

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

1	Classes for MME	3
1.1	Strategy for building normal equations	3
1.1.1	Contributions to LHS	3
1.1.2	Contributions to RHS	4
1.2	The MME classes	4
1.2.1	Method MME::calcWPW	7
1.2.2	The main program	8
1.2.3	Method MME::inputData	10
1.2.4	Method ModelTerm::getTermString	11
1.2.5	Method ModelTerm::getTermValue	12
1.2.6	Method MME::calcStarts	13
1.2.7	Method MME::getSolution	13
1.2.8	Method MME::display	13
1.2.9	Listing of twowayMME.cpp	15
1.3	Extensions to accommodate multiple trait mixed models	22
1.3.1	Listing of multiTraitMixedMME.cpp	25
2	MME from Model String	36
2.1	Generating ModelTerm objects	36
2.1.1	Method MME::putModels	36
2.1.2	Method MME::putModel	37
2.1.3	Method ModelTerm::putFactors	38
2.2	Generating CovBlock objects	39
2.2.1	MME::putVarCovMatrix methods	39
2.2.2	Method CovBlock::buildModelTrmVec	40
2.3	The main program	41
2.4	Listing of additive2TrMSMME.cpp	44

3	Iterative Solvers for MME	56
3.1	The Jacobi method	56
3.2	The conjugate gradient method	57
3.3	Preconditioned conjugate gradient method	61
3.4	Iteration on data	62
3.5	Program changes for iteration on data	63
3.6	Listings of IOD programs	65
3.6.1	ped.h	65
3.6.2	ped.cpp	67
3.6.3	mmel.h	75
3.6.4	mmel.cpp	79
3.6.5	iterDataMSMME.cpp	90
4	Marker Assisted BLUP	92
4.1	Single trait models	92
4.2	Multiple trait models	94
4.3	Classes and methods for MABLUP	94
4.3.1	The main program	95
4.3.2	Method MME::putModels	97
4.3.3	Method MME::putMQTLStuffInModelTerms	97
4.3.4	Method MME::calcStarts	98
4.3.5	Other changes	99
4.4	Listing of MAPLUP programs	100
4.4.1	Listing of MQTL.h	100
4.4.2	Listing of MQTL.cpp	102
4.4.3	Listing of mmeMQTL.h	109
4.4.4	Listing of mmeMQTL.cpp	113

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

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

where \mathbf{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 \mathbf{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 \mathbf{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:

```

54 class MME {
55 public:
56     string fileName;
57     Tokenizer colType;
58     Tokenizer colName;
59     Tokenizer colData;
60     unsigned numCols;
61     unsigned depCol;
62     vector <ModelTerm> modelTrmVec;
63     vector <DataNode> dataVec;
64     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         }

```

```

50         double   getTermValue ();
51     };

```

“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:

```

19     class TermData{
20     public:
21         double value;
22         unsigned level;
23     };
24
25     class DataNode{
26     public:
27         vector<TermData > trmVec;
28         double depVar;
29     };

```

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 \mathbf{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 calculated for each element of `dataVec`, which is a prerequisite for this method. These are calculated in the `MME::inputData`. 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::inputData` 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 `Recorder` 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`

```

80 string ModelTerm::getTermString(){
81     unsigned numFactors = factors.size();
82     string trmStr;
83     unsigned factorIndex = factors[0];
84     if(myMMEPtr->colType[factorIndex]=="COV"){
85         trmStr = myMMEPtr->colName[factorIndex];
86     }
87     else {
88         trmStr = myMMEPtr->colData[factorIndex];
89     }
90     for (unsigned i=1;i<numFactors;i++){
91         factorIndex = factors[i];
92         if(myMMEPtr->colType[factorIndex]=="COV"){
93             trmStr += "*" + myMMEPtr->colName[factorIndex];
94         }
95         else{

```

```

96         trmStr += "*" + myMMEPtr->colData[factorIndex];
97     }
98 }
99     return trmStr;
100 }
```

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`

```

102 double ModelTerm::getTermValue(){
103     unsigned numFactors = factors.size();
104     double value = 1.0;
105     for (unsigned i=0;i<numFactors;i++){
106         unsigned factorIndex = factors[i];
107         if (myMMEPtr->colType[factorIndex]=="COV"){
108             string covStr = myMMEPtr->colData[factorIndex];
109             value *= MME::getDouble(covStr);
110         }
111     }
112     return value;
113 }
```

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 $\text{value} * x$.

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

```

209 void MME::display(){
210     cout << "LHS " << endl;
211     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

```
1  #include <fstream>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <sstream>
6  #include <stdarg.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <map>
10 #include <matvec/doublematrix.h>
11 #include <matvec/vector.h>
12 #include <matvec/session.h>
13 #include "util.h"
14
15 // Single trait, fixed effects models
16
17 using namespace std;
18
19 class TermData{
20 public:
21     double value;
22     unsigned level;
23 };
24
25 class DataNode{
26 public:
27     vector<TermData > trmVec;
28     double depVar;
29 };
30
31 class MME;
32
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
45     unsigned code(string str){return myRecoderPtr->code(str);}
46     unsigned nLevels(){return myRecoderPtr->size();}
47     string  getTermString();
48     unsigned getTermLevel (){
49         return code(getTermString());
50     }
51     double  getTermValue ();
52 };
53
54 class MME {
55 public:
56     string fileName;
57     Tokenizer colType;
58     Tokenizer colName;
59     Tokenizer colData;
60     unsigned numCols;
61     unsigned depCol;
62     vector <ModelTerm> modelTrmVec;
63     vector <DataNode> dataVec;
64     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 };
79 string ModelTerm::getTermString(){
80     unsigned numFactors = factors.size();
81     string trmStr;
82     unsigned factorIndex = factors[0];
83     if(myMMEPtr->colType[factorIndex]=="COV"){
84         trmStr = myMMEPtr->colName[factorIndex];
85     }
86     else {
87         trmStr = myMMEPtr->colData[factorIndex];
88     }
89     for (unsigned i=1;i<numFactors;i++){
90         factorIndex = factors[i];
91         if(myMMEPtr->colType[factorIndex]=="COV"){
92             trmStr += "*" + myMMEPtr->colName[factorIndex];
93         }
94         else{
95             trmStr += "*" + myMMEPtr->colData[factorIndex];
96         }
97     }
98     return trmStr;
99 }
100
101 double ModelTerm::getTermValue(){
102     unsigned numFactors = factors.size();
103     double value = 1.0;
104     for (unsigned i=0;i<numFactors;i++){
105         unsigned factorIndex = factors[i];
106         if (myMMEPtr->colType[factorIndex]=="COV"){
107             string covStr = myMMEPtr->colData[factorIndex];
108             value *= MME::getDouble(covStr);
109         }
110     }
111     return value;
112 }
```

```
113
114 void MME::putColNames(string str){
115     string sep(" ");
116     colName.getTokens(str,sep);
117     numCols = colName.size();
118 }
119
120 void MME::putColTypes(string str){
121     string sep(" ");
122     colType.getTokens(str,sep);
123     if (numCols!=colType.size()){
124         cerr <<"number of column names and column types do not match\n";
125         exit (-1);
126     }
127     for (unsigned i=0;i<numCols;i++){
128         if (colType[i] == "DEP") {
129             depCol = i;
130             return;
131         }
132     }
133     cout << "Could not find dependent variable \n";
134     exit(1);
135 }
136
137 void MME::inputData(){
138     DataNode dataNode;
139     numTerms = modelTrmVec.size();
140     dataNode.trmVec.resize(numTerms);
141     ifstream datafile;
142     datafile.open(fileName.c_str());
143     if(!datafile) {
144         cerr << "Couldn't open data file: " << fileName << endl;
145         exit (-1);
146     }
147     unsigned linewidth = 1024;
148     char *line = new char [linewidth];
149     string sep(" ");
150     while (datafile.getline(line,linewidth)){
```

