C++ programming for Animal Breeding

R. L. Fernando
Department of Animal Science
Iowa State University

S. D. Kachman
Department of Statistics
University of Nebraska–Lincoln

Contents

Chapter 1

Classes for MME

C++ classes for building and solving mixed model equations (MME) are described here. Initially, the classes will be developed for building the normal equations for a univariate fixed linear model. Also, in the initial versions of the classes, the model will be described in terms of a vector of model terms. Subsequently, the classes will be extended to accommodate more complex models. Further, in subsequent versions of the MME classes the model will be described in terms of a model string such as:

```
"y1 = intercept breed directAdditive; \
 y2 = intercept breed directAdditive;"
```

1.1 Strategy for building normal equations

Recall that the contributions to the LHS of the normal equations from observation k can be written as

$$\boldsymbol{x}_k\boldsymbol{x}'_k, \tag{1.1}$$

where $\boldsymbol{x}'_k$ is row k of the incidence matrix. We have seen that given the positions and values of these non-zero elements, the contributions to LHS can be computed efficiently.

1.1.1 Contributions to LHS

- For observation k, for each combination of model terms i and j, add $v_{ki}v_{kj}$ to element $(\text{pos}_{ki},\text{pos}_{kj})$ of the LHS

- pos_{ki} is the position of the non-zero entry in $\boldsymbol{x}_k$ for model term i:

$$\text{pos}_{ki} = \text{start}_i + \text{level}_{ki} - 1$$

- pos_{kj} is the position of the non-zero entry in $\boldsymbol{x}_k$ for model term j:

$$\text{pos}_{kj} = \text{start}_j + \text{level}_{kj} - 1$$

- v_{ki} is the non-zero value at position pos_{ki}
- v_{kj} is the non-zero value at position pos_{kj}
- start_i is the starting position of entries for model term i, and level_{ki} the level of this term for observation k.
- start_j is the starting position of entries for model term j, and level_{kj} the level of this term for observation k.

1.1.2 Contributions to RHS

- For observation k, for each model term i, add $v_{ki}y_{ijk}$ to element (pos_{ki}) of the RHS.

1.2 The MME classes

The primary class for building and solving the mixed model equations is called the MME class and its declaration is given below:

```
class MME {
public:
        string fileName;
        Tokenizer colType;
        Tokenizer colName;
        Tokenizer colData;
        unsigned numCols;
        unsigned depCol;
        vector <ModelTerm> modelTrmVec;
        vector <DataNode> dataVec;
        unsigned numTerms;
```

```
65          unsigned mmeSize;
66          matvec::doubleMatrix lhs;
67          matvec::Vector<double> rhs, sol;
68
69          void putColNames(string str);
70          void putColTypes(string str);
71          void inputData();
72          void displayData();
73          static double getDouble(string& Str);
74          void calcStarts();
75          void getSolution();
76          void calcWPW();
77          void display();
78  };
```

The line numbers in this listing and subsequent listings are from the complete listing of file "twowayMME.cpp" given in section 1.2.9 at the end of this chapter. Line 62 of this file shows that modelTrmVec is declared as a vector of ModelTerm objects. As described later, in more detail, this vector is used to store model term objects that describe each term in the model. In the declaration of the ModelTerm class given here,

```
33  class ModelTerm{
34  public:
35          unsigned start;
36          string name;
37          static MME *myMMEPtr;
38          Recoder<string>* myRecoderPtr;
39          vector<unsigned> factors;
40
41          ModelTerm(void){
42                  myRecoderPtr = new Recoder<string>;
43          }
44          unsigned code(string str){return myRecoderPtr->code(str);}
45          unsigned nLevels(){return myRecoderPtr->size();}
46          string   getTermString();
47          unsigned getTermLevel (){
48                  return code(getTermString());
49          }
```

```
        double   getTermValue ();
};
```

"start", is declared as an unsigned integer and is used to store the "start" position for "this" model term. In the declaration of MME, line 63 shows that dataVec is declared as a vector of DataNode objects, and DataNode contains a vector, trmVec, of TermData objects. The declarations for DataNode and TermData are given here:

```
class TermData{
public:
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  double depVar;
};
```

Given these data structures, within methods of the MME class, we access the "start" position for model term i as

```
 modelTrmVec[i].start
```

the "level" and "value" of the non-zero entry for model term i in $\boldsymbol{x}_k$ as

```
dataVec[k].trmVec[i].level
dataVec[k].trmVec[i].value
```

and the value of y_{ijk} as

```
dataVec[k].depVar
```

Once the "start" position is calculated for each element of modelTrmVec, and "level" and "value" are calculated for each element of trmVec within each element of dataVec, the LHS and RHS are calculated efficiently in method "calcWPW" of class MME.

1.2.1 Method MME::calcWPW

The implementation of this method is given below.

```
180 void MME::calcWPW(){
181         unsigned poski,poskj;
182         double vki,vkj,tr_value;
183         rhs.resize(mmeSize,0.0);
184         lhs.resize(mmeSize,mmeSize,0.0);
185         for (unsigned k=0;k<dataVec.size();k++){
186                 for (unsigned i=0;i<numTerms;i++){
187                         poski = modelTrmVec[i].start
188                               + dataVec[k].trmVec[i].level - 1;
189                         vki = dataVec[k].trmVec[i].value;
190                         tr_value = dataVec[k].depVar;
191                         rhs[poski] += vki*tr_value;
192                         for (unsigned j=0;j<numTerms;j++){
193                                 poskj = modelTrmVec[j].start
194                                       + dataVec[k].trmVec[j].level - 1;
195                                 vkj = dataVec[k].trmVec[j].value;
196                                 lhs[poski][poskj] += vki*vkj;
197                         }
198                 }
199         }
200 }
```

The "for loop" starting on line 185 handles each element in dataVec. Within this loops, the nested "for loops" starting on lines 186 and on 192 go through all combinations of model terms i and j, making the appropriate contributions to the RHS (line 191) and LHS (line 196). Thus, this method is general enough to calculate the LHS and RHS of the normal equations for any univariate fixed linear model without any modifications. Next, we will examine how the "level" and "value" members of each element of trmVec are calculate for each element of dataVec, which is a prerequisite for this method. These are calculated in the MME:ïnputData. However, before we examine this method, it is useful to look at the "main program", where we describe the model in terms of ModelTerm objects.

1.2.2 The main program

This main program uses the MME class to build the normal equations for a two-way fixed linear model with factors A and B. For simplicity, we have not fitted an intercept, and have fitted main effects for factors A and B and an interaction between A and B.

After declaring "mme" as an object of the MME class (line 242), we store its address in the static member, "myMMEPtr", of the ModelTerm class, which was declared as a pointer to a MME class object. Now, the address that is stored in "myMMEPtr" is available to all objects of ModelTerm class. Further, this address of "mme" can be accessed from any method as "ModelTerm::myMMEPtr".

Now, we describe the model in terms of model term objects. The first effect or "model term" is for the main effect of factor A. On line 248 we create a ModelTerm object called mtermA that is used to represent the main effect of A. An important member of the ModelTerm class is factors, which was declared as a vector of unsigned integers. This vector is used to store the column indices of the factors that define the model term. As mtermA represents a main effect we resize factors to have a size of one (line 250). In the data file "twoway.dat", factor A is given on the first column. Thus, the index of zero is stored in the first and only element of mtermA.factors (line 251). Finally, we put mtermA into the ModelTrmVec (line 252) of mme, which we have declared as an MME object (line 242).

Similarly, mtermB is used to represent the model term for the main effect of factor B. The primary difference between mtermA and mtermB is that the index that is stored in factors— in mtermB.factors we store the index 1 because the factor B is given in the second column of the data file.

```
239 int main() {
240         try{
241                 matvec::SESSION.initialize("matvec_trash");
242                 MME mme;
243                 ModelTerm::myMMEPtr = &mme;
244                 mme.fileName = "Data/twoway.dat";
245                 mme.putColNames("A B y");
246                 mme.putColTypes("CLASS CLASS DEP");
247
248                 ModelTerm mtermA;
249                 mtermA.name = "A";
```

```
250                 mtermA.factors.resize(1);
251                 mtermA.factors[0] = 0;
252                 mme.modelTrmVec.push_back(mtermA);
253 
254                 ModelTerm mtermB;
255                 mtermB.name = "B";
256                 mtermB.factors.resize(1);
257                 mtermB.factors[0] = 1;
258                 mme.modelTrmVec.push_back(mtermB);
259 
260                 ModelTerm mtermAB;
261                 mtermAB.name = "A*B";
262                 mtermAB.factors.resize(2);
263                 mtermAB.factors[0] = 0;
264                 mtermAB.factors[1] = 1;
265                 mme.modelTrmVec.push_back(mtermAB);
266 
267                 mme.getSolution();
268                 mme.display();
269         }
270         catch (matvec::exception &ex) {
271                 cerr << ex.what() << "\n";
272                 exit(1);
273         }
274         catch (...) {
275                 cerr << "other exceptions were caught\n";
276                 exit(1);
277         }
278 }
```

Finally, we use mtermAB to represent the interaction between A and B. The factors vector in mtermAB is resized to 2 because we have to store the indices for both A and B (line 262), and then these indices are stored in mtermAB.factors (lines 263 and 264). Now, we can look the MME:ïnputData method.

1.2.3 Method MME::inputData

```
138 void MME::inputData(){
139      DataNode dataNode;
140      numTerms = modelTrmVec.size();
141      dataNode.trmVec.resize(numTerms);
142      ifstream datafile;
143      datafile.open(fileName.c_str());
144      if(!datafile) {
145              cerr << "Couldn't open data file: " << fileName << endl;
146              exit (-1);
147      }
148      unsigned linewidth = 1024;
149      char *line = new char [linewidth];
150      string sep(" ");
151      while (datafile.getline(line,linewidth)){
152          string inputStr(line);
153          colData.getTokens(inputStr,sep);
154          dataNode.depVar = getDouble(colData[depCol]);
155          for (unsigned i=0;i<numTerms;i++){
156             dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
157             dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
158          }
159              dataVec.push_back(dataNode);
160      }
161 }
```

Here, line 151 is the start of a "while loop" to process each line of the data file. Each line is initially read into the character array "line" (line 151). In line 152, "line" is converted to a C++ string called inputStr. The colData member of the MME class, which is used in the next line, is declared to be a Tokenizer (line 59). The declaration of the Tokenizer class given below

```
class Tokenizer:public vector<string> {
 public:
  void getTokens(const string &str, const string &sep);
  int  getIndex(string str);
};
```

is from the file "util.h". The first line in this declaration specifies that Tokenizer is "inherited" from a vector of strings. This means that it has all the properties of a vector of strings with the additional methods: getTokens and getIndex that are declared for Tokenizer.

The columns in the data file are separated by spaces, and the statement on line 153 breaks up the inputStr string into substrings that become elements of the colData vector. In line 154, the string that represents the dependent variable is obtained from colData and converted to a double precision real number; it is stored in dataNode.depVar. Line 155 is the start of a "for loop" that goes over all the elements of the modelTrmVec vector. On line 156, the level for model term i is calculated and stored in dataNode.trmVec[i].level, and on line 157, the value for this model term is calculated and stored in dataNode.trmVec[i].value.

To compute the level for a model term, the ModelTerm::getTermString method is used to generate a string that represents that model term given the substrings stored in colData. Then, the Recoder object that belongs to modelTrmVec[i] is used to get the level that corresponds to the generated string. To get the value for a model term, the ModelTerm::getTermValue is used. We will now examine these methods.

1.2.4 Method ModelTerm::getTermString

```
string ModelTerm::getTermString(){
     unsigned numFactors = factors.size();
     string trmStr;
     unsigned factorIndex = factors[0];
     if(myMMEPtr->colType[factorIndex]=="COV"){
        trmStr = myMMEPtr->colName[factorIndex];
     }
     else {
        trmStr = myMMEPtr->colData[factorIndex];
     }
     for (unsigned i=1;i<numFactors;i++){
         factorIndex = factors[i];
         if(myMMEPtr->colType[factorIndex]=="COV"){
            trmStr += "*" + myMMEPtr->colName[factorIndex];
         }
         else{
```

```
                trmStr += "*" + myMMEPtr->colData[factorIndex];
            }
        }
        return trmStr;
}
```

Recall that the data column indices for the factors of a model term are stored in the vector factors. First consider a model without any factors that are covariables. Then, this method uses the indices stored in factors to get the substrings from colData that correspond to the factors of the model. If the model term has more than one factor, the substrings are concatenated together with a "*" separating the substrings.

If a factor in a model term is a covariable, the substring from colData is not used to generate the trmStr string; rather, the name of this factor is used. This is because, for a covariable, the data file contains the value of the covariable rather than an indicator of the "level" of the factor.

1.2.5 Method ModelTerm::getTermValue

```
double ModelTerm::getTermValue(){
    unsigned numFactors = factors.size();
    double value = 1.0;
    for (unsigned i=0;i<numFactors;i++){
       unsigned factorIndex = factors[i];
       if (myMMEPtr->colType[factorIndex]=="COV"){
               string covStr = myMMEPtr->colData[factorIndex];
               value *= MME::getDouble(covStr);
       }
    }
    return value;
}
```

Here, we begin by setting "value" to one. Then, we consider each factor in the ModelTerm; if the factor is a covariable, the substring is converted to a double precision real number say x, and value is set to value*"x".

In the method MME::calcWPW, in addition to the level and value we needed the "start" positions for each model term. These start positions are calculated in MME::calcStarts, which is described next.

1.2.6 Method MME::calcStarts

```
170 void MME::calcStarts(){
171      modelTrmVec[0].start = 0;
172      for (unsigned i=1;i<numTerms;i++){
173          modelTrmVec[i].start = modelTrmVec[i-1].start
174                               + modelTrmVec[i-1].nLevels();
175      }
176      mmeSize = modelTrmVec[numTerms-1].start
177              + modelTrmVec[numTerms-1].nLevels();
178 }
```

Here, the "start" position for the first model term is set to 0. For each subsequent model term, "start" is set to the value of "start" plus the number of levels in the previous term (lines 173 and 174). In this method we also compute the size of the normal equations (lines 176 and 177). The MME::getSolution method is examined next.

1.2.7 Method MME::getSolution

```
 void MME::getSolution(){
        inputData();
        calcStarts();
        calcWPW();
        sol = lhs.ginv0()*rhs;
}
```

When this method is called, the inputData method is called first, then calcStarts is called. Now we have the prerequisite to build the normal equations in calcWPW, which is called next. Finally, the solution is obtained by pre-multiplying the RHS by the generalized inverse of the LHS. Once the solution is obtained, it can be displayed using the MME::display method. This is described next.

1.2.8 Method MME::display

```
void MME::display(){
    cout << "LHS " << endl;
    for (unsigned i = 0;i<mmeSize;i++){
```

```
212         for (unsigned j = 0;j<mmeSize;j++){
213            cout << setw(10)
214                 << setprecision (4)
215                 << setiosflags (ios::right | ios::fixed)
216                 << lhs[i][j] <<" ";
217         }
218         cout << endl;
219       }
220       cout << "RHS " << endl;
221       cout << rhs << endl;
222       for (unsigned i=0;i<modelTrmVec.size();i++){
223         cout << "Solutions for " << modelTrmVec[i].name << endl;
224         Recoder<string>::iterator it;
225         for (it=modelTrmVec[i].myRecoderPtr->begin();
226            it!=modelTrmVec[i].myRecoderPtr->end();
227            it++){
228              unsigned ii = modelTrmVec[i].start + it->second - 1;
229              cout << setw(10) << it->first << " " << sol[ii] << endl;
230         }
231       }
232   }
```

Lines 210 to 221 of this method are for printing the LHS and RHS of the normal equations. The "for loop" starting on line 222 goes through each of the elements of the vector modelTrmVec. For each element of this vector, in line 223 the name of the model term is printed. Then, the "for loop" starting on line 225 goes through each element of the Recoder for this model term. Recall that a Recoder is a map or container where each element contains a pair of objects. The first member of the pair is the key that is used to access the second member of the pair, which is the data stored in the map. In the Recoder of a model term, the model term string generated by ModelTerm::getTermString is used as the key, and the sequential number or level assigned to this string is stored as the data. In line 228, the "start" value for this model term is added to the level stored in the second member of the pair to get the equation number that corresponds to the string stored in the first member of the pair.

