next n components correspond to the solution at the second node, and so on. For Equation (3), $n = 2$ and thus

$$\textbf{variables} = [\beta(x_0), \rho(x_0), \beta(x_1), \rho(x_1), \ldots, \beta(x_{N-1}), \rho(x_{N-1})]\,. \qquad (4)$$

The arrays **dvariables** and **tvariables** store the guesses at the change over the time step and the solution at the end of the time step, respectively. They are passed to the residual routine, **resid**.

The array **r** stores the residual from Newton's method at a time step. **r** has the same structure as **variables**. The array **terms** stores the value of individual terms in the differential system, at a time step, in an interwoven fashion similar to **variables**.

BuGS uses approximate Jacobians of the difference equations for Newton's method. The approximate Jacobi matrices are restricted to block tridiagonal form. This restriction requires the discretization of the PDE to involve only the solution at the point in question and its neighbors. The structure of a tridiagonal matrix, A, is defined by the dimension of the blocks, $n \times n$, and the number of blocks in either the row or column direction, N. Therefore, A has size $(n \cdot N) \times (n \cdot N)$. For linear algebra, BuGS uses routines that take advantage of this structure. BuGS represents the approximate Jacobi matrices as objects of the class **tridiag**.

2.2 Algorithm

BuGS uses a nonlinear implicit finite difference scheme that is a first order backward difference in time. The algorithm is as follows:

I. Setup the system.
II. Loop over the output times
   A. Advance the solution.
      1. Guess the time step, **dt**.
      2. Do step doubling and compute the reduction in the residual.
      3. Check if the step is good. If not, halve **dt** and go back to 1.
      4. Extrapolate the solution using those obtained in step doubling.
      5. Update the state of the system.
      6. Send time-dependent data to the output files.
      7. Maintain the summary data (end profiles).
   B. Send space-dependent data (profiles) to the output files.
   C. Go back to A until the next output time is reached.
III. Send the summary data to the output files.

In 1, guess the time step by comparing the previous time and time step with the next designated output time. In 2, take two steps with step size **dt**/2, then take one step with step size **dt**. Use the guess at the change over the time step from the first two steps as a better guess of the change over the time step for the single step. In each of these steps, compute the residual and then use Newton's method to compute the solution. In 3, the difference between the solutions obtained by the two steps and the one step must be less than **tol** in the L^∞-norm. For each backward difference step, compute the L^∞-norm of the residual before and after the step. The ratio of these, with a patch for rounding, is the residual reduction. The sum of the reductions in the residual for all three steps must be less than **r_tol**. In 4, set the solution to $2d - s$, where d is the solution from taking two steps over the time interval and s is from a single step [4]. In 5, determine the new state of the system at each node by considering the previous state as well as the current values of the variables. The different types of output consist of the following: time-dependent output, such as the spatial maximum at each point in time; space-dependent output, such as the state of a system at a given time; and summary output, such as the maximum value achieved at each point in space over all time.

2.3 Approximate Jacobian

An important design goal of BuGS is the ability to change the differential problem easily. Therefore, explicitly writing the Jacobi matrix is avoided [3]. This is done by approximating the derivative of the residual, for every dependent variable and at each node, by a finite difference

$$R'(y,x) \approx \frac{R(y+\varepsilon) - R(y)}{\varepsilon}. \tag{5}$$

Here R is the residual function, y is a particular dependent variable at a node x, and ε is chosen to be suitably small.

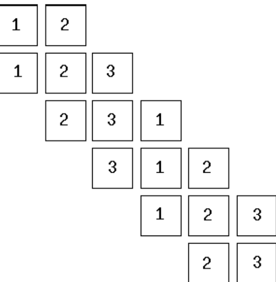

Figure 1: Block Matrix

A forward difference is used instead of a center difference to minimize the number of calls to **resid**. Caution must be taken with a residual that is not differentiable everywhere.

The block tridiagonal structure of the Jacobi matrix can be exploited to require only $3n+1$ calls to **resid**, where n is the number of dependent variables in the system. In Figure 1, the blocks with the same number don't interact with each other. By grouping the blocks with the same number into a single column of blocks, we get a collection of n column vectors,

$$\bar{R}'(Y) \approx \frac{\bar{R}(Y+\bar{\varepsilon}) - \bar{R}(Y)}{\bar{\varepsilon}}, \tag{6}$$

that each require only one call to **resid**. The elements in Y correspond to one of the column vectors. $\bar{R}$ is a vector of residuals, and $\bar{\varepsilon}$ is a vector of appropriate permutations. Since there are three sets of blocks, we need only $3n+1$ calls to **resid**.