```

151     string inputStr(line);
152     colData.getTokens(inputStr,sep);
153     dataNode.depVar = getDouble(colData[depCol]);
154     for (unsigned i=0;i<numTerms;i++){
155         dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
156         dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
157     }
158     dataVec.push_back(dataNode);
159 }
160 }
161
162 double MME::getDouble(string& Str) {
163     istringstream inputStrStream(Str.c_str());
164     double val;
165     inputStrStream >> val;
166     return val;
167 }
168
169 void MME::calcStarts(){
170     modelTrmVec[0].start = 0;
171     for (unsigned i=1;i<numTerms;i++){
172         modelTrmVec[i].start = modelTrmVec[i-1].start
173             + modelTrmVec[i-1].nLevels();
174     }
175     mmeSize = modelTrmVec[numTerms-1].start
176         + modelTrmVec[numTerms-1].nLevels();
177 }
178
179 void MME::calcWPW(){
180     unsigned poski,poskj;
181     double vki,vkj,tr_value;
182     rhs.resize(mmeSize,0.0);
183     lhs.resize(mmeSize,mmeSize,0.0);
184     for (unsigned k=0;k<dataVec.size();k++){
185         for (unsigned i=0;i<numTerms;i++){
186             poski = modelTrmVec[i].start
187                 + dataVec[k].trmVec[i].level - 1;
188             vki = dataVec[k].trmVec[i].value;

```

```

189         tr_value = dataVec[k].depVar;
190         rhs[poski] += vki*tr_value;
191         for (unsigned j=0;j<numTerms;j++){
192             poskj = modelTrmVec[j].start
193                 + dataVec[k].trmVec[j].level - 1;
194             vkj = dataVec[k].trmVec[j].value;
195             lhs[poski][poskj] += vki*vkj;
196         }
197     }
198 }
199 }
200
201 void MME::getSolution(){
202     inputData();
203     calcStarts();
204     calcWPW();
205     sol = lhs.ginv0()*rhs;
206 }
207
208 void MME::display(){
209     cout << "LHS " << endl;
210     for (unsigned i = 0;i<mmeSize;i++){
211         for (unsigned j = 0;j<mmeSize;j++){
212             cout << setw(10)
213                 << setprecision (4)
214                 << setiosflags (ios::right | ios::fixed)
215                 << lhs[i][j] <<" ";
216         }
217         cout << endl;
218     }
219     cout << "RHS " << endl;
220     cout << rhs << endl;
221     for (unsigned i=0;i<modelTrmVec.size();i++){
222         cout << "Solutions for " << modelTrmVec[i].name << endl;
223         Recoder<string>::iterator it;
224         for (it=modelTrmVec[i].myRecoderPtr->begin();
225             it!=modelTrmVec[i].myRecoderPtr->end();
226             it++){

```

```
227         unsigned ii = modelTrmVec[i].start + it->second - 1;
228         cout << setw(10)
229              << it->first << " "
230              << sol[ii] << endl;
231     }
232 }
233 }
234
235
236 MME* ModelTerm::myMMEPtr;
237
238 int main() {
239     try{
240         matvec::SESSION.initialize("matvec_trash");
241         MME mme;
242         ModelTerm::myMMEPtr = &mme;
243         mme.fileName = "Data/twoway.dat";
244         mme.putColNames("A B y");
245         mme.putColTypes("CLASS CLASS DEP");
246
247         ModelTerm mtermA;
248         mtermA.name = "A";
249         mtermA.factors.resize(1);
250         mtermA.factors[0] = 0;
251         mme.modelTrmVec.push_back(mtermA);
252
253         ModelTerm mtermB;
254         mtermB.name = "B";
255         mtermB.factors.resize(1);
256         mtermB.factors[0] = 1;
257         mme.modelTrmVec.push_back(mtermB);
258
259         ModelTerm mtermAB;
260         mtermAB.name = "A*B";
261         mtermAB.factors.resize(2);
262         mtermAB.factors[0] = 0;
263         mtermAB.factors[1] = 1;
264         mme.modelTrmVec.push_back(mtermAB);
```

```

265
266         mme.getSolution();
267         mme.display();
268     }
269     catch (matvec::exception &ex) {
270         cerr << ex.what() << "\n";
271         exit(1);
272     }
273     catch (...) {
274         cerr << "other exceptions were caught\n";
275         exit(1);
276     }
277 }

```

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`.

```

26 class DataNode{
27 public:
28     vector<TermData > trmVec;
29     vector<double>    depVec;
30 };

```

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

```

172     for (unsigned i=0;i<numCols;i++){
173         if (colType[i]=="DEP") {
174             dataNode.depVec[j++] = getDouble(colData[i]);
175         }

```

```
176
177     }
```

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:

```
204 void MME::calcWPW(){
205     unsigned ii,jj,ti,tj;
206     double vi,vj,tr_value;
207     rhs.resize(mmeSize,0.0);
208     lhs.resize(mmeSize,mmeSize,0.0);
209     Ri = R.inv();
210     for (unsigned i=0;i<dataVec.size();i++){
211         for (unsigned mi=0;mi<numTerms;mi++){
212             ii = modelTrmVec[mi].start
213                 + dataVec[i].trmVec[mi].level - 1;
214             ti = modelTrmVec[mi].trait;
215             vi = dataVec[i].trmVec[mi].value;
216             for (unsigned k=0;k<numTraits;k++){
217                 rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
218             }
219             for (unsigned mj=0;mj<numTerms;mj++){
220                 jj = modelTrmVec[mj].start
221                     + dataVec[i].trmVec[mj].level - 1;
222                 tj = modelTrmVec[mj].trait;
223                 vj = dataVec[i].trmVec[mj].value;
224                 lhs[ii][jj] += vi*Ri[ti][tj]*vj;
225             }
226         }
227     }
228 }
```

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


```

55 class CovBlock {
56 public:
57     vector<ModelTerm*> modelTrmPtrVec;
58     matvec::doubleMatrix Var, Vari;
59     Pedigree* pedPtr;
60     CovBlock(void){pedPtr = 0;}
61
62     void addGinv(void);
63 };

```

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, $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$, of k correlated random effects with

$$\text{Var}(\mathbf{u}) = \Sigma = \mathbf{G} \otimes \mathbf{A}. \quad (1.2)$$

The inverse of this covariance matrix is

$$\Sigma^{-1} = \mathbf{G}^{-1} \otimes \mathbf{A}^{-1}. \quad (1.3)$$

To obtain the LHS of the MME we need to add Σ^{-1} to the LHS of the normal equations at the position corresponding to \mathbf{u} . In the method `CovBlock::addGinv`, this is done by adding $\mathbf{A}^{-1}g^{ij}$ to the LHS of the normal equations at the position corresponding to random effects \mathbf{u}_i and \mathbf{u}_j , for all combinations of traits i and j , where g^{ij} is element ij of \mathbf{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 \mathbf{A} is the additive relationship matrix, the `Pedigree::addAinv` method is used to add $\mathbf{A}^{-1}g^{ij}$ to the normal equations (line 240). On the other hand, if \mathbf{A} is the identity matrix, the adding of $\mathbf{I}g^{ij}$ 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++){

```

```

237     ModelTerm* mtermjPtr = modelTrmPtrVec[j];
238     unsigned startj = mtermjPtr->start;
239     if (pedPtr){
240         pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
241     }
242     else{
243         unsigned numLevels = mtermiPtr->nLevels();
244         for (unsigned k=0;k<numLevels;k++){
245             ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
246         }
247     }
248 }
249 }
250 }

```

1.3.1 Listing of multiTraitMixedMME.cpp

```

1  #include <fstream>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <sstream>
6  #include <stdarg.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <map>
10 #include <matvec/doublematrix.h>
11 #include <matvec/vector.h>
12 #include <matvec/session.h>
13 #include "util.h"
14 #include "ped.h"
15
16 // classes for multiple trait, mixed models
17
18 using namespace std;
19
20 class TermData{
21 public:

```

```

22     double value;
23     unsigned level;
24 };
25
26 class DataNode{
27 public:
28     vector<TermData > trmVec;
29     vector<double>     depVec;
30 };
31
32
33 class MME;
34 class ModelTerm{
35 public:
36     unsigned start;
37     unsigned trait;
38
39     string name;
40     static MME *myMMEPtr;
41     Recoder<string>* myRecoderPtr;
42     vector<unsigned> factors;
43
44     ModelTerm(void){
45         myRecoderPtr = new Recoder<string>;
46     }
47     unsigned code(string str){return myRecoderPtr->code(str);}
48     unsigned nLevels(){return myRecoderPtr->size();}
49     string  getTermString();
50     unsigned getTermLevel (){
51         return code(getTermString());
52     }
53     double  getTermValue ();
54 };
55
56 class CovBlock {
57 public:
58     vector<ModelTerm*> modelTrmPtrVec;
59     matvec::doubleMatrix Var, Vari;