1.2.9 Listing of twowayMME.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"

// Single trait, fixed effects models

using namespace std;

class TermData{
public:
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  double depVar;
};

class MME;

class ModelTerm{
public:
    unsigned start;
    string name;
```

```
    static MME *myMMEPtr;
    Recoder<string>* myRecoderPtr;
    vector<unsigned> factors;

    ModelTerm(void){
            myRecoderPtr = new Recoder<string>;
    }

    unsigned code(string str){return myRecoderPtr->code(str);}
    unsigned nLevels(){return myRecoderPtr->size();}
    string   getTermString();
    unsigned getTermLevel (){
            return code(getTermString());
    }
    double   getTermValue ();
};

class MME {
public:
        string fileName;
        Tokenizer colType;
        Tokenizer colName;
        Tokenizer colData;
        unsigned numCols;
        unsigned depCol;
        vector <ModelTerm> modelTrmVec;
        vector <DataNode> dataVec;
        unsigned numTerms;
        unsigned mmeSize;
        matvec::doubleMatrix lhs;
        matvec::Vector<double> rhs, sol;

        void putColNames(string str);
        void putColTypes(string str);
        void inputData();
        void displayData();
        static double getDouble(string& Str);
        void calcStarts();
```

```
        void getSolution();
        void calcWPW();
        void display();
};
string ModelTerm::getTermString(){
     unsigned numFactors = factors.size();
     string trmStr;
     unsigned factorIndex = factors[0];
     if(myMMEPtr->colType[factorIndex]=="COV"){
        trmStr = myMMEPtr->colName[factorIndex];
     }
     else {
        trmStr = myMMEPtr->colData[factorIndex];
     }
     for (unsigned i=1;i<numFactors;i++){
        factorIndex = factors[i];
        if(myMMEPtr->colType[factorIndex]=="COV"){
            trmStr += "*" + myMMEPtr->colName[factorIndex];
        }
        else{
            trmStr += "*" + myMMEPtr->colData[factorIndex];
        }
     }
     return trmStr;
}

double ModelTerm::getTermValue(){
    unsigned numFactors = factors.size();
    double value = 1.0;
    for (unsigned i=0;i<numFactors;i++){
       unsigned factorIndex = factors[i];
       if (myMMEPtr->colType[factorIndex]=="COV"){
          string covStr = myMMEPtr->colData[factorIndex];
          value *= MME::getDouble(covStr);
       }
    }
    return value;
}
```

```

void MME::putColNames(string str){
  string sep(" ");
  colName.getTokens(str,sep);
  numCols = colName.size();
}

void MME::putColTypes(string str){
  string sep(" ");
  colType.getTokens(str,sep);
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
  }
  for (unsigned i=0;i<numCols;i++){
        if (colType[i] == "DEP") {
                depCol = i;
                return;
        }
  }
  cout << "Could not find dependent variable \n";
  exit(1);
}

void MME::inputData(){
     DataNode dataNode;
     numTerms = modelTrmVec.size();
     dataNode.trmVec.resize(numTerms);
     ifstream datafile;
     datafile.open(fileName.c_str());
     if(!datafile) {
             cerr << "Couldn't open data file: " << fileName << endl;
             exit (-1);
     }
     unsigned linewidth = 1024;
     char *line = new char [linewidth];
     string sep(" ");
     while (datafile.getline(line,linewidth)){
```

```
            string inputStr(line);
            colData.getTokens(inputStr,sep);
            dataNode.depVar = getDouble(colData[depCol]);
            for (unsigned i=0;i<numTerms;i++){
               dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
               dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
            }
            dataVec.push_back(dataNode);
         }
}

double MME::getDouble(string& Str) {
  istringstream inputStrStream(Str.c_str());
  double val;
  inputStrStream >> val;
  return val;
}

void MME::calcStarts(){
     modelTrmVec[0].start = 0;
     for (unsigned i=1;i<numTerms;i++){
         modelTrmVec[i].start = modelTrmVec[i-1].start
                              + modelTrmVec[i-1].nLevels();
     }
     mmeSize = modelTrmVec[numTerms-1].start
             + modelTrmVec[numTerms-1].nLevels();
}

void MME::calcWPW(){
    unsigned poski,poskj;
    double vki,vkj,tr_value;
    rhs.resize(mmeSize,0.0);
    lhs.resize(mmeSize,mmeSize,0.0);
    for (unsigned k=0;k<dataVec.size();k++){
       for (unsigned i=0;i<numTerms;i++){
           poski = modelTrmVec[i].start
                 + dataVec[k].trmVec[i].level - 1;
           vki = dataVec[k].trmVec[i].value;
```

```
            tr_value = dataVec[k].depVar;
            rhs[poski] += vki*tr_value;
            for (unsigned j=0;j<numTerms;j++){
               poskj = modelTrmVec[j].start
                     + dataVec[k].trmVec[j].level - 1;
               vkj = dataVec[k].trmVec[j].value;
               lhs[poski][poskj] += vki*vkj;
            }
        }
    }
}

void MME::getSolution(){
        inputData();
        calcStarts();
        calcWPW();
        sol = lhs.ginv0()*rhs;
}

void MME::display(){
    cout << "LHS " << endl;
    for (unsigned i = 0;i<mmeSize;i++){
      for (unsigned j = 0;j<mmeSize;j++){
         cout << setw(10)
              << setprecision (4)
              << setiosflags (ios::right | ios::fixed)
              << lhs[i][j] <<" ";
      }
      cout << endl;
    }
    cout << "RHS " << endl;
    cout << rhs << endl;
    for (unsigned i=0;i<modelTrmVec.size();i++){
       cout << "Solutions for " << modelTrmVec[i].name << endl;
       Recoder<string>::iterator it;
       for (it=modelTrmVec[i].myRecoderPtr->begin();
            it!=modelTrmVec[i].myRecoderPtr->end();
            it++){
```

```
                unsigned ii = modelTrmVec[i].start + it->second - 1;
                cout << setw(10)
                     << it->first << " "
                     << sol[ii] << endl;
        }
    }
}


MME* ModelTerm::myMMEPtr;

int main() {
        try{
                matvec::SESSION.initialize("matvec_trash");
                MME mme;
                ModelTerm::myMMEPtr = &mme;
                mme.fileName = "Data/twoway.dat";
                mme.putColNames("A B y");
                mme.putColTypes("CLASS CLASS DEP");

                ModelTerm mtermA;
                mtermA.name = "A";
                mtermA.factors.resize(1);
                mtermA.factors[0] = 0;
                mme.modelTrmVec.push_back(mtermA);

                ModelTerm mtermB;
                mtermB.name = "B";
                mtermB.factors.resize(1);
                mtermB.factors[0] = 1;
                mme.modelTrmVec.push_back(mtermB);

                ModelTerm mtermAB;
                mtermAB.name = "A*B";
                mtermAB.factors.resize(2);
                mtermAB.factors[0] = 0;
                mtermAB.factors[1] = 1;
                mme.modelTrmVec.push_back(mtermAB);
```

```

                mme.getSolution();
                mme.display();
        }
        catch (matvec::exception &ex) {
                cerr << ex.what() << "\n";
                exit(1);
        }
        catch (...) {
                cerr << "other exceptions were caught\n";
                exit(1);
        }
}
```

1.3 Extensions to accommodate multiple trait mixed models

The complete listing of the program with extensions to accommodate multiple trait mixed models is given in section 1.3.1. Here we describe the changes that were made to extend the program. The program was also extended to automatically create data for the intercept (lines 131, 138 and 169).

In order to store the multiple dependent variables, the DataNode class was modified as shown below, where the dependent variables are stored in a vector depVec.

```
class DataNode{
public:
  vector<TermData > trmVec;
  vector<double>    depVec;
};
```

The following "for loop" was added to the MME::inputData method to read in the dependent variables for each observation:

```
    for (unsigned i=0;i<numCols;i++){
            if (colType[i]=="DEP") {
                    dataNode.depVec[j++] = getDouble(colData[i]);
            }
```

```
    }
```

Further, in the MME class, “R” and “Ri” were declared to be matrices to store the within-observation residual covariance matrix and the inverse of this matrix. In order to compute the LHS and RHS with correlated residuals, an unsigned integer member, “trait”, was added to the ModelTerm class. For each model term, trait is set to the index in “DataNode::depVec” for the dependent variable.

The MME::calcWPW method to compute the LHS and RHS with correlated residuals is:

```
void MME::calcWPW(){
    unsigned ii,jj,ti,tj;
    double vi,vj,tr_value;
    rhs.resize(mmeSize,0.0);
    lhs.resize(mmeSize,mmeSize,0.0);
    Ri = R.inv();
    for (unsigned i=0;i<dataVec.size();i++){
       for (unsigned mi=0;mi<numTerms;mi++){
          ii = modelTrmVec[mi].start
             + dataVec[i].trmVec[mi].level - 1;
          ti =  modelTrmVec[mi].trait;
          vi = dataVec[i].trmVec[mi].value;
          for (unsigned k=0;k<numTraits;k++){
             rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
          }
          for (unsigned mj=0;mj<numTerms;mj++){
            jj = modelTrmVec[mj].start
               + dataVec[i].trmVec[mj].level - 1;
            tj =  modelTrmVec[mj].trait;
            vj = dataVec[i].trmVec[mj].value;
            lhs[ii][jj] += vi*Ri[ti][tj]*vj;
          }
       }
    }
}
```

To accommodate mixed models with correlations between random effects, a new class, “CovBlock”, was declared as shown below:

```
class CovBlock {
public:
        vector<ModelTerm*> modelTrmPtrVec;
        matvec::doubleMatrix Var, Vari;
        Pedigree* pedPtr;
        CovBlock(void){pedPtr = 0;}

        void addGinv(void);
};
```

Here, modelTrmPtrVec is declared as a vector of ModelTerm pointers. Pointers to ModelTerm objects for the correlated random effects are stored in this vector.

Suppose the model includes a set, $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$, of k correlated random effects with

$$\mathrm{Var}(\boldsymbol{u}) = \boldsymbol{\Sigma} = \boldsymbol{G} \otimes \boldsymbol{A}. \tag{1.2}$$

The inverse of this covariance matrix is

$$\boldsymbol{\Sigma}^{-1} = \boldsymbol{G}^{-1} \otimes \boldsymbol{A}^{-1}. \tag{1.3}$$

To obtain the LHS of the MME we need to add $\boldsymbol{\Sigma}^{-1}$ to the LHS of the normal equations at the position corresponding to $\boldsymbol{u}$, In the method CovBlock::addGinv, this is done by adding $\boldsymbol{A}^{-1}g^{ij}$ to the LHS of the normal equations at the position corresponding to random effects $\boldsymbol{u}_i$ and $\boldsymbol{u}_j$, for all combinations of traits i and j, where g^{ij} is element ij of $\boldsymbol{G}^{-1}$. In the listing of this method given below, the "for loops" starting on lines 233 and 236 are for going through all combinations of model terms included in this CovBlock. If $\boldsymbol{A}$ is the additive relationship matrix, the Pedigree::addAinv method is used to add $\boldsymbol{A}^{-1}g^{ij}$ to the normal equations (line 240). On the other hand, if $\boldsymbol{A}$ is the identity matrix, the adding of $\boldsymbol{I}g^{ij}$ is done on lines 244–246.

```
230 void CovBlock::addGinv(void){
231    Vari = Var.inv();
232    unsigned n = modelTrmPtrVec.size();
233    for (unsigned i=0;i<n;i++){
234      ModelTerm* mtermiPtr = modelTrmPtrVec[i];
235      unsigned starti = mtermiPtr->start;
236      for (unsigned j=0;j<n;j++){
```

```
            ModelTerm* mtermjPtr = modelTrmPtrVec[j];
            unsigned startj = mtermjPtr->start;
            if (pedPtr){
               pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
            }
            else{
               unsigned numLevels = mtermiPtr->nLevels();
               for (unsigned k=0;k<numLevels;k++){
                   ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
               }
            }
         }
      }
}
```

1.3.1 Listing of multiTraitMixedMME.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"
#include "ped.h"

// classes for multiple trait, mixed models

using namespace std;

class TermData{
public:
```

```
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  vector<double>    depVec;
};


class MME;
class ModelTerm{
public:
        unsigned start;
        unsigned trait;

        string name;
        static MME *myMMEPtr;
        Recoder<string>* myRecoderPtr;
        vector<unsigned> factors;

        ModelTerm(void){
                myRecoderPtr = new Recoder<string>;
        }
        unsigned code(string str){return myRecoderPtr->code(str);}
        unsigned nLevels(){return myRecoderPtr->size();}
        string   getTermString();
        unsigned getTermLevel (){
                return code(getTermString());
        }
        double   getTermValue ();
};

class CovBlock {
public:
        vector<ModelTerm*> modelTrmPtrVec;
        matvec::doubleMatrix Var, Vari;
```

```
        Pedigree* pedPtr;
        CovBlock(void){pedPtr = 0;}

        void addGinv(void);
};


class MME {
public:
        string fileName;
        Tokenizer colType;
        Tokenizer colName;
        Tokenizer colData;
        unsigned numCols;
        unsigned depCol;
        vector <ModelTerm> modelTrmVec;
        vector <CovBlock>  covBlockVec;
        vector <DataNode> dataVec;
        unsigned numTerms, numTraits;
        unsigned mmeSize;

        matvec::doubleMatrix lhs, R, Ri;
        matvec::Vector<double> rhs, sol;

        void putColNames(string str);
        void putColTypes(string str);
        void inputData();
        void displayData();
        static double getDouble(string& Str);
        void calcStarts();
        void getSolution();
        void calcWPW();
        void addGinv();
        void display();
};

string ModelTerm::getTermString(){
     unsigned numFactors = factors.size();
```

```
     string trmStr;
     unsigned factorIndex = factors[0];
     if(myMMEPtr->colType[factorIndex]=="COV"){
        trmStr = myMMEPtr->colName[factorIndex];
     }
     else {
        trmStr = myMMEPtr->colData[factorIndex];
     }
     for (unsigned i=1;i<numFactors;i++){
        factorIndex = factors[i];
        if(myMMEPtr->colType[factorIndex]=="COV"){
           trmStr += "*" + myMMEPtr->colName[factorIndex];
        }
        else{
           trmStr += "*" + myMMEPtr->colData[factorIndex];
        }
     }
     return trmStr;
}

double ModelTerm::getTermValue(){
    unsigned numFactors = factors.size();
    double value = 1.0;
    for (unsigned i=0;i<numFactors;i++){
       unsigned factorIndex = factors[i];
       if (myMMEPtr->colType[factorIndex]=="COV"){
         string covStr = myMMEPtr->colData[factorIndex];
         value *= MME::getDouble(covStr);
       }
    }
    return value;
}

void MME::putColNames(string str){
  string sep(" ");
  str = "intercept " + str;
  colName.getTokens(str,sep);
  numCols = colName.size();
```

```
}

void MME::putColTypes(string str){
  string sep(" ");
  str = "CLASS " + str;
  colType.getTokens(str,sep);
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
  }
  unsigned n = 0;
  for (unsigned i=0;i<numCols;i++){
        if (colType[i] == "DEP") {
                n++;
        }
  }
  numTraits = n;
}

void MME::inputData(){
    DataNode dataNode;
    numTerms = modelTrmVec.size();
    dataNode.trmVec.resize(numTerms);
    dataNode.depVec.resize(numTraits);
    ifstream datafile;
    datafile.open(fileName.c_str());
    if(!datafile) {
       cerr << "Couldn't open data file: " << fileName << endl;
       exit (-1);
    }
    unsigned linewidth = 1024;
    char *line = new char [linewidth];
    string sep(" ");
    while (datafile.getline(line,linewidth)){
      string inputStr(line);
      inputStr = "--- " + inputStr;
      colData.getTokens(inputStr,sep);
      unsigned j=0;
```

```
      for (unsigned i=0;i<numCols;i++){
        if (colType[i]=="DEP") {
           dataNode.depVec[j++] = getDouble(colData[i]);
        }

      }
      for (unsigned i=0;i<numTerms;i++){
        dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
        dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
      }
      dataVec.push_back(dataNode);
    }
}

double MME::getDouble(string& Str) {
  istringstream inputStrStream(Str.c_str());
  double val;
  inputStrStream >> val;
  return val;
}


void MME::calcStarts(){
     modelTrmVec[0].start = 0;
     for (unsigned i=1;i<numTerms;i++){
       modelTrmVec[i].start = modelTrmVec[i-1].start
                            + modelTrmVec[i-1].nLevels();
     }
     mmeSize = modelTrmVec[numTerms-1].start
             + modelTrmVec[numTerms-1].nLevels();
}

void MME::calcWPW(){
    unsigned ii,jj,ti,tj;
    double vi,vj,tr_value;
    rhs.resize(mmeSize,0.0);
    lhs.resize(mmeSize,mmeSize,0.0);
    Ri = R.inv();
```

```
    for (unsigned i=0;i<dataVec.size();i++){
       for (unsigned mi=0;mi<numTerms;mi++){
          ii = modelTrmVec[mi].start
             + dataVec[i].trmVec[mi].level - 1;
          ti =  modelTrmVec[mi].trait;
          vi = dataVec[i].trmVec[mi].value;
          for (unsigned k=0;k<numTraits;k++){
             rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
          }
          for (unsigned mj=0;mj<numTerms;mj++){
             jj = modelTrmVec[mj].start
                + dataVec[i].trmVec[mj].level - 1;
             tj =  modelTrmVec[mj].trait;
             vj = dataVec[i].trmVec[mj].value;
             lhs[ii][jj] += vi*Ri[ti][tj]*vj;
          }
       }
    }
}

void MME::addGinv(){
        for (unsigned i=0;i<covBlockVec.size();i++){
                covBlockVec[i].addGinv();
        }
}

void MME::getSolution(){
        inputData();
        calcStarts();
        calcWPW();
        addGinv();
        sol = lhs.ginv0()*rhs;
}

void MME::display(){
     cout << "LHS " << endl;
     for (unsigned i = 0;i<mmeSize;i++){
     for (unsigned j = 0;j<mmeSize;j++){
```

```
        cout << setw(10)
        << setprecision (4)
        << setiosflags (ios::right | ios::fixed)
        << lhs[i][j] <<" ";
        }
        cout << endl;
      }
      cout << "RHS " << endl;
      cout << rhs << endl;
      for (unsigned i=0;i<modelTrmVec.size();i++){
         cout << "Solutions for " << modelTrmVec[i].name
              << ", Trait " << modelTrmVec[i].trait+1 << endl;
         Recoder<string>::iterator it;
         for (it=modelTrmVec[i].myRecoderPtr->begin();
              it!=modelTrmVec[i].myRecoderPtr->end();it++){
           unsigned ii = modelTrmVec[i].start + it->second - 1;
           cout << setw(10) << it->first << " " << sol[ii] << endl;
         }
     }
}

void CovBlock::addGinv(void){
   Vari = Var.inv();
   unsigned n = modelTrmPtrVec.size();
   for (unsigned i=0;i<n;i++){
     ModelTerm* mtermiPtr = modelTrmPtrVec[i];
     unsigned starti = mtermiPtr->start;
     for (unsigned j=0;j<n;j++){
        ModelTerm* mtermjPtr = modelTrmPtrVec[j];
        unsigned startj = mtermjPtr->start;
        if (pedPtr){
         pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
        }
        else{
           unsigned numLevels = mtermiPtr->nLevels();
           for (unsigned k=0;k<numLevels;k++){
               ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
           }
```

```
        }
      }
    }
}


MME* ModelTerm::myMMEPtr;

// model abstraction using MME class: two-trait, animal model


int main() {
        try{
                matvec::SESSION.initialize("matvec_trash");
                Pedigree ped;
                ped.inputPed("Data/additive.ped");
                MME mme;
                matvec::doubleMatrix R;
                R.resize(2,2);
                R(1,1) = 1.0;
                R(1,2) = R(2,1) = 0.5;
                R(2,2) = 2.0;
                mme.R = R;
                ModelTerm::myMMEPtr = &mme;
                mme.fileName = "Data/additive2Tr.dat";
                mme.putColNames("direct y1  y2");
                mme.putColTypes("CLASS  DEP DEP");

                ModelTerm mterm0;
                mterm0.trait = 0;
                mterm0.name = "intercept";
                mterm0.factors.resize(1);
                mterm0.factors[0] = 0;// column 0 has been added for intercept
                mme.modelTrmVec.push_back(mterm0);

                ModelTerm mterm1;
                mterm1.trait = 0;
                mterm1.name = "directAdditive";
```

```
                mterm1.factors.resize(1);
                mterm1.factors[0] = 1;
                delete mterm1.myRecoderPtr;
                mterm1.myRecoderPtr = &ped.coder;
                mme.modelTrmVec.push_back(mterm1);

                ModelTerm mterm2;
                mterm2.trait = 1;
                mterm2.name = "intercept";
                mterm2.factors.resize(1);
                mterm2.factors[0] = 0;
                mme.modelTrmVec.push_back(mterm2);

                ModelTerm mterm3;
                mterm3.trait = 1;
                mterm3.name = "directAdditive";
                mterm3.factors.resize(1);
                mterm3.factors[0] = 1;
                delete mterm3.myRecoderPtr;
                mterm3.myRecoderPtr = &ped.coder;
                mme.modelTrmVec.push_back(mterm3);

                CovBlock covBlock;
                covBlock.Var = R;
                covBlock.pedPtr = &ped;
                covBlock.modelTrmPtrVec.push_back(&mme.modelTrmVec[1]);
                covBlock.modelTrmPtrVec.push_back(&mme.modelTrmVec[3]);
                mme.covBlockVec.push_back(covBlock);
                mme.getSolution();
                mme.display();
        }
        catch (matvec::exception &ex) {
                cerr << ex.what() << "\n";
                exit(1);
        }
        catch (...) {
                cerr << "other exceptions were caught\n";
                exit(1);
```

```
        }
}
```

Chapter 2

MME from Model String

Here, we will see how the ModelTerm objects that were used to describe the model in the main program of the previous chapter can be generated from a model string by using a call to the MME::putModels method. Further, to accommodate correlated random effects, a CovBlock object was used, with a vector of pointers to the ModelTerms of the correlated set of random effects (lines 349–350 of listing in section 1.3.1). Here, these statements will be replaced by a single call to the method MME::putVarCovMatrix for each set of correlated random effects.