3 Classes and Prototypes of the Code

BuGS uses several classes for output and matrix representation. The body of the code is divided into three parts. The first part is the library. The library routines

are least likely to be modified by the user. The second part is the collection of non-library routines. Non-library routines are dependent upon problem specific global constants. They are also more likely to need modification than library routines. The third part is the collection of problem dependent routines. These need to be modified for each problem. In addition, the header file, **parameters.h**, is dependent on the particular problem.

3.1 Classes

Compressed Write Channel used for compression of the output [1].

```
class pair{
  public:
    double x,y;
    pair(double a, double b){x=a;y=b;};
};

class CmpChannel{
    double last_point, last_value;
    double prev_point, prev_value;
    double lower_slope, upper_slope;
    double abs_tol, rel_tol;
    ofstream *output;
    char* name;
    int stuffs;
    static int num_comp_ch;
  public:
    CmpChannel(char* ch_name=0, double atol=1.e-20, double rtol=1.e-6);
    void reset(char* remark=0);
    ~CmpChannel();
    void stuff(double, double, double=-1., double=-1.);
    friend CmpChannel& operator << (CmpChannel& cmpch, char* str );
    friend CmpChannel& operator << (CmpChannel& cmpch, double d );
    friend CmpChannel& operator << (CmpChannel& cmpch, int i );
    friend CmpChannel& operator << (CmpChannel& cmpch, pair );
};
```

Representation of a block tridiagonal matrix using standard matrix referencing.

```
class tridiag{
   private:
     int size;
     int blk;
     double *m;
     int *ind;
   public:
     tridiag(int n, int unk){   //contructor
```

```
      size = n;
      blk = unk;
      m = new double[(3*blk*blk)*(size)];
      ind = new int[size*blk];
      for(int rn=0;rn<size*blk;rn++){ind[rn] = rn*(3*blk)-((rn/blk) - 1)*blk;}
     }
     ~tridiag(){delete[] m; delete[] ind;}        //destructor
     double& operator() (int rn, int cn) {
       assert(rn>=0 && rn<size*blk && cn>=0 && cn<size*blk);
       return m[ind[rn]+cn];  //subscripting
     }
     void zero()
     {int loopEnd = 3*blk*blk*size; for (int i=0; i<loopEnd; i++) m[i] = 0.0;}
     int length(){return size;}
     int blkNum(){return blk;}
};
```

Representation of solutions is done using the class **interwoven**.

```
class interwoven{
   private:
     int size;
     int vars;
     double *v;
     int *ind;
   public:
     interwoven(int n, int m){    //constructor from scratch
      size = n;
      vars = m;
      v = new double[size*vars];
      ind = new int[size];
      for(int ln=0; ln<size; ln++) ind[ln] = ln*vars;
     }
     interwoven(interwoven& I){    //contructor from another interwoven
      size = I.size;
      vars = I.vars;
      v = new double[size*vars];
      ind = new int[size];
      for(int ln=0; ln<size; ln++) ind[ln] = ln*vars;
     }
     ~interwoven(){delete[] v; delete[] ind;}        //destructor
     double& operator() (int ln, int lm) {  // double index subscripting
       assert( ln>=0 && ln<size && lm>=0 && lm<vars);
       return v[ind[ln]+lm];
     }
     double& operator() (int l) { // single index subscripting
       assert(l>=0&&l<size*vars);
```

```
        return v[l];
      }
      void operator=(const interwoven& x) { // *this=x
          int i, n=x.size, m=x.vars;
          assert(size==n && vars==m);
          for(i=0; i<n*m; i++) v[i]=x.v[i];
      };
      void operator+=(const interwoven& x) {//form x+y
          int i, n=x.size, m=x.vars;
          assert(size==n && vars==m);
          for(i=0;i<n*m;i++) v[i]+=x.v[i];
      }
      void operator-=(const interwoven& x) {//form x+y
          int i, n=x.size, m=x.vars;
          assert(size==n && vars==m);
          for(i=0;i<n*m;i++) v[i]-=x.v[i];
      }
      void operator*=(const double c){for(int i=0;i<size*vars;i++) v[i]*=c;}
      void zero() { for (int i=0; i<size*vars; i++) v[i] = 0.0;}
      int length(){return size;}
      int varNum(){return vars;}
      double maxabs(){
          double m = fabs(v[0]);
          for(int i=1; i<size*vars; i++){if( fabs(v[i])>m)m=fabs(v[i]);}
          return m;
      }

  friend double operator,(const interwoven&, const interwoven&);
  friend void increment( interwoven& x, double c, interwoven& y);
  friend void lin_comb( interwoven& v, double c1, interwoven& x,
                         double c2, interwoven& y);
  friend void max( interwoven& v, interwoven& x, interwoven& y);
};
```