```

```
59     Pedigree* pedPtr;
60     CovBlock(void){pedPtr = 0;}
61
62     void addGinv(void);
63 };
64
65
66 class MME {
67 public:
68     string fileName;
69     Tokenizer colType;
70     Tokenizer colName;
71     Tokenizer colData;
72     unsigned numCols;
73     unsigned depCol;
74     vector <ModelTerm> modelTrmVec;
75     vector <CovBlock> covBlockVec;
76     vector <DataNode> dataVec;
77     unsigned numTerms, numTraits;
78     unsigned mmeSize;
79
80     matvec::doubleMatrix lhs, R, Ri;
81     matvec::Vector<double> rhs, sol;
82
83     void putColNames(string str);
84     void putColTypes(string str);
85     void inputData();
86     void displayData();
87     static double getDouble(string& Str);
88     void calcStarts();
89     void getSolution();
90     void calcWPW();
91     void addGinv();
92     void display();
93 };
94
95 string ModelTerm::getTermString(){
96     unsigned numFactors = factors.size();
```

```

96     string trmStr;
97     unsigned factorIndex = factors[0];
98     if(myMMEPtr->colType[factorIndex]=="COV"){
99         trmStr = myMMEPtr->colName[factorIndex];
100    }
101    else {
102        trmStr = myMMEPtr->colData[factorIndex];
103    }
104    for (unsigned i=1;i<numFactors;i++){
105        factorIndex = factors[i];
106        if(myMMEPtr->colType[factorIndex]=="COV"){
107            trmStr += "*" + myMMEPtr->colName[factorIndex];
108        }
109        else{
110            trmStr += "*" + myMMEPtr->colData[factorIndex];
111        }
112    }
113    return trmStr;
114 }
115
116 double ModelTerm::getTermValue(){
117     unsigned numFactors = factors.size();
118     double value = 1.0;
119     for (unsigned i=0;i<numFactors;i++){
120         unsigned factorIndex = factors[i];
121         if (myMMEPtr->colType[factorIndex]=="COV"){
122             string covStr = myMMEPtr->colData[factorIndex];
123             value *= MME::getDouble(covStr);
124         }
125     }
126     return value;
127 }
128
129 void MME::putColNames(string str){
130     string sep(" ");
131     str = "intercept " + str;
132     colName.getTokens(str,sep);
133     numCols = colName.size();

```

```
134 }
135
136 void MME::putColTypes(string str){
137     string sep(" ");
138     str = "CLASS " + str;
139     colType.getTokens(str,sep);
140     if (numCols!=colType.size()){
141         cerr <<"number of column names and column types do not match\n";
142         exit (-1);
143     }
144     unsigned n = 0;
145     for (unsigned i=0;i<numCols;i++){
146         if (colType[i] == "DEP") {
147             n++;
148         }
149     }
150     numTraits = n;
151 }
152
153 void MME::inputData(){
154     DataNode dataNode;
155     numTerms = modelTrmVec.size();
156     dataNode.trmVec.resize(numTerms);
157     dataNode.depVec.resize(numTraits);
158     ifstream datafile;
159     datafile.open(fileName.c_str());
160     if(!datafile) {
161         cerr << "Couldn't open data file: " << fileName << endl;
162         exit (-1);
163     }
164     unsigned linewidth = 1024;
165     char *line = new char [linewidth];
166     string sep(" ");
167     while (datafile.getline(line,linewidth)){
168         string inputStr(line);
169         inputStr = "--- " + inputStr;
170         colData.getTokens(inputStr,sep);
171         unsigned j=0;
```

```

172     for (unsigned i=0;i<numCols;i++){
173         if (colType[i]=="DEP") {
174             dataNode.depVec[j++] = getDouble(colData[i]);
175         }
176     }
177     for (unsigned i=0;i<numTerms;i++){
178         dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
179         dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
180     }
181     dataVec.push_back(dataNode);
182 }
183 }
184 }
185
186 double MME::getDouble(string& Str) {
187     istringstream inputStrStream(Str.c_str());
188     double val;
189     inputStrStream >> val;
190     return val;
191 }
192
193
194 void MME::calcStarts(){
195     modelTrmVec[0].start = 0;
196     for (unsigned i=1;i<numTerms;i++){
197         modelTrmVec[i].start = modelTrmVec[i-1].start
198                               + modelTrmVec[i-1].nLevels();
199     }
200     mmeSize = modelTrmVec[numTerms-1].start
201              + modelTrmVec[numTerms-1].nLevels();
202 }
203
204 void MME::calcWPW(){
205     unsigned ii,jj,ti,tj;
206     double vi,vj,tr_value;
207     rhs.resize(mmeSize,0.0);
208     lhs.resize(mmeSize,mmeSize,0.0);
209     Ri = R.inv();

```

```

210     for (unsigned i=0;i<dataVec.size();i++){
211         for (unsigned mi=0;mi<numTerms;mi++){
212             ii = modelTrmVec[mi].start
213                 + dataVec[i].trmVec[mi].level - 1;
214             ti = modelTrmVec[mi].trait;
215             vi = dataVec[i].trmVec[mi].value;
216             for (unsigned k=0;k<numTraits;k++){
217                 rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
218             }
219             for (unsigned mj=0;mj<numTerms;mj++){
220                 jj = modelTrmVec[mj].start
221                     + dataVec[i].trmVec[mj].level - 1;
222                 tj = modelTrmVec[mj].trait;
223                 vj = dataVec[i].trmVec[mj].value;
224                 lhs[ii][jj] += vi*Ri[ti][tj]*vj;
225             }
226         }
227     }
228 }
229
230 void MME::addGinv(){
231     for (unsigned i=0;i<covBlockVec.size();i++){
232         covBlockVec[i].addGinv();
233     }
234 }
235
236 void MME::getSolution(){
237     inputData();
238     calcStarts();
239     calcWPW();
240     addGinv();
241     sol = lhs.ginv0()*rhs;
242 }
243
244 void MME::display(){
245     cout << "LHS " << endl;
246     for (unsigned i = 0;i<mmeSize;i++){
247         for (unsigned j = 0;j<mmeSize;j++){

```



```

248     cout << setw(10)
249     << setprecision (4)
250     << setiosflags (ios::right | ios::fixed)
251     << lhs[i][j] <<" ";
252     }
253     cout << endl;
254 }
255 cout << "RHS " << endl;
256 cout << rhs << endl;
257 for (unsigned i=0;i<modelTrmVec.size();i++){
258     cout << "Solutions for " << modelTrmVec[i].name
259         << ", Trait " << modelTrmVec[i].trait+1 << endl;
260     Recoder<string>::iterator it;
261     for (it=modelTrmVec[i].myRecoderPtr->begin();
262         it!=modelTrmVec[i].myRecoderPtr->end();it++){
263         unsigned ii = modelTrmVec[i].start + it->second - 1;
264         cout << setw(10) << it->first << " " << sol[ii] << endl;
265     }
266 }
267 }
268
269 void CovBlock::addGinv(void){
270     Vari = Var.inv();
271     unsigned n = modelTrmPtrVec.size();
272     for (unsigned i=0;i<n;i++){
273         ModelTerm* mtermiPtr = modelTrmPtrVec[i];
274         unsigned starti = mtermiPtr->start;
275         for (unsigned j=0;j<n;j++){
276             ModelTerm* mtermjPtr = modelTrmPtrVec[j];
277             unsigned startj = mtermjPtr->start;
278             if (pedPtr){
279                 pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
280             }
281             else{
282                 unsigned numLevels = mtermiPtr->nLevels();
283                 for (unsigned k=0;k<numLevels;k++){
284                     ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
285                 }

```

```
286     }
287   }
288 }
289 }
290
291
292 MME* ModelTerm::myMMEPtr;
293
294 // model abstraction using MME class: two-trait, animal model
295
296
297 int main() {
298     try{
299         matvec::SESSION.initialize("matvec_trash");
300         Pedigree ped;
301         ped.inputPed("Data/additive.ped");
302         MME mme;
303         matvec::doubleMatrix R;
304         R.resize(2,2);
305         R(1,1) = 1.0;
306         R(1,2) = R(2,1) = 0.5;
307         R(2,2) = 2.0;
308         mme.R = R;
309         ModelTerm::myMMEPtr = &mme;
310         mme.fileName = "Data/additive2Tr.dat";
311         mme.putColNames("direct y1 y2");
312         mme.putColTypes("CLASS DEP DEP");
313
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);
345
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);
352         mme.getSolution();
353         mme.display();
354     }
355     catch (matvec::exception &ex) {
356         cerr << ex.what() << "\n";
357         exit(1);
358     }
359     catch (...) {
360         cerr << "other exceptions were caught\n";
361         exit(1);
```