2.1 Generating ModelTerm objects

We will now examine in more detail how the ModelTerm objects are generated from the model string. We begin by examining the MME::putModels method, which is called from the main program.

2.1.1 Method MME::putModels

```
void MME::putModels(string str){
  Tokenizer models;
  string sep =";";
  models.getTokens(str,sep);
  for (unsigned i=0; i<models.size();i++){
    putModel(models[i]);
  }
```

```
201 }
```

The getTokens method on line 197 breaks up the "str" string at the semicolons. Then, element i of the Tokenizer object "models" will contain the sub-model for trait i. Each of these sub-models is then processed by calling the putModel method on line 199.

2.1.2 Method MME::putModel

```
203 void MME::putModel(string str){
204    string sep(" =+");
205    Tokenizer modelTokens;
206    modelTokens.getTokens(str,sep);
207    unsigned nTokens = modelTokens.size();
208    int depVarIndex = colName.getIndex(modelTokens[0]);
209    if (depVarIndex == -1){
210      cerr << "Dependent Variable "
211           << modelTokens[0]
212           << " not in list of column names \n";
213      exit (-1);
214    }
215    ModelTerm modelTrm;
216    modelTrm.depVarName = modelTokens[0];
217    modelTrm.trait = depVar.getIndex(modelTokens[0]);
218    for (unsigned i=1;i<nTokens;i++){
219            modelTrm.myRecoderPtr = new Recoder<string>;
220            modelTrm.name = modelTokens[i];
221            modelTrm.putFactors(modelTrm.name);
222            modelTrmVec.push_back(modelTrm);
223    }
224 }
```

The getTokens method on line 206 breaks up the "str" string at "blank", "=" or "+" characters. Now, modelTokens[0] should contain the dependent variable of the model. Recall that "colNames" is a Tokenizer object that contains the names of the columns in the data file. The statements on lines 208–214 are for checking if the dependent variable matches one of the names in "colNames".

On line 215, modelTrm is declared to be a ModelTerm object. On the next line, the name of the dependent variable is stored in modelTrm.depVarName. This name will be used in displaying solutions to the MME. The index of the dependent variable in the list of dependent variables is obtained next by using the getIndex method of the Tokenizer object “depVar”. This index is stored in modelTerm.trait. These two members will be constant for all the terms of a sub-model. Now, the “for loop” starting on line 218 sets the remaining members of modelTerm for each term in the model and saves a copy in the vector, modelTrmVec. On line 219, a string Recoder is created and its address is stored in modelTrm.myRecoderPtr. On the next line, the string that represents the model term is stored in modelTrm.name. This name is also used in displaying solutions. On line 221, the putFactors method of ModelTerm object “modelTrm” is used to set the factors vector of “modelTrm” from the string that represents that model term.

2.1.3 Method ModelTerm::putFactors

```
114   void ModelTerm::putFactors(string str){
115   Tokenizer tokens;
116   string sep("*");
117   tokens.getTokens(str,sep);
118   factors.clear();
119   for (unsigned i=0;i<tokens.size();i++){
120     unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
121     if (factorIndex == -1){
122       cerr <<"Independent Variable "
123            << tokens[i]
124            << " not in list of column names \n";
125       exit(-1);
126     }
127     else {
128       factors.push_back(factorIndex);
129     }
130   }
131 }
```

On line 117, the getTokens method of Tokenizer object “tokens” is used to break up the string “str” at the “*” character and store the resulting factor names in "tokens". Then, within the "for loop" starting at 119 the index in "colNames" for each of these factor names is obtained on line 120. After checking if the factor name is one of the names in "colNames" (lines 121–126), the index for the factor is stored in the factors vector (line 128).

2.2 Generating CovBlock objects

The MME class has two versions of the method putVarCovMatrix. The implementations of these methods are included in the declaration of the MME class. The first version (lines 96–99) is for a set of correlated additive effects, where the $\boldsymbol{A}$ matrix in (1.2) is the additive relationship matrix. The second version is to be used when $\boldsymbol{A}$ is the identity matrix. In object oriented programming, declaring methods with the same name but with different arguments is called "function overloading".

2.2.1 MME::putVarCovMatrix methods

In the first version of putVarCovMatrix, the pedigree object is given as an argument in the constructor for CovBlock (line 97). This version of the constructor (lines 64–68) stores the address of the pedigree object in pedPtr and then calls its CovBlock::buildModelTrmVec method. The second version of putVarCovMatrix is identical to the first, except that a different constructor (lines 60–63) is used to create the CovBlock object "covBlock" (line 101). Here, pedPtr is set to be "null pointer".

```
96   void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
97     CovBlock covBlock(str,V,P);
98     covBlockVec.push_back(covBlock);
99  }
100 void putVarCovMatrix(string str, matvec::doubleMatrix V){
101   CovBlock covBlock(str,V);
102   covBlockVec.push_back(covBlock);
103 }
```

In both constructors, the call to CovBlock::buildModelTrmVec stores pointers to the ModelTerm objects for the correlated random effects in the "modelTrmVec" vector of the CovBlock object. This method is examined next.

2.2.2 Method CovBlock::buildModelTrmVec

This method is called with a string argument “str” which contains the model term names for the correlated random effects separated by blank spaces. The call to getTokens method (line 348) of the Tokenizer object “modelTokens” breaks up the entries in “str” at blank spaces and stores each model term name in “modelTokens”. The “for loop” starting on line 351 goes through all the elements in “modelTokens”. For each of these elements (model term names), the “for loop” starting on line 352 goes through all the elements in “modelTrmVec” of the current MME object. If the name of the model term object stored in “modelTrmVec[j]” matches the name stored in “modelTokens[i]”, the address of that model term object is stored in the modelTrmPtrVec (lines 353–354).

```
345 void CovBlock::buildModelTrmVec(string str){
346    string sep(" ");
347    Tokenizer modelTokens;
348    modelTokens.getTokens(str,sep);
349    unsigned nTokens = modelTokens.size();
350    unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
351    for (unsigned i=0;i<nTokens;i++){
352      for (unsigned j=0;j<numModelTrms;j++){

353        if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
354           modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
355           if (pedPtr){
356              delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
357              ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr
358                = &pedPtr->coder;
359           }
360        }
361      }
362    }
363 }
```

Line 355 checks if “pedPtr” is a null pointer. If it is not a null pointer, this model term is for a additive effect. Thus, the Recoder from the pedigree should be used for this model term. This switching is done on lines 356–358.

2.3 The main program

Following is the main program from 2.4 for a two-trait additive genetic model. In this main program, lines 393–408 are identical to lines 297–312 in the main program from 1.3.1.

```
393 int main() {
394      try{
395          matvec::SESSION.initialize("matvec_trash");
396          Pedigree ped;
397          ped.inputPed("Data/additive.ped");
398          MME mme;
399          matvec::doubleMatrix R;
400          R.resize(2,2);
401          R(1,1) = 1.0;
402          R(1,2) = R(2,1) = 0.5;
403          R(2,2) = 2.0;
404          mme.R = R;
405          ModelTerm::myMMEPtr = &mme;
406          mme.fileName = "Data/additive2Tr.dat";
407          mme.putColNames("directAdditive y1  y2");
408          mme.putColTypes("CLASS  DEP DEP");

409          mme.putModels("y1 = intercept directAdditive; \
410                         y2 = intercept directAdditive;");

411          mme.putVarCovMatrix("directAdditive",R,ped);
412          mme.getSolution();
413          mme.display();
414      }
415      catch (matvec::exception &ex) {
416          cerr << ex.what() << "\n";
417          exit(1);
418      }
419      catch (...) {
420          cerr << "other exceptions were caught\n";
421          exit(1);
422      }
423 }
```

Line 409 of the current main program gives the model. This is a two-trait model with two model terms for each trait, giving a total to four model terms. In the previous main program, these four model terms were defined and stored in the "mme.modelTrmVec" by the statements in lines 314–344 given below.

```
314                 ModelTerm mterm0;
315                 mterm0.trait = 0;
316                 mterm0.name = "intercept";
317                 mterm0.factors.resize(1);
318                 mterm0.factors[0] = 0;// column 0 has been added for intercept
319                 mme.modelTrmVec.push_back(mterm0);
320
321                 ModelTerm mterm1;
322                 mterm1.trait = 0;
323                 mterm1.name = "directAdditive";
324                 mterm1.factors.resize(1);
325                 mterm1.factors[0] = 1;
326                 delete mterm1.myRecoderPtr;
327                 mterm1.myRecoderPtr = &ped.coder;
328                 mme.modelTrmVec.push_back(mterm1);
329
330                 ModelTerm mterm2;
331                 mterm2.trait = 1;
332                 mterm2.name = "intercept";
333                 mterm2.factors.resize(1);
334                 mterm2.factors[0] = 0;
335                 mme.modelTrmVec.push_back(mterm2);
336
337                 ModelTerm mterm3;
338                 mterm3.trait = 1;
339                 mterm3.name = "directAdditive";
340                 mterm3.factors.resize(1);
341                 mterm3.factors[0] = 1;
342                 delete mterm3.myRecoderPtr;
343                 mterm3.myRecoderPtr = &ped.coder;
344                 mme.modelTrmVec.push_back(mterm3);
```

Here, the call to putModels method on line 409 generates these model four model terms and stores them in "mme.modelTrmVec" as described in 2.1. Among these model terms, the second is for the additive effect for trait 1 and the fourth is for the additive effect for trait 2. Thus, these two model terms represent two random effects that are correlated. As shown below, in the previous main program, a CovBlock object was used to accommodate these correlated random effects in the MME.

```
346             CovBlock covBlock;
347             covBlock.Var = R;
348             covBlock.pedPtr = &ped;
349             covBlock.modelTrmPtrVec.push_back(&mme.modelTrmVec[1]);
350             covBlock.modelTrmPtrVec.push_back(&mme.modelTrmVec[3]);
351             mme.covBlockVec.push_back(covBlock);
```

On line 346, "covBlock" is declared as a CovBlock object, on line 347 the 2×2 covariance matrix between the two additive effects is stored in "covBlock.Var", and on line 348 the memory address of the Pedigree object "ped" is stored in "covBlock.pedPtr". In the next two lines (349–350), the memory addresses of the second and fourth model terms are stored in "covBlock.modelTrmPtrVec". Finally, the "covBlock" object is stored in "mme.covBlockVec". In the current version of the main program, these statements are replaced by a call to the putVarCovMatrix method on line 411. As described in 2.2.1 this method creates a CovBlock object "covBlock" and then stores the matrix "R" in "covBlock.Var" and the address of "ped" in "covBlock.pedPtr". Then, the "covBlock.buildModelTrmVec" is called with the string "directAdditive". As described in 2.2.2, on line 353 of the method CovBlock::buildModelTrmVec, the string "directAdditive" is compared with the names of the model term objects stored in "mme.modelTrmVec". The names of the second and fourth model terms match "directAdditive". Thus the addresses of these model terms will be stored in "covBlock.modelTrmPtrVec".

Finally, the last line of the MME::putVarCovMatrix method stores the CovBlock object "covBlock" in the "covBlockVec" vector of the "mme" object.

2.4 Listing of additive2TrMSMME.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"
#include "ped.h"

// classes for multiple trait, mixed models

using namespace std;

class TermData{
public:
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  vector<double>    depVec;
};

class MME;

class ModelTerm{
public:
    unsigned start;
```

```
    unsigned trait;
    string name;
    string depVarName;
    static MME *myMMEPtr;
    Recoder<string>* myRecoderPtr;
    vector<unsigned> factors;

    unsigned code(string str){return myRecoderPtr->code(str);}
    unsigned nLevels(){return myRecoderPtr->size();}
    void putFactors(string str);
    string   getTermString();
    unsigned getTermLevel (){
            return code(getTermString());
    }
    double   getTermValue ();
};

class CovBlock {
public:
   vector<ModelTerm*> modelTrmPtrVec;
   matvec::doubleMatrix Var, Vari;
   Pedigree* pedPtr;
   CovBlock(void){pedPtr = 0;}

   CovBlock(string str, matvec::doubleMatrix V){
      Var = V; pedPtr = 0;
      buildModelTrmVec(str);
   }

   CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
      Var = V;
      pedPtr = &P;
      buildModelTrmVec(str);
   }
   void buildModelTrmVec(string str);
   void addGinv(void);
};

```

```

class MME {
 private:
  void putModel(string str);
public:
     string fileName;
     Tokenizer colType;
     Tokenizer colName;
     Tokenizer depVar;
     Tokenizer colData;
     unsigned numCols;
     unsigned depCol;
     vector <ModelTerm> modelTrmVec;
     vector <CovBlock>  covBlockVec;
     vector <DataNode> dataVec;
     unsigned numTerms, numTraits;
     unsigned mmeSize;
     matvec::doubleMatrix lhs, R, Ri;
     matvec::Vector<double> rhs, sol;

     void putColNames(string str);
     void putColTypes(string str);
     void putModels(string str);
     void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
       CovBlock covBlock(str,V,P);
       covBlockVec.push_back(covBlock);
     }
     void putVarCovMatrix(string str, matvec::doubleMatrix V){
       CovBlock covBlock(str,V);
       covBlockVec.push_back(covBlock);
     }
     void inputData();
     void displayData();
     static double getDouble(string& Str);
     void calcStarts();
     void getSolution();
     void calcWPW();
     void addGinv();
```

```
     void display();
};

void ModelTerm::putFactors(string str){
  Tokenizer tokens;
  string sep("*");
  tokens.getTokens(str,sep);
  factors.clear();
  for (unsigned i=0;i<tokens.size();i++){
    unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
    if (factorIndex == -1){
      cerr <<"Independent Variable "
           << tokens[i]
           << " not in list of column names \n";
      exit(-1);
    }
    else {
      factors.push_back(factorIndex);
    }
  }
}


string ModelTerm::getTermString(){
    unsigned numFactors = factors.size();
    string trmStr;
    unsigned factorIndex = factors[0];
    if(myMMEPtr->colType[factorIndex]=="COV"){
        trmStr = myMMEPtr->colName[factorIndex];
    }
    else {
        trmStr = myMMEPtr->colData[factorIndex];
    }
    for (unsigned i=1;i<numFactors;i++){
      factorIndex = factors[i];
      if(myMMEPtr->colType[factorIndex]=="COV"){
         trmStr += "*" + myMMEPtr->colName[factorIndex];
      }
```

```
      else{
         trmStr += "*" + myMMEPtr->colData[factorIndex];
      }
    }
    return trmStr;
}

double ModelTerm::getTermValue(){
   unsigned numFactors = factors.size();
   double value = 1.0;
   for (unsigned i=0;i<numFactors;i++){
     unsigned factorIndex = factors[i];
     if (myMMEPtr->colType[factorIndex]=="COV"){
        string covStr = myMMEPtr->colData[factorIndex];
        value *= MME::getDouble(covStr);
     }
   }
   return value;
}

void MME::putColNames(string str){
  string sep(" ");
  str = "intercept " + str;
  colName.getTokens(str,sep);
  numCols = colName.size();
}

void MME::putColTypes(string str){
  string sep(" ");
  str = "CLASS " + str;
  colType.getTokens(str,sep);
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
  }
  unsigned n = 0;
  for (unsigned i=0;i<numCols;i++){
    if (colType[i] == "DEP") {
```

```
        depVar.push_back(colName[i]);
        n++;
      }
    }
  numTraits = n;
}

void MME::putModels(string str){
  Tokenizer models;
  string sep =";";
  models.getTokens(str,sep);
  for (unsigned i=0; i<models.size();i++){
    putModel(models[i]);
  }
}

void MME::putModel(string str){
   string sep(" =+");
   Tokenizer modelTokens;
   modelTokens.getTokens(str,sep);
   unsigned nTokens = modelTokens.size();
   int depVarIndex = colName.getIndex(modelTokens[0]);
   if (depVarIndex == -1){
     cerr << "Dependent Variable "
          << modelTokens[0]
          << " not in list of column names \n";
     exit (-1);
   }
   ModelTerm modelTrm;
   modelTrm.depVarName = modelTokens[0];
   modelTrm.trait = depVar.getIndex(modelTokens[0]);
   for (unsigned i=1;i<nTokens;i++){
           modelTrm.myRecoderPtr = new Recoder<string>;
           modelTrm.name = modelTokens[i];
           modelTrm.putFactors(modelTrm.name);
           modelTrmVec.push_back(modelTrm);
   }
}
```

```

void MME::inputData(){
     DataNode dataNode;
     numTerms = modelTrmVec.size();
     dataNode.trmVec.resize(numTerms);
     dataNode.depVec.resize(numTraits);
     ifstream datafile;
     datafile.open(fileName.c_str());
     if(!datafile) {
        cerr << "Couldn't open data file: " << fileName << endl;
        exit (-1);
     }
     unsigned linewidth = 1024;
     char *line = new char [linewidth];
     string sep(" ");
     while (datafile.getline(line,linewidth)){
       string inputStr(line);
       inputStr = "--- " + inputStr;
       colData.getTokens(inputStr,sep);
       unsigned j=0;
       for (unsigned i=0;i<numCols;i++){
         if (colType[i]=="DEP") {
                 dataNode.depVec[j++] = getDouble(colData[i]);
         }

       }
       for (unsigned i=0;i<numTerms;i++){
          dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
          dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
       }
       dataVec.push_back(dataNode);
    }
}

double MME::getDouble(string& Str) {
  istringstream inputStrStream(Str.c_str());
  double val;
  inputStrStream >> val;
```