3.2 The Library

advance advances time. It uses **step** and **ck_step**. It is also where **end_prof** and **time_output** are called.

```
void advance(double& time,double tout,double x[], double order[], int state[],
             interwoven& variables, interwoven& dvariables,
             interwoven& terms, interwoven& endProf,
             double& dt,double& dtold, double dtmax,
             double dtmin,double dtTriv,double tol,
             interwoven& r, double rTol,
             int& numStep,int& rejectedSteps);
```

advance uses **ck_step** to see if a given step is good enough, both in terms of residual reduction and step doubling.

```
int ck_step( interwoven& variables, interwoven& variablesBig,
             double& dt,double& dtold,int nssrs,
             double tol, double rTol, double dtmin,double dtmax,
             double& errorIndicator,
             interwoven& r, double& rr, int errorVect[]);
```

jacobian computes the automated approximation of the Jacobi matrix.

```
void jacobian(double x[], double order[],
                interwoven& terms, double time,double dt,
                interwoven& tvariables, interwoven& dvariables,
                tridiag& a, int state[]);
```

The piecewise linear compressed write routine used to write a profile [1].

```
void write_prof( CmpChannel & out , int len, double x[], interwoven& v,
                 int var );
```

step executes the step doubling and carries information about the reduction in the residual.

```
void step(double x[], double order[],
          interwoven& variables, interwoven& dvariables,
          interwoven& terms, double& time,double dt,
          double& dtold, interwoven& r, double& rr, int state[]);
```

Tridiagonal Matrix Routines.

```
void tridiag_fac(tridiag& a);
void tridiag_sol(tridiag& a, interwoven& r);
```

Utility Functions

```
void error( char* );
double pwl_integral(int n, double x[], interwoven& v, int var);
double pwl_slope_jump(int n, double x[], interwoven& v, int var);
double min(double x,double y);
double max(double x,double y);
int min(double x,double y);
int max(double x,double y);
double max( interwoven& v, int var);
double Alog10( double x);
```

3.3 Non-Library Routines

The main program. It sets up the initial data structures and conditions for the problem, using **setup** for problem specific initialization. It loops over **advance** to advance the solution to the next time step, and uses **prof_output** and **end_prof_output** to send the profiles to the output files.

```
main();
```

next_output_time returns the time of the next profile. Output times are in regular intervals of length **NEXT_OUT_TIME**.

```
double next_output_time( double time, double tout );
```

3.4 Problem Dependent Routines

Data Output Routines. **prof_output** sends the state of the solution at designated intervals to the output files. **end_prof** maintains summary profiles, such as the pointwise maximum in space over all time. **end_prof_output** sends the summary profiles to the output files. **time_output** sends the time dependent data, such as the maximum over time, to the output files.

```
void prof_output( double x[], interwoven& variables, interwoven& r,
                  interwoven& terms, double time);
void end_prof( interwoven& endProf, interwoven& variables);
void end_prof_output( double x[], interwoven& endProf, double time );
void time_output( double x[], interwoven& variables, double time, double dt,
                 int firstTime, double errorIndicator, int errorVect[]);
```

In **resid**, the user provides the residual that defines the system of PDE's.

```
void resid(double x[],double time, double dt, interwoven& tvar,
           interwoven& dvar, interwoven& r, interwoven& terms,
           int state[]);
```

In **setup**, the user provides the initial grid, the initial state of the variables, tolerances for the code, time stepping information, and the order of the dependent variables in the system of PDE's.

```
void setup( double x[], double order[], interwoven& variables,
           interwoven& endProf, double& time, double& tfinal);
```

In **states**, the user provides history dependent information. For most applications, the function will do nothing and consist of an open and close bracket.

```
void states( int state[],double x[], interwoven& variables, int firstTime );
```

The user provides global constants in the file **parameters.h**

3.5 Naming Conventions

Except for the compressed write routines, the following naming conventions are used. If the name is **do nothing**, then it is written as such:

Style	Meaning
doNothing	variable or source code file
do_nothing	function or output file
DO_NOTHING	global constant