362 }
363 }

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`

```
194 void MME::putModels(string str){
195     Tokenizer models;
196     string sep ="";
197     models.getTokens(str,sep);
198     for (unsigned i=0; i<models.size();i++){
199         putModel(models[i]);
200     }
```

201 }

The `getTokens` method on line 197 breaks up the “str” string at the semi-colons. 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.myRecorderPtr = new Recorder<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 `Recorder` is created and its address is stored in `modelTrm.myRecorderPtr`. 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 A matrix in (1.2) is the additive relationship matrix. The second version is to be used when 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].myRecorderPtr;
357                     ModelTerm::myMMEPtr->modelTrmVec[j].myRecorderPtr
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 `Recorder` 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

```
1  #include <fstream>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <sstream>
6  #include <stdarg.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <map>
10 #include <matvec/doublematrix.h>
11 #include <matvec/vector.h>
12 #include <matvec/session.h>
13 #include "util.h"
14 #include "ped.h"
15
16 // classes for multiple trait, mixed models
17
18 using namespace std;
19
20 class TermData{
21 public:
22     double value;
23     unsigned level;
24 };
25
26 class DataNode{
27 public:
28     vector<TermData > trmVec;
29     vector<double>     depVec;
30 };
31
32 class MME;
33
34 class ModelTerm{
35 public:
36     unsigned start;
```

```

37     unsigned trait;
38     string name;
39     string depVarName;
40     static MME *myMMEPtr;
41     Recoder<string>* myRecoderPtr;
42     vector<unsigned> factors;
43
44     unsigned code(string str){return myRecoderPtr->code(str);}
45     unsigned nLevels(){return myRecoderPtr->size();}
46     void putFactors(string str);
47     string  getTermString();
48     unsigned getTermLevel (){
49         return code(getTermString());
50     }
51     double  getTermValue ();
52 };
53
54 class CovBlock {
55 public:
56     vector<ModelTerm*> modelTrmPtrVec;
57     matvec::doubleMatrix Var, Vari;
58     Pedigree* pedPtr;
59     CovBlock(void){pedPtr = 0;}
60
61     CovBlock(string str, matvec::doubleMatrix V){
62         Var = V; pedPtr = 0;
63         buildModelTrmVec(str);
64     }
65
66     CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
67         Var = V;
68         pedPtr = &P;
69         buildModelTrmVec(str);
70     }
71     void buildModelTrmVec(string str);
72     void addGinv(void);
73 };

```

```
73
74 class MME {
75     private:
76         void putModel(string str);
77     public:
78         string fileName;
79         Tokenizer colType;
80         Tokenizer colName;
81         Tokenizer depVar;
82         Tokenizer colData;
83         unsigned numCols;
84         unsigned depCol;
85         vector <ModelTerm> modelTrmVec;
86         vector <CovBlock> covBlockVec;
87         vector <DataNode> dataVec;
88         unsigned numTerms, numTraits;
89         unsigned mmeSize;
90         matvec::doubleMatrix lhs, R, Ri;
91         matvec::Vector<double> rhs, sol;
92
93         void putColNames(string str);
94         void putColTypes(string str);
95         void putModels(string str);
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        }
104        void inputData();
105        void displayData();
106        static double getDouble(string& Str);
107        void calcStarts();
108        void getSolution();
109        void calcWPW();
110        void addGinv();
```

```
111     void display();
112 };
113
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 }
132
133
134 string ModelTerm::getTermString(){
135     unsigned numFactors = factors.size();
136     string trmStr;
137     unsigned factorIndex = factors[0];
138     if(myMMEPtr->colType[factorIndex]=="COV"){
139         trmStr = myMMEPtr->colName[factorIndex];
140     }
141     else {
142         trmStr = myMMEPtr->colData[factorIndex];
143     }
144     for (unsigned i=1;i<numFactors;i++){
145         factorIndex = factors[i];
146         if(myMMEPtr->colType[factorIndex]=="COV"){
147             trmStr += "*" + myMMEPtr->colName[factorIndex];
148         }
149     }
150 }
```



```

149         else{
150             trmStr += "*" + myMMEPtr->colData[factorIndex];
151         }
152     }
153     return trmStr;
154 }
155
156 double ModelTerm::getTermValue(){
157     unsigned numFactors = factors.size();
158     double value = 1.0;
159     for (unsigned i=0;i<numFactors;i++){
160         unsigned factorIndex = factors[i];
161         if (myMMEPtr->colType[factorIndex]=="COV"){
162             string covStr = myMMEPtr->colData[factorIndex];
163             value *= MME::getDouble(covStr);
164         }
165     }
166     return value;
167 }
168
169 void MME::putColNames(string str){
170     string sep(" ");
171     str = "intercept " + str;
172     colName.getTokens(str,sep);
173     numCols = colName.size();
174 }
175
176 void MME::putColTypes(string str){
177     string sep(" ");
178     str = "CLASS " + str;
179     colType.getTokens(str,sep);
180     if (numCols!=colType.size()){
181         cerr <<"number of column names and column types do not match\n";
182         exit (-1);
183     }
184     unsigned n = 0;
185     for (unsigned i=0;i<numCols;i++){
186         if (colType[i] == "DEP") {

```

```
187         depVar.push_back(colName[i]);
188         n++;
189     }
190 }
191 numTraits = n;
192 }
193
194 void MME::putModels(string str){
195     Tokenizer models;
196     string sep = ",";
197     models.getTokens(str,sep);
198     for (unsigned i=0; i<models.size();i++){
199         putModel(models[i]);
200     }
201 }
202
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 }
```

```
225
226 void MME::inputData(){
227     DataNode dataNode;
228     numTerms = modelTrmVec.size();
229     dataNode.trmVec.resize(numTerms);
230     dataNode.depVec.resize(numTraits);
231     ifstream datafile;
232     datafile.open(fileName.c_str());
233     if(!datafile) {
234         cerr << "Couldn't open data file: " << fileName << endl;
235         exit (-1);
236     }
237     unsigned linewidth = 1024;
238     char *line = new char [linewidth];
239     string sep(" ");
240     while (datafile.getline(line,linewidth)){
241         string inputStr(line);
242         inputStr = "--- " + inputStr;
243         colData.getTokens(inputStr,sep);
244         unsigned j=0;
245         for (unsigned i=0;i<numCols;i++){
246             if (colType[i]=="DEP") {
247                 dataNode.depVec[j++] = getDouble(colData[i]);
248             }
249         }
250     }
251     for (unsigned i=0;i<numTerms;i++){
252         dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
253         dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
254     }
255     dataVec.push_back(dataNode);
256 }
257 }
258
259 double MME::getDouble(string& Str) {
260     istringstream inputStrStream(Str.c_str());
261     double val;
262     inputStrStream >> val;
```

```

263     return val;
264 }
265
266 void MME::calcStarts(){
267     modelTrmVec[0].start = 0;
268     for (unsigned i=1;i<numTerms;i++){
269         modelTrmVec[i].start = modelTrmVec[i-1].start
270             + modelTrmVec[i-1].nLevels();
271     }
272     mmeSize = modelTrmVec[numTerms-1].start
273         + modelTrmVec[numTerms-1].nLevels();
274 }
275
276 void MME::calcWPW(){
277     unsigned ii,jj,ti,tj;
278     double vi,vj,tr_value;
279     rhs.resize(mmeSize,0.0);
280     lhs.resize(mmeSize,mmeSize,0.0);
281     Ri = R.inv();
282     for (unsigned i=0;i<dataVec.size();i++){
283         for (unsigned mi=0;mi<numTerms;mi++){
284             ii = modelTrmVec[mi].start + dataVec[i].trmVec[mi].level - 1;
285             ti = modelTrmVec[mi].trait;
286             vi = dataVec[i].trmVec[mi].value;
287             for (unsigned k=0;k<numTraits;k++){
288                 rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
289             }
290             for (unsigned mj=0;mj<numTerms;mj++){
291                 jj = modelTrmVec[mj].start + dataVec[i].trmVec[mj].level - 1;
292                 tj = modelTrmVec[mj].trait;
293                 vj = dataVec[i].trmVec[mj].value;
294                 lhs[ii][jj] += vi*Ri[ti][tj]*vj;
295             }
296         }
297     }
298 }
299
300 void MME::addGinv(){