```
  return val;
}

void MME::calcStarts(){
     modelTrmVec[0].start = 0;
     for (unsigned i=1;i<numTerms;i++){
             modelTrmVec[i].start = modelTrmVec[i-1].start
                                  + modelTrmVec[i-1].nLevels();
     }
     mmeSize = modelTrmVec[numTerms-1].start
             + modelTrmVec[numTerms-1].nLevels();
}

void MME::calcWPW(){
    unsigned ii,jj,ti,tj;
    double vi,vj,tr_value;
    rhs.resize(mmeSize,0.0);
    lhs.resize(mmeSize,mmeSize,0.0);
    Ri = R.inv();
    for (unsigned i=0;i<dataVec.size();i++){
      for (unsigned mi=0;mi<numTerms;mi++){
        ii = modelTrmVec[mi].start + dataVec[i].trmVec[mi].level - 1;
        ti =  modelTrmVec[mi].trait;
        vi = dataVec[i].trmVec[mi].value;
        for (unsigned k=0;k<numTraits;k++){
             rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
        }
        for (unsigned mj=0;mj<numTerms;mj++){
          jj = modelTrmVec[mj].start + dataVec[i].trmVec[mj].level - 1;
          tj =  modelTrmVec[mj].trait;
          vj = dataVec[i].trmVec[mj].value;
          lhs[ii][jj] += vi*Ri[ti][tj]*vj;
        }
      }
    }
}

void MME::addGinv(){
```

```
        for (unsigned i=0;i<covBlockVec.size();i++){
                covBlockVec[i].addGinv();
        }
}

void MME::getSolution(){
        inputData();
        calcStarts();
        calcWPW();
        addGinv();
        sol = lhs.ginv0()*rhs;
}

void MME::display(){
    cout << "LHS " << endl;
    for (unsigned i = 0;i<mmeSize;i++){
        for (unsigned j = 0;j<mmeSize;j++){
           cout << setw(10)
           << setprecision (4)
           << setiosflags (ios::right | ios::fixed)

           << lhs[i][j] <<" ";
        }
        cout << endl;
    }
    cout << "RHS " << endl;
    cout << rhs << endl;
    for (unsigned i=0;i<modelTrmVec.size();i++){
        cout << "Solutions for " << modelTrmVec[i].name
             << ", Trait: " << modelTrmVec[i].depVarName << endl;
        Recoder<string>::iterator it;
        for (it=modelTrmVec[i].myRecoderPtr->begin();
             it!=modelTrmVec[i].myRecoderPtr->end();
             it++){
                unsigned ii = modelTrmVec[i].start + it->second - 1;
                cout << setw(10)
                     << it->first
                     << " "
```

```
                     << sol[ii]
                     << endl;
        }
    }
}

void CovBlock::buildModelTrmVec(string str){
   string sep(" ");
   Tokenizer modelTokens;
   modelTokens.getTokens(str,sep);
   unsigned nTokens = modelTokens.size();
   unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
   for (unsigned i=0;i<nTokens;i++){
     for (unsigned j=0;j<numModelTrms;j++){
       if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
          modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
          if (pedPtr){
             delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
             ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr
               = &pedPtr->coder;
          }
       }
     }
   }
}

void CovBlock::addGinv(void){
  Vari = Var.inv();
  unsigned n = modelTrmPtrVec.size();
  for (unsigned i=0;i<n;i++){
    ModelTerm* mtermiPtr = modelTrmPtrVec[i];
      unsigned starti = mtermiPtr->start;
      for (unsigned j=0;j<n;j++){
        ModelTerm* mtermjPtr = modelTrmPtrVec[j];
        unsigned startj = mtermjPtr->start;
        if (pedPtr){
          pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
        }
```

```
        else{
          unsigned numLevels = mtermiPtr->nLevels();
          for (unsigned k=0;k<numLevels;k++){
            ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
          }
        }
      }
  }
}


MME* ModelTerm::myMMEPtr;

// model abstraction using MME class: two-trait, animal model


int main() {
     try{
         matvec::SESSION.initialize("matvec_trash");
         Pedigree ped;
         ped.inputPed("Data/additive.ped");
         MME mme;
         matvec::doubleMatrix R;
         R.resize(2,2);
         R(1,1) = 1.0;
         R(1,2) = R(2,1) = 0.5;
         R(2,2) = 2.0;
         mme.R = R;
         ModelTerm::myMMEPtr = &mme;
         mme.fileName = "Data/additive2Tr.dat";
         mme.putColNames("directAdditive y1  y2");
         mme.putColTypes("CLASS  DEP DEP");

         mme.putModels("y1 = intercept directAdditive; \
                        y2 = intercept directAdditive;");

         mme.putVarCovMatrix("directAdditive",R,ped);
         mme.getSolution();
```

```
        mme.display();
    }
    catch (matvec::exception &ex) {
        cerr << ex.what() << "\n";
        exit(1);
    }
    catch (...) {
        cerr << "other exceptions were caught\n";
        exit(1);
    }
}
```

Chapter 3

Iterative Solvers for MME

Consider the system of consistent linear equations:

$$\boldsymbol{Ax} = \boldsymbol{b}.$$

Two iterative methods that we will use for solving the MME are the Jacobi method and the Preconditioned Conjugate Gradient (PCCG) method.

3.1 The Jacobi method

In the Jacobi method, the solution at iteration $n+1$ can be written as:

$$\boldsymbol{x}_{n+1} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_n) + \boldsymbol{x}_n. \tag{3.1}$$

Convergence can often be improved by modifying (3.1) as:

$$\boldsymbol{x}^*_{n+1} = \alpha\boldsymbol{x}_{n+1} + (1-\alpha)\boldsymbol{x}^*_n \tag{3.2}$$

for $0 < \alpha < 1$.

Straightforward application of the Jacobi method to solve the MME would require first computing the LHS and RHS of the MME, and then using (3.2) until convergence. Here, the LHS of the MME is represented by $\boldsymbol{A}$ and the RHS by $\boldsymbol{b}$. The LHS of the MME is often too large to store in memory as a "fully-stored" matrix. However, $\boldsymbol{A}$ is often very sparse. Thus, it is may be possible to store only the non-zero elements of $\boldsymbol{A}$ and compute $\boldsymbol{Ax}_n$, using sparse matrix methods.

An alternative method for iteration using (3.2) avoids storing even the non-zero elements of $\boldsymbol{A}$. In this approach, called "iteration of data", $\boldsymbol{A}\boldsymbol{x}_n$ is computed without first computing and storing $\boldsymbol{A}$. Thus, iteration on data (IOD) can be applied to very large systems. This approach is described in section 3.4.

3.2 The conjugate gradient method

In the conjugate gradient method, the solution at iteration $n+1$ is:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \alpha_n \boldsymbol{d}_n, \tag{3.3}$$

where

$$\alpha_n = \frac{-\boldsymbol{r}'_n \boldsymbol{r}_n}{\boldsymbol{d}'_n \boldsymbol{A} \boldsymbol{d}_n}, \tag{3.4}$$

$$\boldsymbol{r}_n = \boldsymbol{A}\boldsymbol{x}_n - \boldsymbol{b}, \tag{3.5}$$

$$\boldsymbol{d}_n = \boldsymbol{r}_n - \beta_{n-1} \boldsymbol{d}_{n-1}, \tag{3.6}$$

and

$$\beta_{n-1} = \frac{-\boldsymbol{r}'_n \boldsymbol{r}_n}{\boldsymbol{r}'_{n-1} \boldsymbol{r}_{n-1}}. \tag{3.7}$$

It can be shown that the residual can be computed as

$$\boldsymbol{r}_n = \boldsymbol{r}_{n-1} + \alpha_{n-1} \boldsymbol{A} \boldsymbol{d}_{n-1}, \tag{3.8}$$

and thus avoiding computation of $\boldsymbol{A}\boldsymbol{x}_n$. However, using (3.8) leads to the accumulation of rounding errors. Thus, it is recommended that (3.5) is used every 50 iterations.

As in the Jacobi method, in this method also the products $\boldsymbol{A}\boldsymbol{x}$ and $\boldsymbol{A}\boldsymbol{d}$ can be computed by IOD without first computing $\boldsymbol{A}$. Unlike the Jacobi method, it is not intuitively obvious why the conjugate gradient method works. Following is an attempt to explain why the method works.

In the conjugate gradient method, the value of $\boldsymbol{x}$ that minimizes

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}'\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}'\boldsymbol{x} \tag{3.9}$$

is obtained by line minimizations in n linearly independent directions, where n is the order of the symmetric positive definite matrix $\boldsymbol{A}$. Note that after minimization of $f(\boldsymbol{x})$ in any direction $\boldsymbol{d}_i$, the gradient $\boldsymbol{r}_{i+1}$ will be orthogonal to $\boldsymbol{d}_i$.

In the conjugate gradient method, each direction $\boldsymbol{d}_i$ is is chosen such that the gradient $\boldsymbol{r}_{i+1}$ will also be orthogonal to all the directions $\boldsymbol{d}_j$, for $j < i$, that have already been used for line minimization. If the directions are also linearly independent, after n line minimizations, the function will be at its minimum in n linearly independent directions. Further, at this point, the gradient

$$\boldsymbol{r} = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$$

is orthogonal to the n direction vectors. Thus, it must be the zero vector.

Following is a description of how the direction vectors are computed. Suppose the search is initiated at $\boldsymbol{x}_0 = \boldsymbol{0}$. At this point, the gradient of the $f(\boldsymbol{x})$ is

$$\begin{aligned} \boldsymbol{r}_0 &= \boldsymbol{A}\boldsymbol{x}_o - \boldsymbol{b} \\ &= -b. \end{aligned} \tag{3.10}$$

Let $\boldsymbol{d}_0 = \boldsymbol{r}_0$ be the first direction for line minimization. After minimization of $f(\boldsymbol{x})$ in the direction $\boldsymbol{d}_0$, the value of $\boldsymbol{x}$ is

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{d}_0. \tag{3.11}$$

At $\boldsymbol{x}_1$, the gradient of the function is

$$\begin{aligned} \boldsymbol{r}_1 &= \boldsymbol{A}\boldsymbol{x}_1 - \boldsymbol{b} \\ &= \boldsymbol{A}(\boldsymbol{x}_0 + \alpha_0 \boldsymbol{d}_0) - \boldsymbol{b} \\ &= \boldsymbol{r}_0 + \alpha_0 \boldsymbol{A}\boldsymbol{d}_0, \end{aligned} \tag{3.12}$$

and $\boldsymbol{r}_1$ is orthogonal to $\boldsymbol{d}_0$. Writing the product $\boldsymbol{d}_0\boldsymbol{r}_1 = 0$ as

$$\begin{aligned} \boldsymbol{d}_0'\boldsymbol{r}_1 &= \boldsymbol{d}_0'\boldsymbol{r}_0 + \alpha_0 \boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_0 \\ &= 0 \end{aligned} \tag{3.13}$$

shows that

$$\alpha_0 = \frac{-\boldsymbol{d}_0'\boldsymbol{r}_0}{\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_0}, \tag{3.14}$$

and in general,

$$\alpha_i = \frac{-\boldsymbol{d}_i'\boldsymbol{r}_i}{\boldsymbol{d}_i'\boldsymbol{A}\boldsymbol{d}_i}. \tag{3.15}$$

At this point, line minimization proceeds in the direction $\boldsymbol{d}_1$, and the value of $\boldsymbol{x}$ at the minimum is

$$\boldsymbol{x}_2 = \boldsymbol{x}_1 + \alpha_1\boldsymbol{d}_1. \tag{3.16}$$

The gradient of the function at $\boldsymbol{x}_2$ is

$$\begin{aligned} \boldsymbol{r}_2 &= \boldsymbol{A}\boldsymbol{x}_2 - \boldsymbol{b} \\ &= \boldsymbol{r}_1 + \alpha_1\boldsymbol{A}\boldsymbol{d}_1, \end{aligned} \tag{3.17}$$

and $\boldsymbol{r}_2$ is orthogonal to $\boldsymbol{d}_1$. In the conjugate gradient method, the direction $\boldsymbol{d}_1$ is chosen such that $\boldsymbol{r}_2$ will also be orthogonal to the direction $\boldsymbol{d}_0$. Thus the product

$$\begin{aligned} \boldsymbol{d}_0'\boldsymbol{r}_2 &= \boldsymbol{d}_0'\boldsymbol{r}_1 + \alpha_1\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_1 \\ &= \alpha_1\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_1 \\ &= 0. \end{aligned} \tag{3.18}$$

So, the direction, $\boldsymbol{d}_1$ must satisfy the condition

$$\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_1 = 0$$

in order for $\boldsymbol{r}_2$ to be orthogonal to $\boldsymbol{d}_0$. To accomplish this, following the Grahm-Schmidt procedure, $\boldsymbol{d}_1$ is written as

$$\boldsymbol{d}_1 = \boldsymbol{r}_1 - \beta_{10}\boldsymbol{d}_0. \tag{3.19}$$

Writing $\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_1 = 0$ as

$$\begin{aligned} \boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_1 &= \boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{r}_1 - \beta_{10}\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_0 \\ &= 0 \end{aligned} \tag{3.20}$$

shows that

$$\beta_{10} = \frac{\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{r}_1}{\boldsymbol{d}_0'\boldsymbol{A}\boldsymbol{d}_0}. \tag{3.21}$$

In general,

$$\boldsymbol{r}_{i+1} = \boldsymbol{r}_i + \alpha_i\boldsymbol{A}\boldsymbol{d}_i, \tag{3.22}$$

and by induction, $\boldsymbol{r}'_{i+1}\boldsymbol{d}_j = 0$ provided $\boldsymbol{d}'_i\boldsymbol{A}\boldsymbol{d}_j = 0$ for $j < i$. This can be achieved by using Grahm-Schmidt as

$$\boldsymbol{d}_i = \boldsymbol{r}_i - \sum_{j=0}^{i-1} \beta_{ij}\boldsymbol{d}_j, \tag{3.23}$$

where

$$\beta_{ij} = \frac{\boldsymbol{d}'_j\boldsymbol{A}\boldsymbol{r}_i}{\boldsymbol{d}'_j\boldsymbol{A}\boldsymbol{d}_j}. \tag{3.24}$$

Note that using the result $\boldsymbol{r}_i\boldsymbol{d}_j = 0$ for $j < i$ in (3.23) implies:

$$\boldsymbol{r}'_i\boldsymbol{r}_j = 0 \,\text{for}\, j < i. \tag{3.25}$$

Using (3.25) in (3.22), shows that $\boldsymbol{r}'_i\boldsymbol{A}\boldsymbol{d}_j = 0$ for $j < i-1$. Thus, in (3.24), $\beta_{ij} = 0$ for $j < i-1$, and the general expression for $\boldsymbol{d}_i$ simplifies to

$$\boldsymbol{d}_i = \boldsymbol{r}_i - \beta_{i-1}\boldsymbol{d}_{i-1}, \tag{3.26}$$

where

$$\beta_{i-1} = \frac{\boldsymbol{d}'_{i-1}\boldsymbol{A}\boldsymbol{r}_i}{\boldsymbol{d}'_{i-1}\boldsymbol{A}\boldsymbol{d}_{i-1}}. \tag{3.27}$$

Writing $\boldsymbol{r}_i$ as

$$\boldsymbol{r}_i = \boldsymbol{r}_{i-1} + \alpha_{i-1}\boldsymbol{A}\boldsymbol{d}_{i-1} \tag{3.28}$$

and pre-multiplying by $\boldsymbol{r}'_i$ gives

$$\boldsymbol{r}'_i\boldsymbol{r}_i = \alpha_{i-1}\boldsymbol{r}'_i\boldsymbol{A}\boldsymbol{d}_{i-1}. \tag{3.29}$$

Pre-multiplying (3.28) by $\boldsymbol{d}_{i-1}$ gives

$$\boldsymbol{d}'_{i-1}\boldsymbol{r}_{i-1} = -\alpha_{i-1}\boldsymbol{d}'_{i-1}\boldsymbol{A}\boldsymbol{d}_{i-1}. \tag{3.30}$$

Further, from (3.26), we can see that

$$\boldsymbol{d}'_i\boldsymbol{r}_i = \boldsymbol{r}'_i\boldsymbol{r}_i. \tag{3.31}$$

Using this is (3.30) gives

$$\boldsymbol{r}'_{i-1}\boldsymbol{r}_{i-1} = -\alpha_{i-1}\boldsymbol{d}'_{i-1}\boldsymbol{A}\boldsymbol{d}_{i-1}. \tag{3.32}$$

Using (3.29) and (3.32) in (3.27) gives

$$\beta_{i-1} = \frac{-\boldsymbol{r}_i'\boldsymbol{r}_i}{\boldsymbol{r}_{i-1}'\boldsymbol{r}_{i-1}}. \tag{3.33}$$

Finally, using (3.31) in (3.15) gives

$$\alpha_i = \frac{-\boldsymbol{r}_i'\boldsymbol{r}_i}{\boldsymbol{d}_i'\boldsymbol{A}\boldsymbol{d}_i}. \tag{3.34}$$

3.3 Preconditioned conjugate gradient method

In the PCCG method, the conjugate gradient method is applied to a transformed system of equations. The transformation of the system is based on a matrix $\boldsymbol{M}$ that is approximately equal to $\boldsymbol{A}$ and is easy to invert. A detailed explanation of PCCG is given in "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain" by Jonathan Richard Shewchuk.

In PCCG, the solution at iteration $n+1$ is:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \alpha_n \boldsymbol{d}_n, \tag{3.35}$$

where

$$\alpha_n = \frac{-\boldsymbol{r}_n'\boldsymbol{M}^{-1}\boldsymbol{r}_n}{\boldsymbol{d}_n'\boldsymbol{A}\boldsymbol{d}_n}, \tag{3.36}$$

$$\boldsymbol{r}_n = \boldsymbol{A}\boldsymbol{x}_n - \boldsymbol{b}, \tag{3.37}$$

$$\boldsymbol{d}_n = \boldsymbol{M}^{-1}\boldsymbol{r}_n - \beta_{n-1}\boldsymbol{d}_{n-1}, \tag{3.38}$$

$$\beta_{n-1} = \frac{-\boldsymbol{r}_n'\boldsymbol{M}^{-1}\boldsymbol{r}_n}{\boldsymbol{r}_{n-1}'\boldsymbol{M}^{-1}\boldsymbol{r}_{n-1}}. \tag{3.39}$$

As in the conjugate gradient method, the residual can be computed more efficiently as

$$\boldsymbol{r}_n = \boldsymbol{r}_{n-1} + \alpha_{n-1}\boldsymbol{A}\boldsymbol{d}_{n-1}. \tag{3.40}$$

However, it is recommended that (3.37) is used to every 50 iterations to avoid the accumulation of errors.

3.4 Iteration on data

Recall that in MME::calcWPW we first compute the LHS of the normal equations by accumulating the non-zero contributions from one observation at a time. Then, to obtain the LHS of the MME we accumulate contributions of the form (1.3). When the random effect is an additive effect, the non-zero contributions from (1.3) are computed one pedigree record at a time. For other random effects, the contributions are only to the diagonal elements of the MME (This can be thought of as a pedigree with unrelated individuals). In order to describe the IOD approach, let $c_k(i,j)$ denote the contribution to a_{ij} from record k. Here, a record may be an observation from the data file or a pedigree record.

Now to understand the principle underlying IOD, observe that in the product

$$\begin{aligned} \boldsymbol{q} &= \boldsymbol{A}\boldsymbol{d} \\ &= \{\sum_j a_{ij} d_j\}, \end{aligned} \tag{3.41}$$

a_{ij} is multiplied by d_j and the result is accumulated to q_i. However, element ij of the LHS can be written as

$$a_{ij} = \sum_k c_k(i,j). \tag{3.42}$$

Thus in the IOD approach, rather than going through all the records and first computing a_{ij} and then multiplying it by d_j, as record k is processed, if $c_k(i,j)$ is non-null, the product $c_k(i,j)x_j$ is computed and the result is accumulated to q_i. More explicitly, $\boldsymbol{Ax}$ is written as

$$\begin{aligned} \boldsymbol{q} &= \{\sum_j a_{ij} d_j\} \\ &= \{\sum_j \sum_k c_k(i,j) d_j\} \\ &= \{\sum_k \sum_j c_k(i,j) d_j\}, \end{aligned} \tag{3.43}$$

but as most of the $c_k(i,j)$ are null, q_i is updated only for non-null $c_k(i,j)$ as:

$$q_i = q_i + c_k(i,j) d_j \quad \text{for} \quad c_k(i,j) \neq 0. \tag{3.44}$$

3.5 Program changes for iteration on data

To iteratively solve the MME by Jacobi, using equations (3.1) and (3.2), the most demanding calculation is obtaining the product $\boldsymbol{Ax}$. A new method MME::mmeTimes, given below, is used to compute this product. In this method, $\boldsymbol{A}$ is the LHS of the MME and $\boldsymbol{x}$ is the argument to the function, "x". The result is placed in the static member of MME "MME::res", which was declared as a vector of "double" variables. Thus, this vector can be accessed from anywhere in the program as "MME::res".