4 Example

4.1 Problem

An example of Equation (1) with suitable initial and boundary conditions is the system:

$$\begin{aligned}
\beta_t - (\beta\beta_x)_x - K(\beta\rho_x)_x &= 0, & (7)\\
\rho_t - \beta - K(\beta\rho_x)_x + \text{rate}(\rho,\alpha) &= 0, & (8)\\
\alpha_t - \text{rate}(\rho,\alpha) &= 0, & (9)
\end{aligned}$$

$$\text{rate}(\rho,\alpha) = \max\{C\rho(\alpha_{\max} - \alpha), 0\}, \quad (10)$$

with boundary and initial conditions

$$\beta_x = \rho_x = 0 \text{ on } \{0, a\}, \quad (11)$$

$$\begin{aligned}
\beta(x,0) &= \max\{b - x^2, 0\},\\
\rho(x,0) &= 0, \qquad (12)\\
\alpha(x,0) &= 0.
\end{aligned}$$

Here $x_0 = 0$, $x_{N-1} = a$, and $T_0 = 0$. This system models the movement of bacteria on a petri dish. The bacteria are moving away from their own waste. β is the bacterial population density, ρ is the concentration of waste or repellent, and α is the amount of repellent absorbed by the laboratory medium.

4.2 Results

For the system (7)- (12) with constants $K = 100$, $C = 100$, $\alpha_{\max} = 0.06$, $a = 25$, $b = 0.5$, $T_f = 16$, and $N = 501$, BuGS gives Figures 2-10 as sample output. The example run was compiled using the Sun C++ compiler CC under optimization level three. Floating point traps for the IEEE exceptions inexact and underflow were turned off, as were the asserts used for index checking. The run used 99.6% of a 66 MHz hyperSPARC CPU for a run time of 586.15 seconds. There were 1745 steps accepted, and 104 steps rejected.

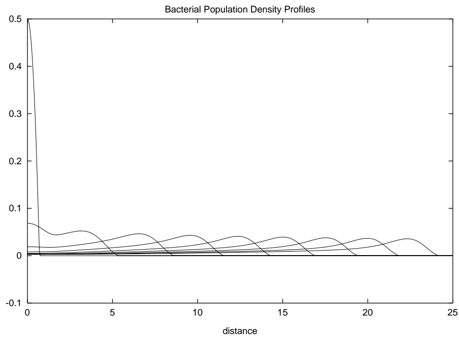

Figure 2: Space-dependent output for β

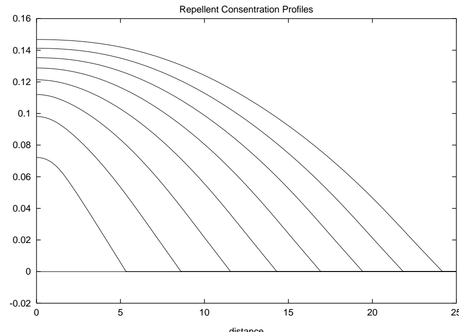

Figure 3: Space-dependent output for ρ

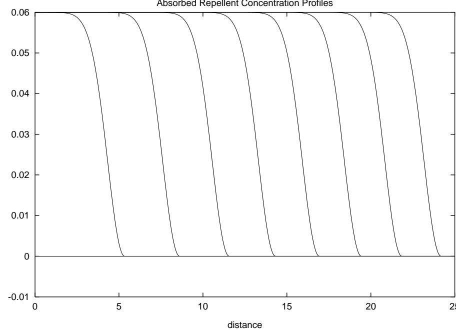

Figure 4: Space-dependent output for α

5 How to Build a BuGS Application

You must tell BuGS how to do four things:

- Initialize the system. This includes defining the spatial grid and the initial conditions for the PDE. This is done in the file **setup.c**.
- Discretize the system. Define the residual that describes the backward difference method for the system of PDE's and boundary conditions. This is done in the file **resid.c**.
- Generate the output. Decide which profiles, time-dependent data, and summary data are to be written to output. This is done in the file **dataOutput.c**.
- Set parameters. Set the numerical value of certain parameters, as well as define and set problem specific parameters. This is done in the file **parameters.h**.

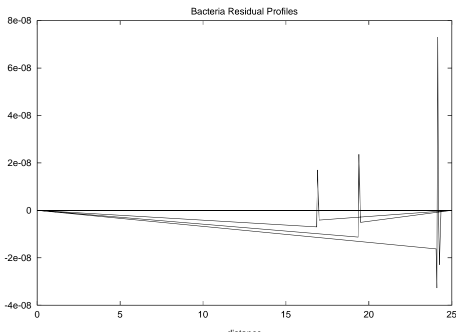

Figure 5: Space-dependent output for the residual of β

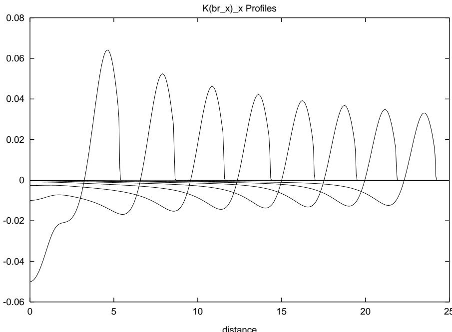

Figure 6: Space-dependent output for the term $K(\beta\rho_x)_x$

For problems that are history dependent, the file **state.c** is provided. For the example problem and most applications, this file can be left as just the prototype and ignored by the user.