```

```

301         for (unsigned i=0;i<covBlockVec.size();i++){
302             covBlockVec[i].addGinv();
303         }
304     }
305
306     void MME::getSolution(){
307         inputData();
308         calcStarts();
309         calcWPW();
310         addGinv();
311         sol = lhs.ginv0()*rhs;
312     }
313
314     void MME::display(){
315         cout << "LHS " << endl;
316         for (unsigned i = 0;i<mmeSize;i++){
317             for (unsigned j = 0;j<mmeSize;j++){
318                 cout << setw(10)
319                     << setprecision (4)
320                     << setiosflags (ios::right | ios::fixed)
321
322                 << lhs[i][j] <<" ";
323             }
324             cout << endl;
325         }
326         cout << "RHS " << endl;
327         cout << rhs << endl;
328         for (unsigned i=0;i<modelTrmVec.size();i++){
329             cout << "Solutions for " << modelTrmVec[i].name
330                 << ", Trait: " << modelTrmVec[i].depVarName << endl;
331             Recoder<string>::iterator it;
332             for (it=modelTrmVec[i].myRecoderPtr->begin();
333                 it!=modelTrmVec[i].myRecoderPtr->end();
334                 it++){
335                 unsigned ii = modelTrmVec[i].start + it->second - 1;
336                 cout << setw(10)
337                     << it->first
338                     << " "

```

```

339             << sol[ii]
340             << endl;
341         }
342     }
343 }
344
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 }
364
365 void CovBlock::addGinv(void){
366     Vari = Var.inv();
367     unsigned n = modelTrmPtrVec.size();
368     for (unsigned i=0;i<n;i++){
369         ModelTerm* mtermiPtr = modelTrmPtrVec[i];
370         unsigned starti = mtermiPtr->start;
371         for (unsigned j=0;j<n;j++){
372             ModelTerm* mtermjPtr = modelTrmPtrVec[j];
373             unsigned startj = mtermjPtr->start;
374             if (pedPtr){
375                 pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
376             }

```

```

377         else{
378             unsigned numLevels = mtermiPtr->nLevels();
379             for (unsigned k=0;k<numLevels;k++){
380                 ModelTerm::myMMEPtr->lhs[starti+k][startj+k] += Vari[i][j];
381             }
382         }
383     }
384 }
385 }
386
387
388 MME* ModelTerm::myMMEPtr;
389
390 // model abstraction using MME class: two-trait, animal model
391
392
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
410         mme.putModels("y1 = intercept directAdditive; \
411                       y2 = intercept directAdditive;");
412
413         mme.putVarCovMatrix("directAdditive",R,ped);
414         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 }
```


Chapter 3

Iterative Solvers for MME

Consider the system of consistent linear equations:

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

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:

$$\mathbf{x}_{n+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Ax}_n) + \mathbf{x}_n. \quad (3.1)$$

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

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

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 \mathbf{A} and the RHS by \mathbf{b} . The LHS of the MME is often too large to store in memory as a “fully-stored” matrix. However, \mathbf{A} is often very sparse. Thus, it is may be possible to store only the non-zero elements of \mathbf{A} and compute \mathbf{Ax}_n , using sparse matrix methods.

An alternative method for iteration using (3.2) avoids storing even the non-zero elements of \mathbf{A} . In this approach, called “iteration of data”, $\mathbf{A}\mathbf{x}_n$ is computed without first computing and storing \mathbf{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:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{d}_n, \quad (3.3)$$

where

$$\alpha_n = \frac{-\mathbf{r}'_n \mathbf{r}_n}{\mathbf{d}'_n \mathbf{A} \mathbf{d}_n}, \quad (3.4)$$

$$\mathbf{r}_n = \mathbf{A}\mathbf{x}_n - \mathbf{b}, \quad (3.5)$$

$$\mathbf{d}_n = \mathbf{r}_n - \beta_{n-1} \mathbf{d}_{n-1}, \quad (3.6)$$

and

$$\beta_{n-1} = \frac{-\mathbf{r}'_n \mathbf{r}_n}{\mathbf{r}'_{n-1} \mathbf{r}_{n-1}}. \quad (3.7)$$

It can be shown that the residual can be computed as

$$\mathbf{r}_n = \mathbf{r}_{n-1} + \alpha_{n-1} \mathbf{A} \mathbf{d}_{n-1}, \quad (3.8)$$

and thus avoiding computation of $\mathbf{A}\mathbf{x}_n$. 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 $\mathbf{A}\mathbf{x}$ and $\mathbf{A}\mathbf{d}$ can be computed by IOD without first computing \mathbf{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 \mathbf{x} that minimizes

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

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

In the conjugate gradient method, each direction \mathbf{d}_i is chosen such that the gradient \mathbf{r}_{i+1} will also be orthogonal to all the directions \mathbf{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

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

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 $\mathbf{x}_0 = \mathbf{0}$. At this point, the gradient of the $f(\mathbf{x})$ is

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

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

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

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

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

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

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

shows that

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

and in general,

$$\alpha_i = \frac{-\mathbf{d}'_i \mathbf{r}_i}{\mathbf{d}'_i \mathbf{A} \mathbf{d}_i}. \quad (3.15)$$

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

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{d}_1. \quad (3.16)$$

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

$$\begin{aligned} \mathbf{r}_2 &= \mathbf{A} \mathbf{x}_2 - \mathbf{b} \\ &= \mathbf{r}_1 + \alpha_1 \mathbf{A} \mathbf{d}_1, \end{aligned} \quad (3.17)$$

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

$$\begin{aligned} \mathbf{d}'_0 \mathbf{r}_2 &= \mathbf{d}'_0 \mathbf{r}_1 + \alpha_1 \mathbf{d}'_0 \mathbf{A} \mathbf{d}_1 \\ &= \alpha_1 \mathbf{d}'_0 \mathbf{A} \mathbf{d}_1 \\ &= 0. \end{aligned} \quad (3.18)$$

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

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

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

$$\mathbf{d}_1 = \mathbf{r}_1 - \beta_{10} \mathbf{d}_0. \quad (3.19)$$

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

$$\begin{aligned} \mathbf{d}'_0 \mathbf{A} \mathbf{d}_1 &= \mathbf{d}'_0 \mathbf{A} \mathbf{r}_1 - \beta_{10} \mathbf{d}'_0 \mathbf{A} \mathbf{d}_0 \\ &= 0 \end{aligned} \quad (3.20)$$

shows that

$$\beta_{10} = \frac{\mathbf{d}'_0 \mathbf{A} \mathbf{r}_1}{\mathbf{d}'_0 \mathbf{A} \mathbf{d}_0}. \quad (3.21)$$

In general,

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \alpha_i \mathbf{A} \mathbf{d}_i, \quad (3.22)$$

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

$$\mathbf{d}_i = \mathbf{r}_i - \sum_{j=0}^{i-1} \beta_{ij} \mathbf{d}_j, \quad (3.23)$$

where

$$\beta_{ij} = \frac{\mathbf{d}'_j \mathbf{A} \mathbf{r}_i}{\mathbf{d}'_j \mathbf{A} \mathbf{d}_j}. \quad (3.24)$$

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

$$\mathbf{r}'_i \mathbf{r}_j = 0 \text{ for } j < i. \quad (3.25)$$

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

$$\mathbf{d}_i = \mathbf{r}_i - \beta_{i-1} \mathbf{d}_{i-1}, \quad (3.26)$$

where

$$\beta_{i-1} = \frac{\mathbf{d}'_{i-1} \mathbf{A} \mathbf{r}_i}{\mathbf{d}'_{i-1} \mathbf{A} \mathbf{d}_{i-1}}. \quad (3.27)$$

Writing \mathbf{r}_i as

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \alpha_{i-1} \mathbf{A} \mathbf{d}_{i-1} \quad (3.28)$$

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

$$\mathbf{r}'_i \mathbf{r}_i = \alpha_{i-1} \mathbf{r}'_i \mathbf{A} \mathbf{d}_{i-1}. \quad (3.29)$$

Pre-multiplying (3.28) by \mathbf{d}'_{i-1} gives

$$\mathbf{d}'_{i-1} \mathbf{r}_{i-1} = -\alpha_{i-1} \mathbf{d}'_{i-1} \mathbf{A} \mathbf{d}_{i-1}. \quad (3.30)$$

Further, from (3.26), we can see that

$$\mathbf{d}'_i \mathbf{r}_i = \mathbf{r}'_i \mathbf{r}_i. \quad (3.31)$$

Using this in (3.30) gives

$$\mathbf{r}'_{i-1} \mathbf{r}_{i-1} = -\alpha_{i-1} \mathbf{d}'_{i-1} \mathbf{A} \mathbf{d}_{i-1}. \quad (3.32)$$