```
352 void MME::mmeTimes(matvec::Vector<double>& x){
353   vec = &x;
354   calcWPW();
355   addGinv();
356 }
```

On line 353, the address of "x" is stored in MME::vec, which was declared as a pointer to a vector of "double" variables. This address is used in methods that make contributions to the LHS. These methods are: MME::calcWPW, CovBlock::addGinv, and Pedigree::addAinv. In these methods, when a direct solution is to be obtained, the contribution to position (ii, jj) of the LHS is accumulated in the matvec::doubleMatrix object "lhs" at position (ii, jj). On the other hand, when an iterative solution is to be obtained by IOD, the contribution to position (ii, jj) of the LHS is multiplied by element jj of the vector pointed to by "vec" and the result is stored in element ii of the vector "res". Further, if ii is equal to jj, the contribution to the diagonal of the LHS is accumulated in "diag". The relevant lines of code in these methods are given below.

MME::calcWPW

```
     if (solMethod=="direct"){
        lhs[ii][jj] += vi*Ri[ti][tj]*vj;
    }
    else {
        res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
        if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
    }
```

CovBlock::addGinv

```
     if (MME::solMethod=="direct") {
        ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
     }
     else {
        MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
        if (ii==jj)  MME::diag[ii] += Vari[i][j];
     }
```

Pedigree::addAinv

```
    if (MME::solMethod=="direct"){
       lhs[ii][jj] += ratio*q[i]*d*q[j];
    }
    else {
     MME::res[ii] += ratio*q[i]*d*q[j] * (*MME::vec)[jj];
     if (ii==jj)  MME::diag[ii] += ratio*q[i]*d*q[j];
    }
```

On the other hand, when an iterative solution is to be obtained by IOD, the contribution to position (ii, jj) of the LHS is multiplied by element jj of the vector pointed to by "vec" and the result is stored in element ii if the vector "res". Thus, if "solMethod" is not equal to "direct", a call to calcWPW results in the calculation of $\boldsymbol{Ax}$, where $\boldsymbol{A}$ is the LHS of the normal equations, and the result is stored in "MME::res".

Finally, the method MME::getJacobiSolution, given below, uses MME::mmeTimes to implement the Jacobi method. Lines 260 and 271 implement equation (3.1), and line 264 implements equation (3.2).

```
255 void MME::getJacobiSolution(double p){
256     solMethod = "jacobi";
257     initSetup();
258     mmeTimes(sol);// result goes into res, also creates rhs and diag
259     matvec::Vector<double> resid = (rhs - res);
260     tempSol = resid/diag + sol;
261     double diff = resid.sumsq();
262     unsigned iter = 0;
263     while(diff/mmeSize > .0000001 && ++iter<200){
```

```
        sol = p*tempSol + (1-p)*sol;
        cout <<"Iteration : "
             << iter <<" diff = "
             <<diff/mmeSize
             << endl << endl;
        mmeTimes(sol);
        resid = (rhs - res);
        tempSol = resid/diag + sol;
        diff = resid.sumsq();
    }
    cout << resid << endl;
}

```

3.6 Listings of IOD programs

3.6.1 ped.h

```
#ifndef PMap_H
#define PMap_H
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <cmath>
#include <map>
#include <vector>
#include <algorithm>
#include <functional>
#include "util.h"
#include <matvec/doublematrix.h>

```

```
using namespace std;

class PNode {
public:

  int     ind, sire, dam;
  double    f;
  string ind_str, sire_str, dam_str;


  PNode(string indstr, string sirestr, string damstr){

    ind = -1;
    sire = -1;
    dam = -1;
    f = -1.0;

    ind_str  = indstr;
    sire_str = sirestr;
    dam_str  = damstr;
  }

};


struct pcomp:public binary_function<double, double, bool> {
  bool operator()(PNode *x, PNode *y)
             { return x->ind < y->ind; }
};


class Pedigree : public map<string,PNode*> {


public:
  unsigned COUNT;
  SparseCij SpCij;
  vector <PNode*> pedVector;
```

```
  Recoder<string> coder;

  void inputPed(char* fname);

  void displayPed(void);
  void generateEntriesforParents(void);
  void codePed();
  void code(PNode *ptr);
  void calc_inbreeding(void);
  void makePedVector(void);
  void fillCoder(void);
  double get_rij(int i, int j);
  void output(char* ped);
  void addAinv(matvec::doubleMatrix& lhs,
               unsigned startRow,
               unsigned statCol,
               double ratio);
};

#endif

```

3.6.2 ped.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
#include <sstream>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <cmath>
#include <vector>
#include <algorithm>
#include <functional>
```

```
#include <map>
#include "ped.h"
#include "mme1.h"

using namespace std;

void Pedigree::inputPed(char* fname){
        cout << "reading pedigree file \n";
        double rec = 0, rec1 = 0;
        string indstr, sirestr, damstr;
        ifstream datafile(fname);
        if(!datafile){
                cout<< "Cannot open data file! \n";
                exit(1);
        }
        datafile.setf(ios::skipws);
        PNode *ptr;
        while (datafile>>indstr>>sirestr>>damstr){
                rec++;
                if(rec==1000){
                        cout<<".";
                        cout.flush();
                        rec1 += rec;
                        rec = 0;
                }
                ptr = new PNode(indstr, sirestr, damstr);
                (*this)[indstr] = ptr;
        }
        datafile.close();
        generateEntriesforParents();
        codePed();
        makePedVector();
        calc_inbreeding();
        fillCoder();
}

void Pedigree::displayPed(void){
        Pedigree::iterator it;
```

```
        vector<PNode*>::iterator vecit;
        for (vecit=pedVector.begin();vecit!=pedVector.end();vecit++){
                cout << setw(10) << (*vecit)->ind
            << setw(10) << (*vecit)->sire
            << setw(10) << (*vecit)->dam
            << setw(20) << ((*vecit)->ind_str).c_str()
            << setw(20) << (*vecit)->f <<     endl;
        }
        cout << endl;

}
void Pedigree::generateEntriesforParents(void) {
    // if a parent does not have an entry, we make it a founder
    cout << "generating missing entries for parents \n";
    unsigned sire_count = 0;
    unsigned dam_count = 0;
    double rec = 0, rec1 = 0;
    Pedigree::iterator it, parent_it;
    for(it=begin();it!=end();it++){
        rec++;
        if(rec==1000){
           cout<<".";
           cout.flush();
           rec1 += rec;
           rec = 0;
        }
        PNode *ptr = (*it).second;
        if(ptr->sire_str!="0"){
           parent_it = (*this).find(ptr->sire_str);
           if(parent_it == end()){ // sire has no entry
                PNode *ptrs = new PNode(ptr->sire_str, "0", "0");
                (*this)[ptr->sire_str] = ptrs;
           }
        }
        if(ptr->dam_str!="0"){
           parent_it = (*this).find(ptr->dam_str);
           if(parent_it == end()){ // dam has no entry
                PNode *ptrd =  new PNode(ptr->dam_str, "0", "0");
```

```
                (*this)[ptr->dam_str] = ptrd;
            }
        }
    }
}


void Pedigree::codePed(){
        cout << "coding pedigree \n";
        Pedigree::iterator it;
        COUNT = 0;
        unsigned rec = 0, rec1 = 0;

        for(it=begin();it!=end();it++){
                rec++;
                if(rec==1000){
                        cout<<".";
                        cout.flush();
                        rec1 += rec;
                        rec = 0;
                };
                cout.flush();
                PNode *ptr =(*it).second;
                code(ptr);
        }
}

void Pedigree::code(PNode *ptr){

        if(ptr->ind != -1) { // already coded
                return;
        }
        if(ptr->sire_str == "0" && ptr->dam_str == "0"){ // founder
                ptr->ind  = ++COUNT;
                ptr->sire = 0;
                ptr->dam  = 0;
        }
        else if(ptr->sire_str != "0" && ptr->dam_str == "0"){
```

```
                // dam missing, sire is not missing
                PNode* sire_ptr = (*this)[ptr->sire_str];
                if (sire_ptr->ind == -1) {
                        code(sire_ptr);
                }
                ptr->ind  = ++COUNT;
                ptr->sire = sire_ptr->ind;
                ptr->dam  = 0;
        }
        else if(ptr->dam_str != "0" && ptr->sire_str == "0"){
                // sire missing, dam is not missing
                PNode* dam_ptr = (*this)[ptr->dam_str];
                if (dam_ptr->ind == -1) {
                        code(dam_ptr);
                }
                ptr->ind  = ++COUNT;
                ptr->sire = 0;
                ptr->dam  = dam_ptr->ind;
        }
        else{
                PNode* sire_ptr = (*this)[ptr->sire_str];
                if (sire_ptr->ind == -1) {
                        code(sire_ptr);
                }
                PNode* dam_ptr = (*this)[ptr->dam_str];
                if (dam_ptr->ind == -1) {
                        code(dam_ptr);
                }
                ptr->ind = ++COUNT;
                ptr->sire = sire_ptr->ind;
                ptr->dam  = dam_ptr->ind;
        }
}

void Pedigree::calc_inbreeding(void){
        vector <PNode*>::iterator it;
        unsigned rec = 0, rec1 = 0, non_rec = 0;
        cout << "calculating inbreeding \n";
```

```
        for (it=pedVector.begin();it!=pedVector.end();it++){
                rec++;
                if(rec==1000){
                        cout<<".";
                        cout.flush();
                        rec1 += rec;
                        rec = 0;
                };
                (*it)->f = get_rij((*it)->sire,(*it)->dam);
        }
}


double Pedigree::get_rij(int i, int j){

        if (i==0||j==0){
                return 0.0;
        }
        double x = SpCij.retrieve_cij(i,j);
        if(x != -1.0) {
                return x;
        }
        int old, young;
        if(i < j){
                old = i;
                young = j;
        }
        else if(j < i){
                old = j;
                young = i;
        }
        else{
                double f = pedVector[i-1]->f;
                x = 0.5*(1 + f);
                SpCij.put_cij(i,j,x);
                return x;
        }
        int y_sire = pedVector[young-1]->sire;
```

```
        int y_dam  = pedVector[young-1]->dam;
        x = (get_rij(old,y_sire)+get_rij(old,y_dam))/2.0;
        SpCij.put_cij(i,j,x);
        return x;
}

void Pedigree:: output(char* ped){

        ofstream pedfile(ped);
        vector<PNode*>::iterator vecit;
        pedfile.setf(ios::fixed | ios::right);
        for (vecit=pedVector.begin();vecit!=pedVector.end();vecit++){
                pedfile << setw(10) << (*vecit)->ind
            << setw(10) << (*vecit)->sire
            << setw(10) << (*vecit)->dam
            << setw(20) << ((*vecit)->ind_str).c_str()
            << setw(20) << (*vecit)->f <<     endl;
        }
}

void Pedigree::makePedVector(void){
        Pedigree::iterator it;
        pedVector.resize(size());
        for(it=begin();it!=end();it++){
                PNode *ptr = (*it).second;
                unsigned i = ptr->ind - 1;
                pedVector[i] = ptr;
        }
}

void Pedigree::fillCoder(void){
        vector<PNode *>::iterator it;
        for(it=pedVector.begin();it!=pedVector.end();it++){
                coder.code((*it)->ind_str);
        }
}
```

```
void Pedigree::addAinv(matvec::doubleMatrix& lhs,
                  unsigned startRow,
                  unsigned startCol,
                  double ratio){
   double q[3];
   double d,fs,fd;
   unsigned pos[3];
   vector<PNode*>::iterator it;
   if (coder.size()>pedVector.size()){
        cout << "Pedigree is not complete \n";
        exit(-1);
   }
   for (it=pedVector.begin();it!=pedVector.end();it++){
        pos[0] = (*it)->sire;
        pos[1] = (*it)->dam;
        pos[2] = (*it)->ind;
        if((*it)->sire && (*it)->dam){
                q[0] = -0.5;
                q[1] = -0.5;
                q[2] =  1.0;
                fs = pedVector[pos[0]-1]->f;
                fd = pedVector[pos[1]-1]->f;
                d = 4.0/(2 - fs - fd);
        }
        else if((*it)->sire){
                q[0] = -0.5;
                q[1] =  0.0;
                q[2] =  1.0;
                fs = pedVector[pos[0]-1]->f;
                d = 4.0/(3-fs);
        }
        else if((*it)->dam){
                q[0] =  0.0;
                q[1] = -0.5;
                q[2] =  1.0;
                fd = pedVector[pos[1]-1]->f;
                d = 4.0/(3-fd);
        }
```

```
        else{
                q[0] =  0.0;
                q[1] =  0.0;
                q[2] =  1.0;
                d = 1.0;
        }
        for (unsigned i=0;i<3;i++){
           if(pos[i]){
              unsigned ii = startRow + pos[i] - 1;
              for (unsigned j=0;j<3;j++){
                 if(pos[j]) {
                    unsigned jj = startCol + pos[j] - 1;
                    if (MME::solMethod=="direct"){
                       lhs[ii][jj] += ratio*q[i]*d*q[j];
                    }
                    else {
                     MME::res[ii] += ratio*q[i]*d*q[j] * (*MME::vec)[jj];
                     if (ii==jj)  MME::diag[ii] += ratio*q[i]*d*q[j];
                    }
                 }
              }
           }
        }
   }
}
```

3.6.3 mme1.h

```
ifndef MME_H
#define MME_H
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
```

```
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"
#include "ped.h"


// classes for multiple trait, mixed models with iteration on data

using namespace std;

class TermData{
public:
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  vector<double>    depVec;
};

class MME;

class ModelTerm{
public:
        unsigned start;
        unsigned trait;
        string name;
        string depVarName;
        static MME *myMMEPtr;
        Recoder<string>* myRecoderPtr;
        vector<unsigned> factors;
```

```

        unsigned code(string str){return myRecoderPtr->code(str);}
        unsigned nLevels(){return myRecoderPtr->size();}
        void putFactors(string str);
        string   getTermString();
        unsigned getTermLevel (){
                return code(getTermString());
        }
        double   getTermValue ();
};

class CovBlock {
public:
     vector<ModelTerm*> modelTrmPtrVec;
     matvec::doubleMatrix Var, Vari;
     Pedigree* pedPtr;
     CovBlock(void){pedPtr = 0;}
     CovBlock(string str, matvec::doubleMatrix V){
         Var = V;
         pedPtr = 0;
         buildModelTrmVec(str);
     }
     CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
         Var = V;
         pedPtr = &P;
         buildModelTrmVec(str);
     }
     void buildModelTrmVec(string str);
     void addGinv(void);
};


class MME {
 private:
  void putModel(string str);
public:
    static string solMethod;
    static matvec::Vector<double> *vec, diag, res;
```

```
    string fileName;
    Tokenizer colType;
    Tokenizer colName;
    Tokenizer depVar;
    Tokenizer colData;
    unsigned numCols;
    unsigned depCol;
    vector <ModelTerm> modelTrmVec;
    vector <CovBlock>  covBlockVec;
    vector <DataNode> dataVec;
    unsigned numTerms, numTraits;
    unsigned mmeSize;
    matvec::doubleMatrix lhs, R, Ri;
    matvec::Vector<double> rhs, sol, tempSol;

    void putColNames(string str);
    void putColTypes(string str);
    void putModels(string str);
    void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
       CovBlock covBlock(str,V,P);
       covBlockVec.push_back(covBlock);
    }
    void putVarCovMatrix(string str, matvec::doubleMatrix V){
       CovBlock covBlock(str,V);
       covBlockVec.push_back(covBlock);
    }
    void initSetup();
    void inputData();
    void displayData();
    static double getDouble(string& Str);
    void calcStarts();
    void getDirectSolution();
    void getJacobiSolution(double p);
    void getCGSolution();
    void getPCCGSolution();

    void mmeTimes(matvec::Vector<double>& x);
    void calcWPW();
```

```
    void addGinv();
    void display();
};
#endif
```