The example files presented in this section are based on the problem described in Section 4.1.

5.1 Initializing the System

In the file **setup.c**, define the spatial grid by defining the values of the array **x**. Also set the initial conditions by specifying values for the array **variables**. Since BuGS approximates the Jacobi matrix, specify the order of each of the dependent variables in the array **order**. The value of **order**(i) is the order of the dependent variable corresponding to **variables**$(\cdot, i)$. Specify the temporal domain by setting the values of **time** and **tfinal**. Initialize the summary data by setting the array **end_profs**.

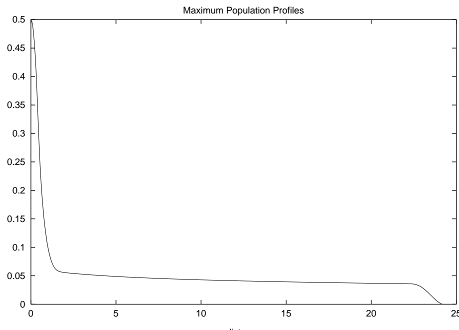

Figure 7: Summary profile for β

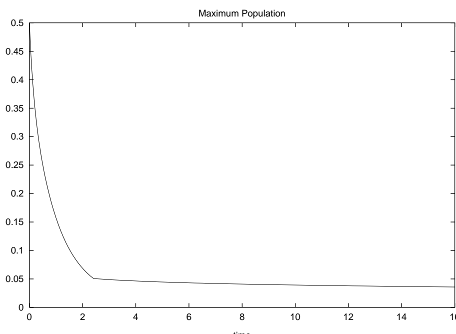

Figure 8: Time-dependent output for β

5.1.1 Example of setup.c

```
#include "parameters.h"
#include "interwoven.h"
#include <iostream.h>

// setup the grid
void set_grid( double x[] )
{
   x[0]=0.0;     // data
   x[N-1]=X_L;    // data

   int i;
   double dx = (x[N-1]-x[0])/(N-1);

   for( i=1 ; i<N-1 ; i++ )
      x[i] = x[0] + i*dx;

   for( i=1; i<N ; i++ ){ // check grid
```

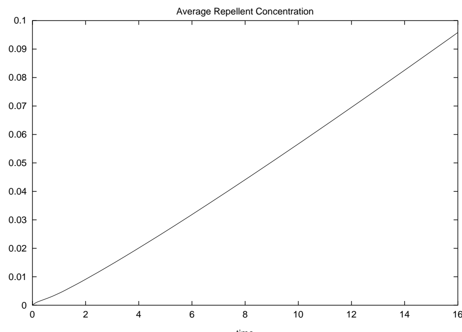

Figure 9: Time-dependent output for ρ

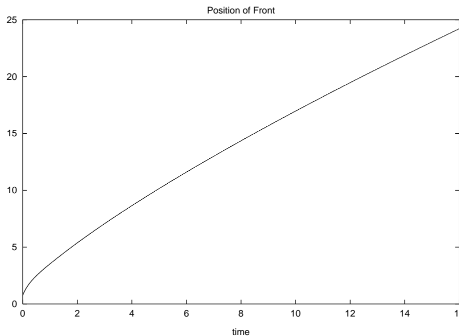

Figure 10: Time-dependent output for β

```
      if( x[i]<=x[i-1] ){
        cerr << "Bad grid" << endl;
        exit(0);
      }
   }
}

// setup the initial conditions
double init_cond( double x )
{
  double ic;
  ic = 0.5 - x*x;
  if(ic > 0)
    {return ic;}
  else
    {return 0;}
}
```

```
void setup( double x[], double order[], interwoven& variables,
            interwoven& endProf, double& time, double& tfinal)
{
    // The spatial grid.
    set_grid(x);

    // Initial conditions.
    for(int i=0;i<N;i++){    // initial variables
       variables(i,0) = init_cond( x[i] );
       variables(i,1) = 0;
       variables(i,2) = 0;
    }

    // Set pointwise maxes.
    endProf = variables;

    // Set order of magnitude of variables.  Must have UNK entries.
    order[0] = 10e-1;
    order[1] = 10e-1;
    order[2] = 10e-2;

    // Initial and final times.
    time = 0.0;
    tfinal = T_F;
}
```

5.2 Discretizing the system of PDE's

Write a residual function that computes the value of **r**, where **r**$= \hat{F}(\hat{U}, \hat{U}_t)$, a discretization of the operator F from Section 1.2. Due to constraints on the Jacobi matrix, the discretization can only include the point in question and its nearest neighbors. Boundary conditions are set when discretizing the spatial nodes 0 and $N-1$.