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

$$\beta_{i-1} = \frac{-\mathbf{r}'_i \mathbf{r}_i}{\mathbf{r}'_{i-1} \mathbf{r}_{i-1}}. \quad (3.33)$$

Finally, using (3.31) in (3.15) gives

$$\alpha_i = \frac{-\mathbf{r}'_i \mathbf{r}_i}{\mathbf{d}'_i \mathbf{A} \mathbf{d}_i}. \quad (3.34)$$

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 \mathbf{M} that is approximately equal to \mathbf{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:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{d}_n, \quad (3.35)$$

where

$$\alpha_n = \frac{-\mathbf{r}'_n \mathbf{M}^{-1} \mathbf{r}_n}{\mathbf{d}'_n \mathbf{A} \mathbf{d}_n}, \quad (3.36)$$

$$\mathbf{r}_n = \mathbf{A} \mathbf{x}_n - \mathbf{b}, \quad (3.37)$$

$$\mathbf{d}_n = \mathbf{M}^{-1} \mathbf{r}_n - \beta_{n-1} \mathbf{d}_{n-1}, \quad (3.38)$$

$$\beta_{n-1} = \frac{-\mathbf{r}'_n \mathbf{M}^{-1} \mathbf{r}_n}{\mathbf{r}'_{n-1} \mathbf{M}^{-1} \mathbf{r}_{n-1}}. \quad (3.39)$$

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

$$\mathbf{r}_n = \mathbf{r}_{n-1} + \alpha_{n-1} \mathbf{A} \mathbf{d}_{n-1}. \quad (3.40)$$

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

$$\begin{aligned} \mathbf{q} &= \mathbf{A}\mathbf{d} \\ &= \left\{ \sum_j a_{ij}d_j \right\}, \end{aligned} \quad (3.41)$$

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

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

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 $c_k(i, j)d_j$ is computed and the result is accumulated to q_i . More explicitly, $\mathbf{A}\mathbf{x}$ is written as

$$\begin{aligned} \mathbf{q} &= \left\{ \sum_j a_{ij}d_j \right\} \\ &= \left\{ \sum_j \sum_k c_k(i, j)d_j \right\} \\ &= \left\{ \sum_k \sum_j c_k(i, j)d_j \right\}, \end{aligned} \quad (3.43)$$

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

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

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 \mathbf{Ax} . A new method `MME::mmeTimes`, given below, is used to compute this product. In this method, \mathbf{A} is the LHS of the MME and \mathbf{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

```

201     if (solMethod=="direct"){
202         lhs[ii][jj] += vi*Ri[ti][tj]*vj;
203     }
204     else {
205         res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
206         if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
207     }
```


CovBlock::addGinv

```

396     if (MME::solMethod=="direct") {
397         ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
398     }
399     else {
400         MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
401         if (ii==jj) MME::diag[ii] += Vari[i][j];
402     }

```

Pedigree::addAinv

```

291     if (MME::solMethod=="direct"){
292         lhs[ii][jj] += ratio*q[i]*d*q[j];
293     }
294     else {
295         MME::res[ii] += ratio*q[i]*d*q[j] * (*MME::vec)[jj];
296         if (ii==jj) MME::diag[ii] += ratio*q[i]*d*q[j];
297     }

```

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 Ax , where 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){

```

```
264     sol = p*tempSol + (1-p)*sol;
265     cout <<"Iteration : "
266           << iter <<" diff = "
267           <<diff/mmeSize
268           << endl << endl;
269     mmeTimes(sol);
270     resid = (rhs - res);
271     tempSol = resid/diag + sol;
272     diff = resid.sumsq();
273 }
274 cout << resid << endl;
275 }
276
```

3.6 Listings of IOD programs

3.6.1 ped.h

```
1 #ifndef PMap_H
2 #define PMap_H
3 #include <fstream>
4 #include <iostream>
5 #include <iomanip>
6 #include <string>
7 #include <cctype>
8 #include <sstream>
9 #include <stdarg.h>
10 #include <stdlib.h>
11 #include <math.h>
12 #include <cmath>
13 #include <map>
14 #include <vector>
15 #include <algorithm>
16 #include <functional>
17 #include "util.h"
18 #include <matvec/doublematrix.h>
19
```

```
20 using namespace std;
21
22 class PNode {
23 public:
24
25     int    ind, sire, dam;
26     double f;
27     string ind_str, sire_str, dam_str;
28
29
30     PNode(string indstr, string sirestr, string damstr){
31
32         ind = -1;
33         sire = -1;
34         dam = -1;
35         f = -1.0;
36
37         ind_str = indstr;
38         sire_str = sirestr;
39         dam_str = damstr;
40     }
41
42 };
43
44
45 struct pcomp:public binary_function<double, double, bool> {
46     bool operator()(PNode *x, PNode *y)
47         { return x->ind < y->ind; }
48 };
49
50
51 class Pedigree : public map<string,PNode*> {
52
53
54 public:
55     unsigned COUNT;
56     SparseCij SpCij;
57     vector <PNode*> pedVector;
```

```
58   Recoder<string> coder;
59
60   void inputPed(char* fname);
61
62   void displayPed(void);
63   void generateEntriesforParents(void);
64   void codePed();
65   void code(PNode *ptr);
66   void calc_inbreeding(void);
67   void makePedVector(void);
68   void fillCoder(void);
69   double get_rij(int i, int j);
70   void output(char* ped);
71   void addAinv(matvec::doubleMatrix& lhs,
72               unsigned startRow,
73               unsigned statCol,
74               double ratio);
75 };
76
77 #endif
78
```

3.6.2 ped.cpp

```
1  #include <fstream>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <cctype>
6  #include <sstream>
7  #include <sstream>
8  #include <stdarg.h>
9  #include <stdlib.h>
10 #include <math.h>
11 #include <cmath>
12 #include <vector>
13 #include <algorithm>
14 #include <functional>
```

```
15 #include <map>
16 #include "ped.h"
17 #include "mme1.h"
18
19 using namespace std;
20
21 void Pedigree::inputPed(char* fname){
22     cout << "reading pedigree file \n";
23     double rec = 0, rec1 = 0;
24     string indstr, sirestr, damstr;
25     ifstream datafile(fname);
26     if(!datafile){
27         cout<< "Cannot open data file! \n";
28         exit(1);
29     }
30     datafile.setf(ios::skipws);
31     PNode *ptr;
32     while (datafile>>indstr>>sirestr>>damstr){
33         rec++;
34         if(rec==1000){
35             cout<<". ";
36             cout.flush();
37             rec1 += rec;
38             rec = 0;
39         }
40         ptr = new PNode(indstr, sirestr, damstr);
41         (*this)[indstr] = ptr;
42     }
43     datafile.close();
44     generateEntriesforParents();
45     codePed();
46     makePedVector();
47     calc_inbreeding();
48     fillCoder();
49 }
50
51 void Pedigree::displayPed(void){
52     Pedigree::iterator it;
```

```

53     vector<PNode*>::iterator vecit;
54     for (vecit=pedVector.begin();vecit!=pedVector.end();vecit++){
55         cout << setw(10) << (*vecit)->ind
56         << setw(10) << (*vecit)->sire
57         << setw(10) << (*vecit)->dam
58         << setw(20) << ((*vecit)->ind_str).c_str()
59         << setw(20) << (*vecit)->f << endl;
60     }
61     cout << endl;
62
63 }
64 void Pedigree::generateEntriesforParents(void) {
65     // if a parent does not have an entry, we make it a founder
66     cout << "generating missing entries for parents \n";
67     unsigned sire_count = 0;
68     unsigned dam_count = 0;
69     double rec = 0, rec1 = 0;
70     Pedigree::iterator it, parent_it;
71     for(it=begin();it!=end();it++){
72         rec++;
73         if(rec==1000){
74             cout<<". ";
75             cout.flush();
76             rec1 += rec;
77             rec = 0;
78         }
79         PNode *ptr = (*it).second;
80         if(ptr->sire_str!="0"){
81             parent_it = (*this).find(ptr->sire_str);
82             if(parent_it == end()){ // sire has no entry
83                 PNode *ptrs = new PNode(ptr->sire_str, "0", "0");
84                 (*this)[ptr->sire_str] = ptrs;
85             }
86         }
87         if(ptr->dam_str!="0"){
88             parent_it = (*this).find(ptr->dam_str);
89             if(parent_it == end()){ // dam has no entry
90                 PNode *ptrd = new PNode(ptr->dam_str, "0", "0");

```