3.6.4 mme1.cpp

```
/*
 *  mme1.cpp
 *  C++WkShp
 *
 *  Created by Rohan Fernando on 5/6/05.
 *  Copyright 2005,   All rights reserved.
 *
 */

#include "mme1.h"

MME* ModelTerm::myMMEPtr;
string MME::solMethod;
matvec::Vector<double> *MME::vec, MME::diag, MME::res;

 void ModelTerm::putFactors(string str){
  Tokenizer tokens;
  string sep("*");
  tokens.getTokens(str,sep);
  factors.clear();
  for (unsigned i=0;i<tokens.size();i++){
    unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
    if (factorIndex == -1){
      cerr <<"Independent Variable "
           << tokens[i]
           << " not in list of column names \n";
      exit(-1);
    }
    else {
      factors.push_back(factorIndex);
    }
```

```
  }
}


string ModelTerm::getTermString(){
    unsigned numFactors = factors.size();
    string trmStr;
    unsigned factorIndex = factors[0];
    if(myMMEPtr->colType[factorIndex]=="COV"){
            trmStr = myMMEPtr->colName[factorIndex];
    }
    else {
            trmStr = myMMEPtr->colData[factorIndex];
    }
    for (unsigned i=1;i<numFactors;i++){
       factorIndex = factors[i];
       if(myMMEPtr->colType[factorIndex]=="COV"){
               trmStr += "*" + myMMEPtr->colName[factorIndex];
       }
       else{
               trmStr += "*" + myMMEPtr->colData[factorIndex];
       }
    }
    return trmStr;
}

double ModelTerm::getTermValue(){
    unsigned numFactors = factors.size();
    double value = 1.0;
    for (unsigned i=0;i<numFactors;i++){
     unsigned factorIndex = factors[i];
      if (myMMEPtr->colType[factorIndex]=="COV"){
          string covStr = myMMEPtr->colData[factorIndex];
          value *= MME::getDouble(covStr);
      }
    }
    return value;
}
```

```

void MME::putColNames(string str){
  string sep(" ");
  str = "intercept " + str;
  colName.getTokens(str,sep);
  numCols = colName.size();
}

void MME::putColTypes(string str){
  string sep(" ");
  str = "CLASS " + str;
  colType.getTokens(str,sep);
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
  }
  unsigned n = 0;
  for (unsigned i=0;i<numCols;i++){
        if (colType[i] == "DEP") {
                depVar.push_back(colName[i]);
                n++;
        }
  }
  numTraits = n;
}

void MME::putModels(string str){
  Tokenizer models;
  string sep =";";
  models.getTokens(str,sep);
  for (unsigned i=0; i<models.size();i++){
    putModel(models[i]);
  }
}

void MME::putModel(string str){
   string sep(" =+");
   Tokenizer modelTokens;
```

```
   modelTokens.getTokens(str,sep);
   unsigned nTokens = modelTokens.size();
   int depVarIndex = colName.getIndex(modelTokens[0]);
   if (depVarIndex == -1){
     cerr << "Dependent Variable "
          << modelTokens[0]
          << " not in list of column names \n";
     exit (-1);
   }
   ModelTerm modelTrm;
   modelTrm.depVarName = modelTokens[0];
   modelTrm.trait = depVar.getIndex(modelTokens[0]);
   for (unsigned i=1;i<nTokens;i++){
           modelTrm.myRecoderPtr = new Recoder<string>;
           modelTrm.name = modelTokens[i];
           modelTrm.putFactors(modelTrm.name);
           modelTrmVec.push_back(modelTrm);
   }
}

void MME::inputData(){
     DataNode dataNode;
     numTerms = modelTrmVec.size();
     dataNode.trmVec.resize(numTerms);
     dataNode.depVec.resize(numTraits);
     ifstream datafile;
     datafile.open(fileName.c_str());
     if(!datafile) {
       cerr << "Couldn't open data file: " << fileName << endl;
       exit (-1);
     }
     unsigned linewidth = 1024;
     char *line = new char [linewidth];
     string sep(" ");
     while (datafile.getline(line,linewidth)){
       string inputStr(line);
       inputStr = "--- " + inputStr;
       colData.getTokens(inputStr,sep);
```

```
      unsigned j=0;
      for (unsigned i=0;i<numCols;i++){
        if (colType[i]=="DEP") {
           dataNode.depVec[j++] = getDouble(colData[i]);
        }

      }
      for (unsigned i=0;i<numTerms;i++){
        dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
        dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
      }
      dataVec.push_back(dataNode);
   }
}

double MME::getDouble(string& Str) {
  ifstream inputStrStream(Str.c_str());
  double val;
  inputStrStream >> val;
  return val;
}

void MME::calcStarts(){
    modelTrmVec[0].start = 0;
    for (unsigned i=1;i<numTerms;i++){
      modelTrmVec[i].start = modelTrmVec[i-1].start
                           + modelTrmVec[i-1].nLevels();
    }
    mmeSize = modelTrmVec[numTerms-1].start
            + modelTrmVec[numTerms-1].nLevels();
}

void MME::calcWPW(){
    unsigned ii,jj,ti,tj;
    double vi,vj,tr_value;
    if(solMethod!="direct"){
       diag.resize(mmeSize,0.0);
       res.resize(mmeSize,0.0);
```

```
    }
    rhs.resize(mmeSize,0.0);
    Ri = R.inv();
    for (unsigned i=0;i<dataVec.size();i++){
       for (unsigned mi=0;mi<numTerms;mi++){
           ii = modelTrmVec[mi].start
              + dataVec[i].trmVec[mi].level - 1;
           ti =  modelTrmVec[mi].trait;
           vi = dataVec[i].trmVec[mi].value;
           for (unsigned k=0;k<numTraits;k++){
             rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
           }
           for (unsigned mj=0;mj<numTerms;mj++){
             jj = modelTrmVec[mj].start
                + dataVec[i].trmVec[mj].level - 1;
             tj =  modelTrmVec[mj].trait;
             vj = dataVec[i].trmVec[mj].value;
             if (solMethod=="direct"){
                lhs[ii][jj] += vi*Ri[ti][tj]*vj;
             }
             else {
                res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
                if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
             }
           }
       }
    }
}

void MME::addGinv(){
        for (unsigned i=0;i<covBlockVec.size();i++){
                covBlockVec[i].addGinv();
        }
}

void MME::initSetup(){
  inputData();
  calcStarts();
```

```
  if(solMethod=="direct"){
    lhs.resize(mmeSize,mmeSize,0.0);
  }
  else {
    sol.resize(mmeSize,0.0);
  }
}


void MME::getDirectSolution(){
        solMethod = "direct";
        initSetup();
        calcWPW();
        addGinv();
        sol = lhs.ginv0()*rhs;
}

void MME::display(){

    if (solMethod=="direct"){
      cout << "LHS " << endl;
      for (unsigned i = 0;i<mmeSize;i++){
        for (unsigned j = 0;j<mmeSize;j++){
                cout << setw(5) << lhs[i][j] <<" ";
        }
        cout << endl;
      }
    }
    cout << "RHS " << endl;
    cout << rhs << endl;
    for (unsigned i=0;i<modelTrmVec.size();i++){
      cout << "Solutions for " << modelTrmVec[i].name
           << ", Trait: " << modelTrmVec[i].depVarName << endl;
      Recoder<string>::iterator it;
      for (it=modelTrmVec[i].myRecoderPtr->begin();
           it!=modelTrmVec[i].myRecoderPtr->end();
           it++){
         unsigned ii = modelTrmVec[i].start + it->second - 1;
```

```
          cout << setw(10) << it->first << " " << sol[ii] << endl;
       }
     }
}

void MME::getJacobiSolution(double p){
     solMethod = "jacobi";
     initSetup();
     mmeTimes(sol);// result goes into res, also creates rhs and diag
     matvec::Vector<double> resid = (rhs - res);
     tempSol = resid/diag + sol;
     double diff = resid.sumsq();
     unsigned iter = 0;
     while(diff/mmeSize > .0000001 && ++iter<200){
        sol = p*tempSol + (1-p)*sol;
        cout <<"Iteration : "
             << iter <<" diff = "
             <<diff/mmeSize
             << endl << endl;
        mmeTimes(sol);
        resid = (rhs - res);
        tempSol = resid/diag + sol;
        diff = resid.sumsq();
     }
     cout << resid << endl;
}

void MME::getCGSolution(){
     solMethod = "cg";
     initSetup();
     mmeTimes(sol); // result goes into res, also creates rhs and diag
     matvec::Vector<double> resid = (res-rhs);
     matvec::Vector<double> d = resid;
     double oldDiffSq;
     double newDiffSq = resid.sumsq();
     unsigned iter = 0;
     while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
       cout <<"Iteration : "
```

```
            << iter <<" diff = "
            <<newDiffSq/mmeSize
            << endl << endl;
      mmeTimes(d);
      double alpha = -newDiffSq/(res*d).sum();
      sol += alpha*d;
      if (iter%10){
         resid += alpha*res;
      }
      else {
         mmeTimes(sol);
         resid = (res-rhs);
      }
      oldDiffSq = newDiffSq;
      newDiffSq = resid.sumsq();
      double beta = -newDiffSq/oldDiffSq;
      d = resid - beta*d;
    }
    cout << resid << endl;
}

void MME::getPCCGSolution(){
    solMethod = "pccg";
    initSetup();
    mmeTimes(sol);// result goes into res, also creates rhs and diag
    matvec::Vector<double> resid = (res - rhs);
    matvec::Vector<double> d = resid/diag;
    double oldDiffSq;
    double newDiffSq = (resid*d).sum();
    unsigned iter = 0;
    while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
      cout <<"Iteration : "
            << iter <<" diff = "
            <<newDiffSq/mmeSize
            << endl << endl;
      mmeTimes(d);
      double alpha = -newDiffSq/(res*d).sum();
      sol += alpha*d;
```

```
      if (iter % 10){
              resid += alpha*res;
      }
      else {
              mmeTimes(sol);
              resid = (res - rhs);
      }
      matvec::Vector<double> s = resid/diag;
      oldDiffSq = newDiffSq;
      newDiffSq = (resid*s).sum();
      double beta = -newDiffSq/oldDiffSq;
      d = s - beta*d;
    }
    cout << resid << endl;
}

void MME::mmeTimes(matvec::Vector<double>& x){
  vec = &x;
  calcWPW();
  addGinv();
}

void CovBlock::buildModelTrmVec(string str){
  string sep(" ");
  Tokenizer modelTokens;
  modelTokens.getTokens(str,sep);
  unsigned nTokens = modelTokens.size();
  unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
  for (unsigned i=0;i<nTokens;i++){
    for (unsigned j=0;j<numModelTrms;j++){
      if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
        modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
        if (pedPtr){
          delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
          ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr
          = &pedPtr->coder;
        }
      }
```

```
    }
  }
}

void CovBlock::addGinv(void){
   Vari = Var.inv();
   unsigned n = modelTrmPtrVec.size();
   for (unsigned i=0;i<n;i++){
     ModelTerm* mtermiPtr = modelTrmPtrVec[i];
     unsigned starti = mtermiPtr->start;
     for (unsigned j=0;j<n;j++){
       ModelTerm* mtermjPtr = modelTrmPtrVec[j];
       unsigned startj = mtermjPtr->start;
       if (pedPtr){
         pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,
                         starti,startj,Vari[i][j]);
       }
       else{
         unsigned numLevels = mtermiPtr->nLevels();
         for (unsigned k=0;k<numLevels;k++){
            unsigned ii = starti + k;
            unsigned jj = startj + k;
            if (MME::solMethod=="direct") {
               ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
            }
            else {
               MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
               if (ii==jj)  MME::diag[ii] += Vari[i][j];
            }
         }
       }
     }
   }
}
```

3.6.5 iterDataMSMME.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <strstream>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"
#include "ped.h"
#include "mme1.h"


// model abstraction using MME class: animal model, iteration on data, PCCG


int main() {
        try{
                matvec::SESSION.initialize("matvec_trash");
                Pedigree ped;
                ped.inputPed("Data/additive.ped");
                ped.displayPed();
                MME mme;
                matvec::doubleMatrix R;
                R.resize(1,1);
                R(1,1) = 1.0;
                mme.R = R;
                ModelTerm::myMMEPtr = &mme;
                mme.fileName = "Data/additive2Tr.dat";
                mme.putColNames("directAdditive y1  y2");
                mme.putColTypes("CLASS  DEP DEP");
                mme.putModels("y1 = intercept directAdditive;");
```

```
                mme.putVarCovMatrix("directAdditive",R,ped);
                mme.getPCCGSolution();
                mme.display();
        }
        catch (matvec::exception &ex) {
                cerr << ex.what() << "\n";
                exit(1);
        }
        catch (...) {
                cerr << "other exceptions were caught\n";
                exit(1);
        }
}

```

Chapter 4

Marker Assisted BLUP

4.1 Single trait models

Consider the following univariate, mixed linear model:

$$y_i = \boldsymbol{x}_i\boldsymbol{\beta} + v_i^p + v_i^m + u_i + e_i, \tag{4.1}$$

where y_i is the phenotypic value for animal i, $\boldsymbol{x}_i\boldsymbol{\beta}$ is the fixed part of the model, v_i^p and v_i^m are the random additive values for the paternal and maternal alleles at a marked QTL (MQTL), u_i is the random additive genotypic value for all the remaining QTL (RQTL), and e_i is the residual. We assume here that the markers and the MQTL are in gametic phase equilibrium.

Let $\boldsymbol{v}$ denote the vector of random allelic values at the MQTL. Note that two elements of this vector appear in the model for an observation. In all the previous models that we considered, only one element from a set of effects was in the model for a particular observation. Thus, although we will have two model terms to represent v_i^p and v_i^m, both of these will have the same "start" value as they are elements of the same vector $\boldsymbol{v}$. Also, to setup the MME for this model, we need the inverse of $\text{Var}(\boldsymbol{v}) = \boldsymbol{\Sigma}_v$. This covariance matrix can be approximated by the recursive formula:

$$\begin{aligned}\text{Cov}(v_i^m, v_k^p|\boldsymbol{M}) &= \Pr(O_Q(m,i) = m|\boldsymbol{M})\text{Cov}(v_d^m, v_k^p|\boldsymbol{M}) \\ &\quad + \Pr(O_Q(m,i) = p|\boldsymbol{M})\text{Cov}(v_d^p, v_k^p|\boldsymbol{M}),\end{aligned} \tag{4.2}$$

where $O_Q(m,i) = m$, for example, is the event that the maternal MQTL allele of individual i originates in its dam's maternal allele. These allele origin events are also called segregation events and the probability of this allele origin event is called a segregation probability, and in the animal breeding literature, this segregation probability at an MQTL has been called the "probability of descent for QTL allele" (PDQ).

As described below, use of equation (4.2) to construct the matrix $\mathbf{\Sigma}_v$ of gametic covariances can be expressed in matrix notation. To do so, the rows and columns of $\mathbf{\Sigma}_v$ are ordered such that those for ancestors precede those for descendants. Suppose $\mathbf{\Sigma}_s$ is the gametic covariance matrix for individuals $1, 2, \ldots, i-1$. This matrix can be expanded to include the covariances with v_i^m, for example, as

$$\mathbf{\Sigma}_{s+1} = \begin{bmatrix} \mathbf{\Sigma}_s & \mathbf{\Sigma}_s \boldsymbol{q} \\ \boldsymbol{q}'\mathbf{\Sigma}_s & \mathrm{Var}(v_i^m) \end{bmatrix}, \tag{4.3}$$

where $\boldsymbol{q}$ is a $2(i-1) \times 1$ vector with the maternal and paternal PDQ's for v_i^m at the positions corresponding to v_d^m and v_d^p, and zero at all the other positions. Under the assumption of equilibrium between the markers and the MQTL, $\mathrm{Var}(v_i^m) = \sigma_Q^2$ is a constant. Suppose the PDQ for an MQTL allele is 1 or zero. Then, it is not necessary to include the random effect for this allele in the model, and including this random effect in the model will make $\mathbf{\Sigma}_v$ singular. For example, suppose individual 1 is the dam of 3 and $\Pr(O_Q(m,3) = m|\boldsymbol{M}) = 0.0$ and $\Pr(O_Q(p,3) = m|\boldsymbol{M}) = 0.9$. Then, the model for the phenotypic value of individual 3 should be written as

$$y_3 = \boldsymbol{x}_3\boldsymbol{\beta} + v_3^p + v_1^p + u_3 + e_i, \tag{4.4}$$

because we can trace the maternal MQTL allele of individual 3 to the paternal MQTL allele of 1 with certainty.

Given (4.3), as described here, partitioned matrix theory can be used to obtain the inverse of $\mathbf{\Sigma}_v$ efficiently. Suppose that $\mathbf{\Sigma}_s^{-1}$ is the inverse of the sub matrix $\mathbf{\Sigma}_s$ defined previously, then the inverse of $\mathbf{\Sigma}_{s+1}$ is

$$\mathbf{\Sigma}_{s+1}^{-1} = \begin{bmatrix} \mathbf{\Sigma}_s^{-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} -\boldsymbol{q} \\ 1 \end{bmatrix} (\frac{1}{v_{ii}}) \begin{bmatrix} -\boldsymbol{q}' & 1 \end{bmatrix}, \tag{4.5}$$

where

$$v_{ii} = [\mathrm{Var}(v_i^m) - \boldsymbol{q}'\mathbf{\Sigma}_s\boldsymbol{q}]^{\cdot} \tag{4.6}$$

Note that $\boldsymbol{q}$ has only two non-zero elements. Thus, (4.5) leads to an efficient algorithm, and the resulting inverse is very sparse. Also, because $\boldsymbol{q}$ has only two non-zero elements, only four elements from $\mathbf{\Sigma}_{\boldsymbol{v}}$ are needed to compute v_{ii}, and these elements can be obtained efficiently without constructing the entire $\mathbf{\Sigma}_v$ matrix by use of (4.2).

4.2 Multiple trait models

Suppose an MQTL has only two alleles, Q_1 and Q_2, segregating in the population. Then, the value of a random MQTL allele for trait t is $(Q - p)a_t$, where $Q = 0$ for MQTL allele Q_1 and $Q = 1$ for Q_2, p is the frequency of Q_2, and a_t is the difference between the effect of alleles Q_2 and Q_1 on trait t. Now, let $\boldsymbol{v}_t$ denote the vector of random MQTL allele values for trait t. Then, the vector for trait t' is:

$$\boldsymbol{v}_{t'} = \boldsymbol{v}_t \frac{a_{t'}}{a_t}. \tag{4.7}$$

Thus, even when an MQTL has an effect on more than one trait, a vector of MQTL allelic values is fitted for only one of the traits. The effects of the MQTL alleles on all other traits is modeled as multiple of this vector. Note that for all other random effects we fitted a separate random effect for each trait.

4.3 Classes and methods for MABLUP

Objects of a new class, MQTL, will be used to model terms for marked QTL. The declaration of this class and related QNode class are given in section 4.4.1.

Note that MQTL is a vector of pointers to QNode objects. A QNode has two unsigned members, "mLevel" and "pLevel", to store the levels for the maternal and paternal alleles of an individual. Also, it has two double members to store the maternal and paternal PDQ, and another double member to store the inbreeding coefficient at the MQTL conditional on the marker information.

The MQTL::inputPDQ method reads in the PDQ's for each individual from a text file. After reading in all the PDQ's, the MQTL::generateMQTLLevels method is called. This method calls the MQTL::codeMQTL method, which makes sure that an individual parents are coded before coding its alleles (lines 82 and 98; the line numbers are from the listing of MQTL.cpp in section 4.4.2). If an individuals paternal PDQ is equal to one, the paternal allele gets the same code as the sires maternal allele; if it is equal to zero, the paternal allele gets the same code as the sires paternal allele; if neither of these conditions is true, the paternal allele will get its own code (lines 83–91). The same strategy is used to code the maternal allele (lines 99–107).

Inbreeding is calculated by method MQTL::calcMQTLInbreeding, which calls MQTL::getMQTLrij to compute Malecot's coefficient of relationship between the maternal and paternal MQTL alleles for each individual, conditional on the marker information. The method MQTL::getMQTLrij uses recursive formula (4.2); an object, "spCij", of type SparseCij is used to store all relationship coefficients to avoid repeated computation of the same coefficient. The method MQTL::addGinv implements the efficient algorithm to invert $\mathbf{\Sigma}_v$.

4.3.1 The main program

The main program given below shows the use of an MQTL object,"Q", for MABLUP. The elements in "Q.regCoeff" are the coefficients in (4.7) that give the relationship between the vectors of MQTL values for different traits (lines 31–34). The "Q.MQTLNames" are the names used in the model for the paternal and maternal allelic values in the model (line 36).

Now, the statement on line 44 of the main program saves the memory address of "Q" in "mme.MQTLVec", which was declared in mmeMQTL.h as a vector of pointers to MQTL objects. The statement on line 45 merges the data for the MQTL levels in "Q" with the observations in the data file by the variable "directAdditive". The statements on lines 46 and 47 give the names and types for the variables added to the data file. Finally, the statement on line 60 is used to tell the mme object that the inverse of $\mathbf{\Sigma}_v$ has to be added to the position of the LHS corresponding to $\boldsymbol{v}$. In the following section, we will examine in more detail the changes to the methods of the MME class. The declaration of the MME and related classes is in 4.4.3, and the implementation of the methods are given in 4.4.4.