The arrays **tvariables** and **dvariables/dt** represent $\hat{U}$ and $\hat{U}_t$, respectively. In the example problem,

$$U = \begin{pmatrix} \beta \\ \rho \\ \alpha \end{pmatrix}. \tag{13}$$

At spatial node i, the approximated value of β is stored in **tvariables**$(i, 0)$, the value of ρ in **tvariables**$(i, 1)$, and the value of α in **tvariables**$(i, 2)$. Indexing is similar for **dvariables**.

The array **terms** can be used to store the value of specific terms in the system of PDE's, such as a diffusion term, for output.

5.2.1 Example of resid.c

```
#include "parameters.h"
#include "interwoven.h"

double rate(double r, double a)
{//auxillary function called by resid
 double rr = 0;
 if(r*(A_MAX - a) > 0){rr = C*r*(A_MAX - a);}
 return rr;
}

void resid(double x[],double time, double dt, interwoven& tvar,
           interwoven& dvar, interwoven& r, interwoven& terms,
           int state[])
{/*
    Backward difference equation residual for an UNKxUNK system

    Here tvariables is the UNK*N-vector of the guesses at the
    end of the step. dvariables is the guesses at the change
    over the step.

    The residual is returned in the UNK*N-vector r.

    At each node, unknown1 is first ... unknownUNK is UNK.
 */

int i, p, m;
double dx, dx1, dx0, dxx;
double b, b1, b0, bx, bx0, bx1, bt, re, rt, rx, rx1, rx0, a, at, arate;

for (i=1;i<N-1;i++){
  p = i+1;
  m = i-1;

  dx = x[p] - x[m];
  dx1 = x[p] - x[i];
  dx0 = x[i] - x[m];
  dxx = 0.5*(x[p]-x[m]);
  b = tvar(i,0);
  bt = dvar(i,0)/dt;
  bx = (tvar(p,0) - tvar(m,0))/dx;
  b1 = (tvar(p,0) + b )/2;
```

```
  b0 = ( b + tvar(m,0))/2;
  bx1 = (tvar(p,0)-tvar(i,0))/dx1;
  bx0 = (tvar(i,0)-tvar(m,0))/dx0;
  re = tvar(i,1);
  rt = dvar(i,1)/dt;
  rx = (tvar(p,1) - tvar(m,1))/dx;
  rx1 = (tvar(p,1)-tvar(i,1))/dx1;
  rx0 = (tvar(i,1)-tvar(m,1))/dx0;
  a = tvar(i,2);
  at = dvar(i,2)/dt;
  arate = rate(re,a);

  terms(i,0) = ( b1*bx1 - b0*bx0 )/dxx;
  terms(i,1) = K*(b1*rx1 - b0*rx0 )/dxx;

  r(i,0) =  bt - terms(i,0) - terms(i,1);
  r(i,1) =  rt - b - terms(i,1) + arate;
  r(i,2) =  at - arate;
}

// Impose Neumann Boundary Conditions
  b = tvar(0,0);
  bt = dvar(0,0)/dt;
  b1 = (tvar(1,0)+tvar(0,0))/2;
  bx1 = (tvar(1,0)-tvar(0,0))/(x[1]-x[0]);
  re = tvar(0,1);
  rt = dvar(0,1)/dt;
  rx1 = (tvar(1,1)-tvar(0,1))/(x[1]-x[0]);
  a = tvar(0,2);
  at = dvar(0,2)/dt;
  arate = rate(re,a);

  terms(0,0) = ( b1*bx1 - 0 )/(0.5*(x[1]-x[0]));
  terms(0,1) = K*(b1*rx1 - 0 )/(0.5*(x[1]-x[0]));

  r(0,0) = bt - terms(0,0) - terms(0,1);
  r(0,1) = rt - b - terms(0,1) + arate;
  r(0,2) = at - arate;

  p = N-1;
  m = N-2;

  b = tvar(p,0);
  bt = dvar(p,0)/dt;
  b0 = (tvar(p,0)+tvar(m,0))/2;
  bx0 = (tvar(p,0)-tvar(m,0))/(x[p]-x[m]);
```

```
  re = tvar(p,1);
  rt = dvar(p,1)/dt;
  rx0 = (tvar(p,1)-tvar(m,1))/(x[p]-x[m]);
  a = tvar(p,2);
  at = dvar(p,2)/dt;
  arate = rate(re,a);

  terms(p,0) = ( 0 - b0*bx0 )/(0.5*(x[p]-x[m]));
  terms(p,1) = K*(0 - b0*rx0 )/(0.5*(x[p]-x[m]));

  r(p,0) = bt - terms(p,0) - terms(p,1);
  r(p,1) = rt - b - terms(p,1) + arate;
  r(p,2) = at - arate;

}//resid
```