```

91             (*this)[ptr->dam_str] = ptrd;
92         }
93     }
94 }
95 }
96
97
98 void Pedigree::codePed(){
99     cout << "coding pedigree \n";
100     Pedigree::iterator it;
101     COUNT = 0;
102     unsigned rec = 0, rec1 = 0;
103
104     for(it=begin();it!=end();it++){
105         rec++;
106         if(rec==1000){
107             cout<<". ";
108             cout.flush();
109             rec1 += rec;
110             rec = 0;
111         };
112         cout.flush();
113         PNode *ptr =(*it).second;
114         code(ptr);
115     }
116 }
117
118 void Pedigree::code(PNode *ptr){
119
120     if(ptr->ind !=-1) { // already coded
121         return;
122     }
123     if(ptr->sire_str == "0" && ptr->dam_str == "0"){ // founder
124         ptr->ind = ++COUNT;
125         ptr->sire = 0;
126         ptr->dam = 0;
127     }
128     else if(ptr->sire_str != "0" && ptr->dam_str == "0"){

```

```

129         // dam missing, sire is not missing
130         PNode* sire_ptr = (*this)[ptr->sire_str];
131         if (sire_ptr->ind == -1) {
132             code(sire_ptr);
133         }
134         ptr->ind = ++COUNT;
135         ptr->sire = sire_ptr->ind;
136         ptr->dam = 0;
137     }
138     else if(ptr->dam_str != "0" && ptr->sire_str == "0"){
139         // sire missing, dam is not missing
140         PNode* dam_ptr = (*this)[ptr->dam_str];
141         if (dam_ptr->ind == -1) {
142             code(dam_ptr);
143         }
144         ptr->ind = ++COUNT;
145         ptr->sire = 0;
146         ptr->dam = dam_ptr->ind;
147     }
148     else{
149         PNode* sire_ptr = (*this)[ptr->sire_str];
150         if (sire_ptr->ind == -1) {
151             code(sire_ptr);
152         }
153         PNode* dam_ptr = (*this)[ptr->dam_str];
154         if (dam_ptr->ind == -1) {
155             code(dam_ptr);
156         }
157         ptr->ind = ++COUNT;
158         ptr->sire = sire_ptr->ind;
159         ptr->dam = dam_ptr->ind;
160     }
161 }
162
163 void Pedigree::calc_inbreeding(void){
164     vector <PNode*>::iterator it;
165     unsigned rec = 0, rec1 = 0, non_rec = 0;
166     cout << "calculating inbreeding \n";

```



```

167         for (it=pedVector.begin();it!=pedVector.end();it++){
168             rec++;
169             if(rec==1000){
170                 cout<<".";
171                 cout.flush();
172                 rec1 += rec;
173                 rec = 0;
174             };
175             (*it)->f = get_rij((*it)->sire,(*it)->dam);
176         }
177     }
178
179
180 double Pedigree::get_rij(int i, int j){
181
182     if (i==0||j==0){
183         return 0.0;
184     }
185     double x = SpCij.retrieve_cij(i,j);
186     if(x != -1.0) {
187         return x;
188     }
189     int old, young;
190     if(i < j){
191         old = i;
192         young = j;
193     }
194     else if(j < i){
195         old = j;
196         young = i;
197     }
198     else{
199         double f = pedVector[i-1]->f;
200         x = 0.5*(1 + f);
201         SpCij.put_cij(i,j,x);
202         return x;
203     }
204     int y_sire = pedVector[young-1]->sire;

```

```

205         int y_dam = pedVector[young-1]->dam;
206         x = (get_rij(old,y_sire)+get_rij(old,y_dam))/2.0;
207         SpCij.put_cij(i,j,x);
208         return x;
209     }
210
211 void Pedigree:: output(char* ped){
212
213     ofstream pedfile(ped);
214     vector<PNode*>::iterator vecit;
215     pedfile.setf(ios::fixed | ios::right);
216     for (vecit=pedVector.begin();vecit!=pedVector.end();vecit++){
217         pedfile << setw(10) << (*vecit)->ind
218         << setw(10) << (*vecit)->sire
219         << setw(10) << (*vecit)->dam
220         << setw(20) << ((*vecit)->ind_str).c_str()
221         << setw(20) << (*vecit)->f <<      endl;
222     }
223 }
224
225 void Pedigree::makePedVector(void){
226     Pedigree::iterator it;
227     pedVector.resize(size());
228     for(it=begin();it!=end();it++){
229         PNode *ptr = (*it).second;
230         unsigned i = ptr->ind - 1;
231         pedVector[i] = ptr;
232     }
233 }
234
235 void Pedigree::fillCoder(void){
236     vector<PNode *>::iterator it;
237     for(it=pedVector.begin();it!=pedVector.end();it++){
238         coder.code((*it)->ind_str);
239     }
240 }
241

```

```
241 void Pedigree::addAinv(matvec::doubleMatrix& lhs,
242                       unsigned startRow,
243                       unsigned startCol,
244                       double ratio){
245     double q[3];
246     double d,fs,fd;
247     unsigned pos[3];
248     vector<PNode*>::iterator it;
249     if (coder.size()>pedVector.size()){
250         cout << "Pedigree is not complete \n";
251         exit(-1);
252     }
253     for (it=pedVector.begin();it!=pedVector.end();it++){
254         pos[0] = (*it)->sire;
255         pos[1] = (*it)->dam;
256         pos[2] = (*it)->ind;
257         if((*it)->sire && (*it)->dam){
258             q[0] = -0.5;
259             q[1] = -0.5;
260             q[2] = 1.0;
261             fs = pedVector[pos[0]-1]->f;
262             fd = pedVector[pos[1]-1]->f;
263             d = 4.0/(2 - fs - fd);
264         }
265         else if((*it)->sire){
266             q[0] = -0.5;
267             q[1] = 0.0;
268             q[2] = 1.0;
269             fs = pedVector[pos[0]-1]->f;
270             d = 4.0/(3-fs);
271         }
272         else if((*it)->dam){
273             q[0] = 0.0;
274             q[1] = -0.5;
275             q[2] = 1.0;
276             fd = pedVector[pos[1]-1]->f;
277             d = 4.0/(3-fd);
278         }
279     }
```

```

279         else{
280             q[0] = 0.0;
281             q[1] = 0.0;
282             q[2] = 1.0;
283             d = 1.0;
284         }
285         for (unsigned i=0;i<3;i++){
286             if(pos[i]){
287                 unsigned ii = startRow + pos[i] - 1;
288                 for (unsigned j=0;j<3;j++){
289                     if(pos[j]) {
290                         unsigned jj = startCol + pos[j] - 1;
291                         if (MME::solMethod=="direct"){
292                             lhs[ii][jj] += ratio*q[i]*d*q[j];
293                         }
294                         else {
295                             MME::res[ii] += ratio*q[i]*d*q[j] * (*MME::vec)[jj];
296                             if (ii==jj) MME::diag[ii] += ratio*q[i]*d*q[j];
297                         }
298                     }
299                 }
300             }
301         }
302     }
303 }
304
305
306

```

3.6.3 mme1.h

```

1  ifndef MME_H
2  #define MME_H
3  #include <fstream>
4  #include <iostream>
5  #include <iomanip>
6  #include <string>
7  #include <sstream>

```

```
8 #include <stdarg.h>
9 #include <stdlib.h>
10 #include <math.h>
11 #include <map>
12 #include <matvec/doublematrix.h>
13 #include <matvec/vector.h>
14 #include <matvec/session.h>
15 #include "util.h"
16 #include "ped.h"
17
18
19 // classes for multiple trait, mixed models with iteration on data
20
21 using namespace std;
22
23 class TermData{
24 public:
25     double value;
26     unsigned level;
27 };
28
29 class DataNode{
30 public:
31     vector<TermData > trmVec;
32     vector<double>     depVec;
33 };
34
35 class MME;
36
37 class ModelTerm{
38 public:
39     unsigned start;
40     unsigned trait;
41     string name;
42     string depVarName;
43     static MME *myMMEPtr;
44     Recoder<string>* myRecoderPtr;
45     vector<unsigned> factors;
```

```

46
47     unsigned code(string str){return myRecoderPtr->code(str);}
48     unsigned nLevels(){return myRecoderPtr->size();}
49     void putFactors(string str);
50     string  getTermString();
51     unsigned getTermLevel (){
52         return code(getTermString());
53     }
54     double  getTermValue ();
55 };
56
57 class CovBlock {
58 public:
59     vector<ModelTerm*> modelTrmPtrVec;
60     matvec::doubleMatrix Var, Vari;
61     Pedigree* pedPtr;
62     CovBlock(void){pedPtr = 0;}
63     CovBlock(string str, matvec::doubleMatrix V){
64         Var = V;
65         pedPtr = 0;
66         buildModelTrmVec(str);
67     }
68     CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
69         Var = V;
70         pedPtr = &P;
71         buildModelTrmVec(str);
72     }
73     void buildModelTrmVec(string str);
74     void addGinv(void);
75 };
76
77
78 class MME {
79 private:
80     void putModel(string str);
81 public:
82     static string solMethod;
83     static matvec::Vector<double> *vec, diag, res;

```