```
22 int main() {
23    try{
24        matvec::SESSION.initialize("matvec_trash");
25        Pedigree ped;
26        ped.inputPed("Data/additive.ped");
27        ped.displayPed();
28
29        MQTL Q(ped);
30        Q.inputPDQ("Data/additive.pM");
31        matvec::Vector<double> regCoeff(2);
```

```
        regCoeff(1) = 1.0;
        regCoeff(2) = 0.9;
        Q.regCoeff = regCoeff;
        Q.variance = 0.25;
        Q.MQTLNames = "pQ1 mQ1";
        Q.display();

        MME mme;

        mme.fileName = "Data/additive2Tr.dat";
        mme.putColNames("directAdditive y1 y2");
        mme.putColTypes("CLASS          DEP DEP");
        mme.putMQTL(Q);
        mme.mergeMQTLLevelsBy("directAdditive");
        mme.addColNames("pQ1  mQ1");
        mme.addColTypes("MQTL MQTL");
        string modelString = "y1  = intercept + directAdditive + pQ1 + mQ1;";
        modelString       += "y2  = intercept + directAdditive + pQ1 + mQ1 ";
        mme.putModels(modelString);

        matvec::doubleMatrix Va;
        Va.resize(2,2,0.0);
        Va(1,1) = 1.0;
        Va(1,2) = 0.0;
        Va(2,1) = 0.0;
        Va(2,2) = 1.0;
        mme.putVarCovMatrix("directAdditive",Va,ped);

        mme.putVarCovMatrix(Q);

        matvec::doubleMatrix resVar;
        resVar.resize(2,2,0.0);
        resVar(1,1) = 1.0;
        resVar(1,2) = 0.0;
        resVar(2,1) = 0.0;
        resVar(2,2) = 1.0;
        mme.R = resVar;

```

```
        mme.getDirectSolution();
        mme.display();

        }
         catch (matvec::exception &ex) {
                 cerr << ex.what() << "\n";
                 exit(1);
         }
         catch (...) {
                 cerr << "other exceptions were caught\n";
                 exit(1);
         }
}
```

4.3.2 Method MME::putModels

A "for loop" (lines 104–106) was added here to add information to model terms corresponding to the MQTL allelic values by calling method "putMQTLStuffInModelTerms".

```
96   void MME::putModels(string str){
97          ModelTerm::myMMEPtr = this;
98          Tokenizer models;
99          string sep =";";
100         models.getTokens(str,sep);
101         for (unsigned i=0; i<models.size();i++){
102                 putModel(models[i]);
103         }
104         for(unsigned i=0;i<MQTLVec.size();i++){
105                 putMQTLStuffInModelTerms(*MQTLVec[i]);
106   }
107  }
108
```

4.3.3 Method MME::putMQTLStuffInModelTerms

The "for loop" starting on line 469 is used to examine the name of each model term in mme to see if it matches one of the names of the MQTL allelic values. If there is a match, the address of the MQTL object is stored in the "myMQTLPtr" member of ModelTerm (line 471) and the default recoder for this model term is switched with that for the MQTL object (lines 372–373). If the name of the model term matches the second name in the MQTL object (line 475), the "secondMQTLEffect" member of ModelTerm is set to "true" (line 476). Now that the information from the MQTL objects have been transfered to the model terms, the "starts" can be calculated using calcStarts.

```
462 void MME::putMQTLStuffInModelTerms(MQTL& mQTL){
463     CovBlock covBlock(mQTL);
464     Tokenizer names;
465     string sep = " ,";
466     names.getTokens(mQTL.MQTLNames,sep);
467     string firstMQTL  = names[0];
468     string secondMQTL = names[1];
469     for (unsigned i=0;i<modelTrmVec.size();i++){
470        if(modelTrmVec[i].name==firstMQTL || modelTrmVec[i].name==secondMQTL){
471           modelTrmVec[i].myMQTLPtr = &mQTL;
472           delete modelTrmVec[i].myRecoderPtr;
473           modelTrmVec[i].myRecoderPtr = &mQTL.myRecoder;
474        }
475        if(modelTrmVec[i].name==secondMQTL){
476           modelTrmVec[i].secondMQTLEffect = true;
477
478        }
479     }
480 }
```

4.3.4 Method MME::calcStarts

Recall that in the main program, the model contained four terms for MQTL allelic values. However, we only include one set of MQTL allelic effects in the model. Thus, the start for the fist MQTL model term is calculated the usual way, and for all subsequent model terms the start value from the first model term is used.

```
void MME::calcStarts(){
    unsigned prevI =0,  maxI = 0;
```

```
182     modelTrmVec[0].start = 0;
183     for (unsigned i=1;i<numTerms;i++){
184         if (modelTrmVec[i].myMQTLPtr) {
185             if(modelTrmVec[i].myMQTLPtr->myStart == -1){
186                modelTrmVec[i].start = modelTrmVec[prevI].start
187                                     + modelTrmVec[prevI].nLevels();
188                prevI = i;
189                modelTrmVec[i].myMQTLPtr->myStart = modelTrmVec[i].start;
190             }
191             else {
192                modelTrmVec[i].start = modelTrmVec[i].myMQTLPtr->myStart;
193             }
194         }
195         else {
196             modelTrmVec[i].start = modelTrmVec[prevI].start
197                                  + modelTrmVec[prevI].nLevels();
198             prevI = i;
199         }
200         if (modelTrmVec[i].start > modelTrmVec[maxI].start){
201             maxI = i;
202         }
203     }
204     mmeSize = modelTrmVec[maxI].start + modelTrmVec[maxI].nLevels();
205 }
```

The "if statement" on line 184 checks to see if this model term is for an MQTL, and if this is true, the start value is calculated as described above; else, the start is calculated as done in previous versions of the method. Note that "prevI" is the index of the previous model term that was included in the model, which may not be equal to $i-1$.

4.3.5 Other changes

Once the "starts" are calculated for the model terms, "positions" are calculated as before. However, "values" for MQTL model terms are calculated using the coefficients stored in the MQTL memeber "regCoeff" in "MME::inputData" (lines 161–164).

In order to add $\mathbf{\Sigma}_v^{-1}$ to the LHS of the MME, the following "if block" was added to CovBlock::addGinv:

```
  if(myMQTLPtr){
      myMQTLPtr->addGinv(ModelTerm::myMMEPtr->lhs,
                                          myMQTLPtr->myStart,
                                          myMQTLPtr->myStart,
                                          1.0/myMQTLPtr->variance);
      return;
  }
```

4.4 Listing of MAPLUP programs

4.4.1 Listing of MQTL.h

```
#ifndef MQTL_H
#define MQTL_H
#include <fstream>
#include <iostream>
#include <iomanip>
#include <vector>
#include "ped.h"
#include "util.h"

using namespace std;

class QNode {
public:
        unsigned mLevel, pLevel;
        double   mPDQ,   pPDQ, f;

        QNode(){
                mLevel = 0;
                pLevel = 0;
                f=-1;
        }
};

class MQTL:public vector<QNode*> {
```

```
public:
        unsigned count;
        MQTL(Pedigree& P){myPedPtr = &P; myStart = -1;}
        static Pedigree* myPedPtr;
        SparseCij SpCij;
        string MQTLNames;
        matvec::Vector<double> regCoeff;
        Recoder<string> myRecoder;
        double variance;
        int myStart;

        void inputPDQ(char *filename);
        void generateMQTLLevels(void);
        void codeMQTL(unsigned i);
        void calcMQTLInbreeding(void);
        double getMQTLrij(unsigned i,
                          unsigned j,
                          unsigned pi, unsigned pj);
        void display(void);
        string getString (unsigned i);
        void codeLevels(void);
        void addGinv(matvec::doubleMatrix& lhs,
                     unsigned startRow,
                     unsigned statCol,
                     double ratio);
        void addtoGinv(unsigned pos[], double q[], double v,
                       matvec::doubleMatrix& lhs,
                       unsigned startRow,
                       unsigned startCol,
                       double ratio);
};

#endif
```

4.4.2 Listing of MQTL.cpp

```
#include "MQTL.h"
#include "mmeMQTL.h"
#include "ped.h"
#include <matvec/session.h>
using namespace std;

Pedigree* MQTL::myPedPtr=0;

void MQTL::inputPDQ(char *filename){
  unsigned rec=0, rec1=0;
  QNode *ptr;
  double pPDQ, mPDQ;
  string indStr;
  ifstream pdqfile;
  pdqfile.open(filename);
  if(!pdqfile){
    cerr << "Couldn't open " << filename << endl;
    exit (-1);
  }
  resize(myPedPtr->size(),0);
  while (pdqfile >> indStr >> pPDQ >> mPDQ ){
          rec++;
          if(rec==1000){
                  cout<<".";
                  cout.flush();
                  rec1 += rec;
                  rec = 0;
          }
          Pedigree::iterator pedIt = myPedPtr->find(indStr);
          if (pedIt == myPedPtr->end()) {
                  cout << indStr << " in record " << rec1
                  << " of " << filename
                  <<" not found in pedigree file \n";
                  exit(1);
          }
          ptr = new QNode();
```

```
          ptr->pPDQ = pPDQ;
          ptr->mPDQ = mPDQ;
          (*this)[pedIt->second->ind-1] = ptr;
  }
  pdqfile.close();
  generateMQTLLevels();
  calcMQTLInbreeding();
}

void MQTL::generateMQTLLevels(void){
    MQTL::iterator it;
    count = 0;
    Pedigree::iterator pedIt;
    for (unsigned i=0;i<size();i++){
      if((*this)[i]==0){
         if (myPedPtr->pedVector[i]->sire==0 &&
             myPedPtr->pedVector[i]->dam==0) {
            QNode *ptr = new QNode();
            ptr->pPDQ = -1.0;
            ptr->mPDQ = -1.0;
            (*this)[i] = ptr;
         }
         else {
            cout << myPedPtr->pedVector[i]->ind_str
                 << " not in PDQ file \n";
            exit(-1);
         }
      }
    }
    for (unsigned i=0;i<size();i++){
        if ((*this)[i]->mLevel==0) {
                codeMQTL(i);
        }
    }
        codeLevels();
}
void MQTL::codeMQTL(unsigned i){
        if ((*this)[i]->mLevel) return;       // already coded
```

```
        int dam   = myPedPtr->pedVector[i]->dam;
        int sire  = myPedPtr->pedVector[i]->sire;

        if (sire==0) {
            (*this)[i]->pLevel = ++count;
        }
        else {
                codeMQTL(sire-1);

                if((*this)[i]->pPDQ==1){
                        (*this)[i]->pLevel = (*this)[sire-1]->mLevel;
                }
                else if ((*this)[i]->pPDQ==0){
                    (*this)[i]->pLevel = (*this)[sire-1]->pLevel;
                }
                else {
                        (*this)[i]->pLevel = ++count;
                }
        }

        if (dam==0) {
            (*this)[i]->mLevel = ++count;
        }
        else {
                codeMQTL(dam-1);

                if((*this)[i]->mPDQ==1){
                        (*this)[i]->mLevel = (*this)[dam-1]->mLevel;
                }
                else if ((*this)[i]->mPDQ==0){
                    (*this)[i]->mLevel = (*this)[dam-1]->pLevel;
                }
                else {
                        (*this)[i]->mLevel = ++count;
                }
        }

}
```

```

void MQTL::codeLevels(void){
  for (unsigned i=1;i<=count;i++){
    string str = getString(i);
    myRecoder.code(str);
  }
}

string MQTL::getString(unsigned i){
  ostringstream outputStrStream(ostringstream::out);
  outputStrStream << i;
  return outputStrStream.str();
}


void MQTL::display(){
        for (unsigned i=0;i<size();i++){
    cout << setw(10) <<  myPedPtr->pedVector[i]->ind_str
              << setw(5)  << (*this)[i]->pLevel
              << setw(5)  << (*this)[i]->mLevel
              << setw(10) << (*this)[i]->f
                  << endl;
  }
}

void MQTL::calcMQTLInbreeding(void){
     SpCij.clear();
     for (unsigned i=0;i<size();i++){
        if(myPedPtr->pedVector[i]->sire==0 ||
           myPedPtr->pedVector[i]->dam==0){
                (*this)[i]->f= 0.0;
        }
        else{
                unsigned ind = i+1;
                (*this)[i]->f = getMQTLrij(ind,ind,0,1);
        }
        }
}
```

```
double MQTL::getMQTLrij(unsigned i,
                        unsigned j,
                        unsigned pi, unsigned pj){
  if (i==0 || j==0){
    return 0.0;
  }
  unsigned vi = pi ? (*this)[i-1]->pLevel : (*this)[i-1]->mLevel;
  unsigned vj = pj ? (*this)[j-1]->pLevel : (*this)[j-1]->mLevel;
  float x = SpCij.retrieve_cij(vi,vj);
  if(x != -1.0) {
    return x;
  }
  unsigned old, young,oldp,youngp;
  if(i < j){
    old = i;
    oldp = pi;
    young = j;
    youngp = pj;
  }
  else if(j < i){
    old = j;
    oldp = pj;
    young = i;
    youngp = pi;
  }
  else if(pi==pj) {
    return 1.0;
  }
  else {
    old = i;
    oldp = pi;
    young = j;
    youngp = pj;
  }
  unsigned y_parent;
  double PDQym, PDQyp;
```

```
  if (youngp==1){
    y_parent = myPedPtr->pedVector[young-1]->sire;
    PDQym = (*this)[young-1]->pPDQ;
  }
  else{
    y_parent = myPedPtr->pedVector[young-1]->dam;
    PDQym    = (*this)[young-1]->mPDQ;
  }
  PDQyp = 1.0 - PDQym;
  x = PDQym*getMQTLrij(old,y_parent,oldp,0) +
      PDQyp*getMQTLrij(old,y_parent,oldp,1);
  SpCij.put_cij(vi,vj,x);
  return x;

void MQTL::addGinv(matvec::doubleMatrix& lhs,
                   unsigned startRow,
                   unsigned startCol,
                   double ratio){
    MQTL::iterator it;
    unsigned maxLevel=0;
    double q[3];
    unsigned pos[3];
    double v;
    for (unsigned i=0;i<size();i++){
       if ((*this)[i]->pLevel > maxLevel){
           if (myPedPtr->pedVector[i]->sire==0){ // founder
                pos[0] = 0;
                pos[1] = 0;
                pos[2] = (*this)[i]->pLevel;
                q[0]   = 0;
                q[1]   = 0;
                q[2]   = 1.0;
                v = 1.0;
                addtoGinv(pos,q,v,lhs,startRow,startCol,ratio);
           }
           else { // not founder
                unsigned dad = myPedPtr->pedVector[i]->sire;
                pos[0] = (*this)[dad-1]->mLevel;
```

```
                pos[1] = (*this)[dad-1]->pLevel;
                pos[2] = (*this)[i]->pLevel;
                q[0]   = -(*this)[i]->pPDQ;
                q[1]   = -(1+q[0]);
                q[2]   = 1.0;
                v      = 1 -   q[0]*q[0]
                         -   q[1]*q[1]
                         - 2*q[0]*q[1]*(*this)[dad-1]->f;
                addtoGinv(pos,q,1.0/v,lhs,startRow,startCol,ratio);
           }
                maxLevel = pos[2];
        }
        if ((*this)[i]->mLevel > maxLevel){
           if (myPedPtr->pedVector[i]->dam==0) { // founder
                pos[0] = 0;
                pos[1] = 0;
                pos[2] = (*this)[i]->mLevel;
                q[0]   = 0;
                q[1]   = 0;
                q[2]   = 1.0;
                v = 1.0;
                addtoGinv(pos,q,v,lhs,startRow,startCol,ratio);
           }
           else { // not founder
                unsigned mom = myPedPtr->pedVector[i]->dam;
                pos[0] = (*this)[mom-1]->mLevel;
                pos[1] = (*this)[mom-1]->pLevel;
                pos[2] = (*this)[i]->mLevel;
                q[0]   = -(*this)[i]->mPDQ;
                q[1]   = -(1+q[0]);
                q[2]   = 1.0;
                v      = 1 -   q[0]*q[0]
                         -   q[1]*q[1]
                         - 2*q[0]*q[1]*(*this)[mom-1]->f;
                addtoGinv(pos,q,1.0/v,lhs,startRow,startCol,ratio);
           }
           maxLevel = pos[2];
        }
```

```
   }
}

void MQTL::addtoGinv(unsigned pos[], double q[], double v,
                     matvec::doubleMatrix& lhs,
                     unsigned startRow,
                     unsigned startCol,
                     double ratio){
     double vi,vj;
     unsigned ii,jj;
     for (unsigned i=0;i<3;i++){
        if(pos[i]){
           ii = startRow + pos[i] - 1;
           vi = q[i]*v;
           for (unsigned j=0;j<3;j++){
               if(pos[j]) {
                  vj = q[j];
                  jj = startCol + pos[j] - 1;
                  if(MME::solMethod=="direct"){
                     lhs[ii][jj] += ratio*vi*vj;
                  }
                  else{
                     MME::res[ii] += ratio*vi*vj * (*MME::vec)[jj];
                     if(ii==jj) MME::diag[ii] += ratio*vi*vj;
                  }
               }
           }
        }
    }
}
```

4.4.3 Listing of mmeMQTL.h

```
#ifndef MME_H
#define MME_H
#include <fstream>
#include <iostream>
#include <iomanip>
```

```
#include <string>
#include <sstream>
#include <stdarg.h>
#include <stdlib.h>
#include <cmath>
#include <map>
#include <matvec/doublematrix.h>
#include <matvec/vector.h>
#include <matvec/session.h>
#include "util.h"
#include "ped.h"
#include "MQTL.h"


// classes for multiple trait, mixed models with iteration on data

using namespace std;

class TermData{
public:
  double value;
  unsigned level;
};

class DataNode{
public:
  vector<TermData > trmVec;
  vector<double>    depVec;
};

class MME;

class ModelTerm{
public:
        unsigned start;
        unsigned trait;
        string name;
        string depVarName;
```