5.3 Generating Output

To generate output, write four routines in the file **dataOutput.c**:

- **end_prof**, which maintains data that is a summary of other data over time. In the example problem, the summary data is the maximum of β at each point in space over the time of the simulation.
- **end_prof_output**, which sends the summary to an output file.
- **time_output**, which sends time-dependent data to an output file. An example is the maximum value reached by β at each time step.
- **prof_output**, which sends space-dependent data to an output file. An example is a series of profiles.

These routines must be in place, even if they are not going to be used.

The output is generated by compressed write routines for Gnuplot. The file **dataOutput.c** can be easily modified for other visualization software, mainly by removing comments. To open an output file, declare a variable of type **CmpChannel**. To write a profile of the solution, use the routine **write_prof**. **write_prof** takes as arguments, in order, the name of the compressed write channel, the number of nodes in the spatial mesh, an object of class **interwoven**, and the number of the desired array interwoven into that object. To write a pair of numbers, such as for time-dependent output, use the routine **pair**. The utility functions described in the library can be used to customize the type of data sent to output.

5.3.1 Example of dataOutput.c

```
#include "parameters.h"
#include "pwlcmp.h"
#include "interwoven.h"
#include "utility.h"

void prof_output( double x[], interwoven& variables, interwoven& r,
                  interwoven& terms, double time)
{ /*output profiles */

   static CmpChannel p_profiles ("p_profiles",1.e-8,1.e-6);
   static CmpChannel r_profiles ("r_profiles",1.e-8,1.e-6);
   static CmpChannel a_profiles ("a_profiles",1.e-8,1.e-6);
   static CmpChannel p_resid ("p_resid",1.e-8,1.e-6);
   static CmpChannel r_resid ("r_resid",1.e-8,1.e-6);
   static CmpChannel a_resid ("a_resid",1.e-8,1.e-6);
   static CmpChannel terms0 ("terms0",1.e-8,1.e-6);
   static CmpChannel terms1 ("terms1",1.e-8,1.e-6);

   p_profiles << "# Population at time " << time << "\n";
   write_prof( p_profiles, N, x, variables, 0 );
   r_profiles << "# Repellent at time " << time << "\n";
   write_prof( r_profiles, N, x, variables, 1 );
   a_profiles << "# Absorption at time " << time << "\n";
   write_prof( a_profiles, N, x, variables, 2 );
   p_resid << "# Population Residual at time " << time << "\n";
   write_prof( p_resid, N, x, r, 0 );
   r_resid << "# Repellent Residual at time " << time << "\n";
   write_prof( r_resid, N, x, r, 1 );
   a_resid << "# Absorbed Repellent Residual at time " << time << "\n";
   write_prof( a_resid, N, x, r, 2 );
   terms0 << "# term 0 at time " << time << "\n";
   write_prof( terms0, N, x, terms, 0 );
   terms1 << "# term 1 at time " << time << "\n";
   write_prof( terms1, N, x, terms, 1 );

}//end prof_output


void end_prof( interwoven& endProf, interwoven& variables)
{
 max(endProf,endProf, variables);
}//end end_prof
```

```
void end_prof_output( double x[], interwoven& endProf, double time )
{ /*output pointwise maximums of the quantities*/

  static CmpChannel p_max_profiles ("p_max_profiles",1.e-8,1.e-6);

  p_max_profiles << "# Pointwise Maximum Population at time "
                << time << "\n";

  write_prof( p_max_profiles, N, x, endProf, 0 );
}//end end_prof_output


double front_position( double x[], interwoven& pop )
{
 double tol = 10e-10;
 for(int i=N-1; i>=0; i--){
    if(pop(i,0) > 10e-8)
      {return x[i];}
 }
}//end front_position


void time_output( double x[], interwoven& variables, double time, double dt,
                 int firstTime, double errorIndicator, int errorVect[])
{
   double fpos;
   double spaceErrorIndicator = 0.0;

   static CmpChannel out_dt ("out_dt",1.e-4,1.e-2);
   static CmpChannel out_pop_max ("out_pop_max",1.e-6,1.e-4);
   static CmpChannel out_pop_ave ("out_pop_ave",1.e-6,1.e-4);
   static CmpChannel out_rep_ave ("out_rep_ave",1.e-6,1.e-4);
   static CmpChannel out_abs_ave ("out_abs_ave",1.e-6,1.e-4);
   static CmpChannel out_time_error ("out_time_error",1.e-4,1.e-2);
   static CmpChannel out_space_error ("out_space_error",1.e-4,1.e-2);
   static CmpChannel out_front_position ("out_front_position",1.e-6,1.e-4);
   static CmpChannel out_perror_position ("out_perror_position",1.e-6,1.e-4);
   static CmpChannel out_rerror_position ("out_rerror_position",1.e-6,1.e-4);
   static CmpChannel out_aerror_position ("out_aerror_position",1.e-6,1.e-4);

   fpos = front_position( x, variables );

   out_dt << pair(time,Alog10(dt));
   out_pop_max << pair(time, max(variables,0));
   out_pop_ave << pair(time, pwl_integral(N,x,variables,0)/(x[N-1]-x[0]));
   out_rep_ave << pair(time, pwl_integral(N,x,variables,1)/(x[N-1]-x[0]));
```

```
   out_abs_ave << pair(time, pwl_integral(N,x,variables,2)/(x[N-1]-x[0]));

   out_front_position << pair(time, fpos);
   out_perror_position << pair(time, x[errorVect[0]]);
   out_rerror_position << pair(time, x[errorVect[1]]);
   out_aerror_position << pair(time, x[errorVect[2]]);

   if( !firstTime ){
     out_time_error << pair(time,Alog10(errorIndicator));
     spaceErrorIndicator = pwl_slope_jump(N,x,variables,0);
     out_space_error << pair(time,Alog10(spaceErrorIndicator));
   }

}// end time_output
```

5.4 Setting Parameters

Set five parameters for every problem: **N**, the number of nodes in the spatial mesh; **UNK**, the number of dependent variables in the problem; **TERM_NUM**, the number of terms in the system of PDE's sent separately to output; **END_PROF_NUM**, the number of summary profiles sent to output; and **NEXT_TIME_OUT**, the time between profiles. In this example, the initial time and left boundary are assumed to be always 0, so the only other parameters to set are **X_L**, the spatial endpoint, and **T_F**, the final time.

In addition, problem specific parameters, such as constants used in **resid**, can be set in **parameters.h**.

5.4.1 Example of parameters.h

```
#ifndef PARAMETERS_h
#define PARAMETERS_h

const int N = 101; // number of mesh points
const int UNK = 3; // number of dependent variables
const int TERM_NUM = 2; // number of terms in the system
const int END_PROF_NUM = 3; //number of final profiles for output
const double NEXT_TIME_OUT = 2.0; //Output interval for profiles
const double X_L = 25.0;
const double T_F = 16.0;

/*Problem Specific Parameters*/
const double K = 100;
const double C = 100;
```

```
const double A_MAX = 0.06;

#endif
```

5.5 Additional Modifications and Compilation

For some applications, additional modifications may be required. These modifications may include library functions. For example, conditions on terminating a problem or determining when a step is good should be implemented in the routines **ck_step** and **advance**, possibly by having them call other routines.

The makefiles provided with BuGS are for machines running Solaris. Problems may arise concerning the compilation of libraries. Changes for other platforms may include putting all files in the same directory with a single makefile.

For the most efficient binary, asserts and floating point traps for inexact and underflow IEEE exceptions should be deactivated. Asserts are turned off using the flag -**DNDEBUG** and the floating point traps on the Suns are turned off using -**fnonstd** for CC and -**ffast**-**math** for g++.

References

[1] The compressed write routines are provided by Todd F. Dupont and Mark Weber.

[2] John, F. 1982 *Partial Differential Equations*, Springer-Verlag, New York.

[3] Radhakrishnan, K., Hindmarsh, A.C. 1993 *Description and Use of LSODE: the Livermore Solver for Ordinary Differential Equations*, Lawerence Livermore National Laboratory Report UCRL-ID-113855 or NASA Reference Publication 1327.

[4] Stoer, J., Bulirsch, R. 1980 *Introduction to Numerical Analysis*, Springer-Verlag, New York.

[5] Stroustrup, B., 1991 *The C++ Programming Language*, Addison-Wesley, Reading, Mass.