```
        static MME *myMMEPtr;
        Recoder<string>* myRecoderPtr;
        MQTL* myMQTLPtr;
        bool secondMQTLEffect;
        vector<unsigned> factors;

        unsigned code(string str){return myRecoderPtr->code(str);}
        unsigned nLevels(){return myRecoderPtr->size();}
        void putFactors(string str);
        string   getTermString();
        unsigned getTermLevel (){
                return code(getTermString());
        }
        double   getTermValue ();
};

class CovBlock {
public:
        vector<ModelTerm*> modelTrmPtrVec;
        matvec::doubleMatrix Var, Vari;
        Pedigree* pedPtr;
        MQTL* myMQTLPtr;
        CovBlock(void){pedPtr = 0;}
        CovBlock(string str, matvec::doubleMatrix V){
                Var = V;
                buildModelTrmVec(str);
        }
        CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
                Var = V;
                pedPtr = &P;
                myMQTLPtr = 0;
                buildModelTrmVec(str);
        }
        CovBlock(MQTL &mQTL){
                pedPtr = 0;
                myMQTLPtr = &mQTL;
        }
        void buildModelTrmVec(string str);
```

```
        void addGinv(void);
};


class MME {
 private:
  void putModel(string str);
public:
    static string solMethod;
        static matvec::Vector<double> *vec, diag, res;

        string fileName;
        Tokenizer colType;
        Tokenizer colName;
        Tokenizer depVar;
        Tokenizer colData;
        unsigned numCols;
        vector <ModelTerm> modelTrmVec;
        vector <CovBlock>  covBlockVec;
        vector <DataNode> dataVec;
        vector<MQTL*> MQTLVec;
        unsigned numTerms, numTraits;
        unsigned mmeSize;
        matvec::doubleMatrix lhs, R, Ri;
        matvec::Vector<double> rhs, sol, tempSol;

        void putColNames(string str);
        void putColTypes(string str);
        void putModels(string str);
        void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
                CovBlock covBlock(str,V,P);
                covBlockVec.push_back(covBlock);
        }
        void putVarCovMatrix(string str, matvec::doubleMatrix V){
                CovBlock covBlock(str,V);
                covBlockVec.push_back(covBlock);
        }
        void putVarCovMatrix(MQTL &Q){
```

```
                CovBlock covBlock(Q);
                covBlockVec.push_back(covBlock);
        }
        void initSetup();
        void inputData();
        void displayData();
        static double getDouble(string& Str);
        void calcStarts();
        void getDirectSolution();
        void getJacobiSolution(double p);
        void getCGSolution();
        void getPCCGSolution();

        void mmeTimes(matvec::Vector<double>& x);
        void calcWPW();
        void addGinv();
        void display();

        void putMQTL(MQTL& Q){MQTLVec.push_back(&Q);};
        void mergeMQTLLevelsBy(string mergeStr);
        void addColNames(string str);
        void addColTypes(string str);
        void putMQTLStuffInModelTerms(MQTL& mQTL);
        void processMQTL(void);
};
#endif

```

4.4.4 Listing of mmeMQTL.cpp

```
/*
 *  mme1.cpp
 *  C++WkShp
 *
 *  Created by Rohan Fernando on 5/6/05.
 *  Copyright 2005,   All rights reserved.
 *
 */
```

```
#include "mmeMQTL.h"

MME* ModelTerm::myMMEPtr;
string MME::solMethod;
matvec::Vector<double> *MME::vec, MME::diag, MME::res;

 void ModelTerm::putFactors(string str){
  Tokenizer tokens;
  string sep("*");
  tokens.getTokens(str,sep);
  factors.clear();
  for (unsigned i=0;i<tokens.size();i++){
    unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
    if (factorIndex == -1){
      cerr <<"Independent Variable "
           << tokens[i]
           << " not in list of column names \n";
      exit(-1);
    }
    else {
      factors.push_back(factorIndex);
    }
  }
}


string ModelTerm::getTermString(){
        unsigned numFactors = factors.size();
        string trmStr;
        unsigned factorIndex = factors[0];
        if(myMMEPtr->colType[factorIndex]=="COV"){
                trmStr = myMMEPtr->colName[factorIndex];
        }
        else {
                trmStr = myMMEPtr->colData[factorIndex];
        }
        for (unsigned i=1;i<numFactors;i++){
```

```
                factorIndex = factors[i];
                if(myMMEPtr->colType[factorIndex]=="COV"){
                        trmStr += "*" + myMMEPtr->colName[factorIndex];
                }
                else{
                        trmStr += "*" + myMMEPtr->colData[factorIndex];
                }
        }
        return trmStr;
}

double ModelTerm::getTermValue(){
        unsigned numFactors = factors.size();
        double value = 1.0;
        for (unsigned i=0;i<numFactors;i++){
                unsigned factorIndex = factors[i];
                if (myMMEPtr->colType[factorIndex]=="COV"){
                        string covStr = myMMEPtr->colData[factorIndex];
                        value *= MME::getDouble(covStr);
                }
        }
        return value;
}

void MME::putColNames(string str){
  string sep(" ");
  str = "intercept " + str;
  colName.getTokens(str,sep);
  numCols = colName.size();
}

void MME::putColTypes(string str){
  string sep(" ");
  str = "CLASS " + str;
  colType.getTokens(str,sep);
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
```

```
  }
  unsigned n = 0;
  for (unsigned i=0;i<numCols;i++){
        if (colType[i] == "DEP") {
                depVar.push_back(colName[i]);
                n++;
        }
  }
  numTraits = n;
}

void MME::putModels(string str){
        ModelTerm::myMMEPtr = this;
        Tokenizer models;
        string sep =";";
        models.getTokens(str,sep);
        for (unsigned i=0; i<models.size();i++){
                putModel(models[i]);
        }
        for(unsigned i=0;i<MQTLVec.size();i++){
                putMQTLStuffInModelTerms(*MQTLVec[i]);
  }
}

void MME::putModel(string str){
   string sep(" =+");
   Tokenizer modelTokens;
   modelTokens.getTokens(str,sep);
   unsigned nTokens = modelTokens.size();
   int depVarIndex = colName.getIndex(modelTokens[0]);
   if (depVarIndex == -1){
     cerr << "Dependent Variable "
          << modelTokens[0]
          << " not in list of column names \n";
     exit (-1);
   }
   ModelTerm modelTrm;
   modelTrm.secondMQTLEffect = false;
```

```
   modelTrm.myMQTLPtr = 0;
   modelTrm.depVarName = modelTokens[0];
   modelTrm.trait = depVar.getIndex(modelTokens[0]);
   for (unsigned i=1;i<nTokens;i++){
           modelTrm.myRecoderPtr = new Recoder<string>;
           modelTrm.name = modelTokens[i];
           modelTrm.putFactors(modelTrm.name);
           modelTrmVec.push_back(modelTrm);
   }
}

void MME::inputData(){
    DataNode dataNode;
    numTerms = modelTrmVec.size();
    dataNode.trmVec.resize(numTerms);
    dataNode.depVec.resize(numTraits);
    ifstream datafile;
    datafile.open(fileName.c_str());
    if(!datafile) {
        cerr << "Couldn't open data file: " << fileName << endl;
        exit (-1);
    }
    unsigned linewidth = 1024;
    char *line = new char [linewidth];
    string sep(" ");
    while (datafile.getline(line,linewidth)){
      string inputStr(line);
      inputStr = "--- " + inputStr;
      colData.getTokens(inputStr,sep);
      unsigned j=0;
      for (unsigned i=0;i<numCols;i++){
         if (colType[i]=="DEP") {
            dataNode.depVec[j++] = getDouble(colData[i]);
         }

      }
      for (unsigned i=0;i<numTerms;i++){
         dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
```

```
          if (modelTrmVec[i].myMQTLPtr){
             unsigned tr = modelTrmVec[i].trait;
             dataNode.trmVec[i].value = modelTrmVec[i].myMQTLPtr->regCoeff[tr];
          }
          else {
             dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
         }
       }
       dataVec.push_back(dataNode);
     }
}

double MME::getDouble(string& Str) {
  istringstream inputStrStream(Str.c_str());
  double val;
  inputStrStream >> val;
  return val;
}

void MME::calcStarts(){
    unsigned prevI =0,  maxI = 0;
    modelTrmVec[0].start = 0;
    for (unsigned i=1;i<numTerms;i++){
        if (modelTrmVec[i].myMQTLPtr) {
            if(modelTrmVec[i].myMQTLPtr->myStart == -1){
               modelTrmVec[i].start = modelTrmVec[prevI].start
                                    + modelTrmVec[prevI].nLevels();
               prevI = i;
               modelTrmVec[i].myMQTLPtr->myStart = modelTrmVec[i].start;
            }
            else {
               modelTrmVec[i].start = modelTrmVec[i].myMQTLPtr->myStart;
            }
        }
        else {
            modelTrmVec[i].start = modelTrmVec[prevI].start
                                 + modelTrmVec[prevI].nLevels();
            prevI = i;
```

```
        }
        if (modelTrmVec[i].start > modelTrmVec[maxI].start){
            maxI = i;
        }
    }
    mmeSize = modelTrmVec[maxI].start + modelTrmVec[maxI].nLevels();
}

void MME::calcWPW(){
    unsigned ii,jj,ti,tj;
    double vi,vj,tr_value;
    if(solMethod!="direct"){
        diag.resize(mmeSize,0.0);
        res.resize(mmeSize,0.0);
    }
    rhs.resize(mmeSize,0.0);
    Ri = R.inv();
    for (unsigned i=0;i<dataVec.size();i++){
        for (unsigned mi=0;mi<numTerms;mi++){
           ii = modelTrmVec[mi].start + dataVec[i].trmVec[mi].level - 1;
           ti =  modelTrmVec[mi].trait;
           vi = dataVec[i].trmVec[mi].value;
           for (unsigned k=0;k<numTraits;k++){
              rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
           }
           for (unsigned mj=0;mj<numTerms;mj++){
              jj = modelTrmVec[mj].start + dataVec[i].trmVec[mj].level - 1;
              tj =  modelTrmVec[mj].trait;
              vj = dataVec[i].trmVec[mj].value;
              if (solMethod=="direct"){
                 lhs[ii][jj] += vi*Ri[ti][tj]*vj;
              }
              else {
                 res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
                 if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
              }
           }
        }
```

```
    }
}

void MME::addGinv(){
   for (unsigned i=0;i<covBlockVec.size();i++){
       covBlockVec[i].addGinv();
   }
}

void MME::initSetup(){
  inputData();
  calcStarts();
  if(solMethod=="direct"){
    lhs.resize(mmeSize,mmeSize,0.0);
  }
  else {
    sol.resize(mmeSize,0.0);
  }
}


void MME::getDirectSolution(){
        solMethod = "direct";
        initSetup();
        calcWPW();
        addGinv();
        sol = lhs.ginv0()*rhs;
}

void MME::display(){
    if (solMethod=="direct"){
       cout << "LHS " << endl;
       for (unsigned i = 0;i<mmeSize;i++){
           for (unsigned j = 0;j<mmeSize;j++){
                   cout << setw(8) << setprecision (3)
                        << setiosflags (ios::right | ios::fixed)
                        << lhs[i][j] <<" ";
           }
```

```
            cout << endl;
        }
    }
    cout << "RHS " << endl;
    cout << rhs << endl;
    for (unsigned i=0;i<modelTrmVec.size();i++){
        cout << "Solutions for " << modelTrmVec[i].name
             << ", Trait: " << modelTrmVec[i].depVarName << endl;
        Recoder<string>::iterator it;
        for (it=modelTrmVec[i].myRecoderPtr->begin();
             it!=modelTrmVec[i].myRecoderPtr->end();
             it++){
                unsigned ii = modelTrmVec[i].start + it->second - 1;
                cout << setw(10) << it->first << " " << sol[ii] << endl;
        }
    }
}

void MME::getJacobiSolution(double p){
        solMethod = "jacobi";
        initSetup();
        mmeTimes(sol);  // result goes into res, also creates rhs and diag
        matvec::Vector<double> resid = (rhs - res);
        tempSol = resid/diag + sol;
        double diff = resid.sumsq();
        unsigned iter = 0;
        while(diff/mmeSize > .0000001 && ++iter<200){
                sol = p*tempSol + (1-p)*sol;
                cout <<"Iteration : "
                     << iter <<" diff = "
                     <<diff/mmeSize << endl << endl;
                mmeTimes(sol);
                resid = (rhs - res);
                tempSol = resid/diag + sol;
                diff = resid.sumsq();
        }
        cout << resid << endl;
}
```

```
void MME::getCGSolution(){
        solMethod = "cg";
        initSetup();
        mmeTimes(sol); // result goes into res, also creates rhs and diag
        matvec::Vector<double> resid = (res-rhs);
        matvec::Vector<double> d = resid;
        double oldDiffSq;
        double newDiffSq = resid.sumsq();
        unsigned iter = 0;
        while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
                cout <<"Iteration : "
                     << iter
                     <<" diff = "
                     <<newDiffSq/mmeSize << endl << endl;
                mmeTimes(d);
                double alpha = -newDiffSq/(res*d).sum();
                sol += alpha*d;
                if (iter%10){
                        resid += alpha*res;
                }
                else {
                        mmeTimes(sol);
                        resid = (res-rhs);
                }
                oldDiffSq = newDiffSq;
                newDiffSq = resid.sumsq();
                double beta = -newDiffSq/oldDiffSq;
                d = resid - beta*d;
        }
        cout << resid << endl;
}

void MME::getPCCGSolution(){
        solMethod = "pccg";
        initSetup();
        mmeTimes(sol); // result goes into res, also creates rhs and diag
        matvec::Vector<double> resid = (res - rhs);
```

```
        matvec::Vector<double> d = resid/diag;
        double oldDiffSq;
        double newDiffSq = (resid*d).sum();
        unsigned iter = 0;
        while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
                cout <<"Iteration : "
                     << iter <<" diff = "
                     <<newDiffSq/mmeSize
                     << endl << endl;
                mmeTimes(d);
                double alpha = -newDiffSq/(res*d).sum();
                sol += alpha*d;
                if (iter % 10){
                        resid += alpha*res;
                }
                else {
                        mmeTimes(sol);
                        resid = (res - rhs);
                }
                matvec::Vector<double> s = resid/diag;
                oldDiffSq = newDiffSq;
                newDiffSq = (resid*s).sum();
                double beta = -newDiffSq/oldDiffSq;
                d = s - beta*d;
        }
        cout << resid << endl;
}

void MME::mmeTimes(matvec::Vector<double>& x){
  vec = &x;
  calcWPW();
  addGinv();
}

void MME::mergeMQTLLevelsBy(string mergeStr){
  unsigned numColsInFile = numCols - 1;
  int indexMergeStr = colName.getIndex(mergeStr) - 1;
  if(indexMergeStr == -1){
```

```
  cerr << mergeStr <<" not in Column Names \n";
  exit(-1);
}
ifstream datafile;
ofstream outfile;
string outFileName = fileName + ".mrgMQTL";
datafile.open(fileName.c_str());
if(!datafile) {
  cerr << "Couldn't open data file: " << fileName << endl;
  exit (-1);
}
outfile.open(outFileName.c_str());
if(!outfile) {
  cerr << "Couldn't open file: " << outFileName << endl;
  exit (-1);
}
size_t linewidth = 1024;
char *line = new char [linewidth];
string sep(" ");
unsigned lineNumber=0;
while (datafile.getline(line,linewidth)){
  lineNumber++;
  string inputStr(line);
  colData.getTokens(inputStr,sep);
  unsigned n = colData.size();
  if (n != numColsInFile){
    cerr << "Line " << lineNumber << " of data file has " << n <<" columns" << e
    cerr << numColsInFile <<" expected " << endl;
    exit(-1);
  }
  string mergeVar = colData[indexMergeStr];
  int mergeID = MQTLVec[0]->myPedPtr->coder.code(mergeVar);
  if (mergeID>MQTLVec[0]->myPedPtr->size()){
    cerr << "Line " << lineNumber << " of data file has merge string "<<mergeVar
    cerr << "which is not in Pedigree file\n";
  }
  outfile << inputStr << " ";
  for (unsigned i=0;i<MQTLVec.size();i++){
```

```
      outfile << (*MQTLVec[i])[mergeID-1]->pLevel << " "
              << (*MQTLVec[i])[mergeID-1]->mLevel << " ";
    }
    outfile << endl;
  }
  datafile.close();
  outfile.close();
  fileName = outFileName;
}

void MME::addColNames(string str){
  string sep(" ");
  Tokenizer tokens;
  tokens.getTokens(str,sep);
  Tokenizer::iterator it;
  for(it=tokens.begin();it!=tokens.end();it++){
    colName.push_back(*it);
  }
  numCols = colName.size();
}

void MME::addColTypes(string str){
  string sep(" ");
  Tokenizer tokens;
  tokens.getTokens(str,sep);
  Tokenizer::iterator it;
  for(it=tokens.begin();it!=tokens.end();it++){
    colType.push_back(*it);
  }
  if (numCols!=colType.size()){
    cerr <<"number of column names and column types do not match\n";
    exit (-1);
  }
}

void MME::putMQTLStuffInModelTerms(MQTL& mQTL){
    CovBlock covBlock(mQTL);
    Tokenizer names;
```

```
    string sep = " ,";
    names.getTokens(mQTL.MQTLNames,sep);
    string firstMQTL  = names[0];
    string secondMQTL = names[1];
    for (unsigned i=0;i<modelTrmVec.size();i++){
       if(modelTrmVec[i].name==firstMQTL || modelTrmVec[i].name==secondMQTL){
          modelTrmVec[i].myMQTLPtr = &mQTL;
          delete modelTrmVec[i].myRecoderPtr;
          modelTrmVec[i].myRecoderPtr = &mQTL.myRecoder;
       }
       if(modelTrmVec[i].name==secondMQTL){
          modelTrmVec[i].secondMQTLEffect = true;

       }
    }
}

void CovBlock::buildModelTrmVec(string str){
    string sep(" ");
    Tokenizer modelTokens;
    modelTokens.getTokens(str,sep);
    unsigned nTokens = modelTokens.size();
    unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
    for (unsigned i=0;i<nTokens;i++){
       for (unsigned j=0;j<numModelTrms;j++){
          if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
              modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
              if (pedPtr){
                 delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
                 ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr = &pedPtr->code
              }
          }
       }
   }
}

void CovBlock::addGinv(void){
    if(myMQTLPtr){
```

```
        myMQTLPtr->addGinv(ModelTerm::myMMEPtr->lhs,
                           myMQTLPtr->myStart,
                           myMQTLPtr->myStart,
                           1.0/myMQTLPtr->variance);
        return;
    }
    Vari = Var.inv();
    unsigned n = modelTrmPtrVec.size();
    for (unsigned i=0;i<n;i++){
      ModelTerm* mtermiPtr = modelTrmPtrVec[i];
      unsigned starti = mtermiPtr->start;
      for (unsigned j=0;j<n;j++){
         ModelTerm* mtermjPtr = modelTrmPtrVec[j];
         unsigned startj = mtermjPtr->start;
         if (pedPtr){
           pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
         }

         else{
            unsigned numLevels = mtermiPtr->nLevels();
            for (unsigned k=0;k<numLevels;k++){
              unsigned ii = starti + k;
              unsigned jj = startj + k;
              if (MME::solMethod=="direct") {
                 ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
              }
              else {
                 MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
                 if (ii==jj)  MME::diag[ii] += Vari[i][j];
              }
            }
         }
      }
    }
}
```