```
84     string fileName;
85     Tokenizer colType;
86     Tokenizer colName;
87     Tokenizer depVar;
88     Tokenizer colData;
89     unsigned numCols;
90     unsigned depCol;
91     vector <ModelTerm> modelTrmVec;
92     vector <CovBlock> covBlockVec;
93     vector <DataNode> dataVec;
94     unsigned numTerms, numTraits;
95     unsigned mmeSize;
96     matvec::doubleMatrix lhs, R, Ri;
97     matvec::Vector<double> rhs, sol, tempSol;
98
99     void putColNames(string str);
100    void putColTypes(string str);
101    void putModels(string str);
102    void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
103        CovBlock covBlock(str,V,P);
104        covBlockVec.push_back(covBlock);
105    }
106    void putVarCovMatrix(string str, matvec::doubleMatrix V){
107        CovBlock covBlock(str,V);
108        covBlockVec.push_back(covBlock);
109    }
110    void initSetup();
111    void inputData();
112    void displayData();
113    static double getDouble(string& Str);
114    void calcStarts();
115    void getDirectSolution();
116    void getJacobiSolution(double p);
117    void getCGSolution();
118    void getPCCGSolution();
119
120    void mmeTimes(matvec::Vector<double>& x);
121    void calcWPW();
```

```
122     void addGinv();
123     void display();
124 };
125 #endif
```

3.6.4 mme1.cpp

```
1  /*
2  *  mme1.cpp
3  *  C++WkShp
4  *
5  *  Created by Rohan Fernando on 5/6/05.
6  *  Copyright 2005, All rights reserved.
7  *
8  */
9
10 #include "mme1.h"
11
12 MME* ModelTerm::myMMEPtr;
13 string MME::solMethod;
14 matvec::Vector<double> *MME::vec, MME::diag, MME::res;
15
16 void ModelTerm::putFactors(string str){
17     Tokenizer tokens;
18     string sep("*");
19     tokens.getTokens(str,sep);
20     factors.clear();
21     for (unsigned i=0;i<tokens.size();i++){
22         unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
23         if (factorIndex == -1){
24             cerr <<"Independent Variable "
25                 << tokens[i]
26                 << " not in list of column names \n";
27             exit(-1);
28         }
29         else {
30             factors.push_back(factorIndex);
31         }
32     }
```



```
32     }
33 }
34
35
36 string ModelTerm::getTermString(){
37     unsigned numFactors = factors.size();
38     string trmStr;
39     unsigned factorIndex = factors[0];
40     if(myMMEPtr->colType[factorIndex]=="COV"){
41         trmStr = myMMEPtr->colName[factorIndex];
42     }
43     else {
44         trmStr = myMMEPtr->colData[factorIndex];
45     }
46     for (unsigned i=1;i<numFactors;i++){
47         factorIndex = factors[i];
48         if(myMMEPtr->colType[factorIndex]=="COV"){
49             trmStr += "*" + myMMEPtr->colName[factorIndex];
50         }
51         else{
52             trmStr += "*" + myMMEPtr->colData[factorIndex];
53         }
54     }
55     return trmStr;
56 }
57
58 double ModelTerm::getTermValue(){
59     unsigned numFactors = factors.size();
60     double value = 1.0;
61     for (unsigned i=0;i<numFactors;i++){
62         unsigned factorIndex = factors[i];
63         if (myMMEPtr->colType[factorIndex]=="COV"){
64             string covStr = myMMEPtr->colData[factorIndex];
65             value *= MME::getDouble(covStr);
66         }
67     }
68     return value;
69 }
```

```
70
71 void MME::putColNames(string str){
72     string sep(" ");
73     str = "intercept " + str;
74     colName.getTokens(str,sep);
75     numCols = colName.size();
76 }
77
78 void MME::putColTypes(string str){
79     string sep(" ");
80     str = "CLASS " + str;
81     colType.getTokens(str,sep);
82     if (numCols!=colType.size()){
83         cerr <<"number of column names and column types do not match\n";
84         exit (-1);
85     }
86     unsigned n = 0;
87     for (unsigned i=0;i<numCols;i++){
88         if (colType[i] == "DEP") {
89             depVar.push_back(colName[i]);
90             n++;
91         }
92     }
93     numTraits = n;
94 }
95
96 void MME::putModels(string str){
97     Tokenizer models;
98     string sep =";";
99     models.getTokens(str,sep);
100    for (unsigned i=0; i<models.size();i++){
101        putModel(models[i]);
102    }
103 }
104
105 void MME::putModel(string str){
106     string sep(" =+");
107     Tokenizer modelTokens;
```

```

108     modelTokens.getTokens(str,sep);
109     unsigned nTokens = modelTokens.size();
110     int depVarIndex = colName.getIndex(modelTokens[0]);
111     if (depVarIndex == -1){
112         cerr << "Dependent Variable "
113             << modelTokens[0]
114             << " not in list of column names \n";
115         exit (-1);
116     }
117     ModelTerm modelTrm;
118     modelTrm.depVarName = modelTokens[0];
119     modelTrm.trait = depVar.getIndex(modelTokens[0]);
120     for (unsigned i=1;i<nTokens;i++){
121         modelTrm.myRecorderPtr = new Recorder<string>;
122         modelTrm.name = modelTokens[i];
123         modelTrm.putFactors(modelTrm.name);
124         modelTrmVec.push_back(modelTrm);
125     }
126 }
127
128 void MME::inputData(){
129     DataNode dataNode;
130     numTerms = modelTrmVec.size();
131     dataNode.trmVec.resize(numTerms);
132     dataNode.depVec.resize(numTraits);
133     ifstream datafile;
134     datafile.open(fileName.c_str());
135     if(!datafile) {
136         cerr << "Couldn't open data file: " << fileName << endl;
137         exit (-1);
138     }
139     unsigned linewidth = 1024;
140     char *line = new char [linewidth];
141     string sep(" ");
142     while (datafile.getline(line,linewidth)){
143         string inputStr(line);
144         inputStr = "--- " + inputStr;
145         colData.getTokens(inputStr,sep);

```

```

146     unsigned j=0;
147     for (unsigned i=0;i<numCols;i++){
148         if (colType[i]=="DEP") {
149             dataNode.depVec[j++] = getDouble(colData[i]);
150         }
151     }
152 }
153 for (unsigned i=0;i<numTerms;i++){
154     dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
155     dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
156 }
157 dataVec.push_back(dataNode);
158 }
159 }
160
161 double MME::getDouble(string& Str) {
162     ifstream inputStrStream(Str.c_str());
163     double val;
164     inputStrStream >> val;
165     return val;
166 }
167
168 void MME::calcStarts(){
169     modelTrmVec[0].start = 0;
170     for (unsigned i=1;i<numTerms;i++){
171         modelTrmVec[i].start = modelTrmVec[i-1].start
172             + modelTrmVec[i-1].nLevels();
173     }
174     mmeSize = modelTrmVec[numTerms-1].start
175         + modelTrmVec[numTerms-1].nLevels();
176 }
177
178 void MME::calcWPW(){
179     unsigned ii,jj,ti,tj;
180     double vi,vj,tr_value;
181     if(solMethod!="direct"){
182         diag.resize(mmeSize,0.0);
183         res.resize(mmeSize,0.0);

```

```

184     }
185     rhs.resize(mmeSize,0.0);
186     Ri = R.inv();
187     for (unsigned i=0;i<dataVec.size();i++){
188         for (unsigned mi=0;mi<numTerms;mi++){
189             ii = modelTrmVec[mi].start
190                 + dataVec[i].trmVec[mi].level - 1;
191             ti = modelTrmVec[mi].trait;
192             vi = dataVec[i].trmVec[mi].value;
193             for (unsigned k=0;k<numTraits;k++){
194                 rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
195             }
196             for (unsigned mj=0;mj<numTerms;mj++){
197                 jj = modelTrmVec[mj].start
198                     + dataVec[i].trmVec[mj].level - 1;
199                 tj = modelTrmVec[mj].trait;
200                 vj = dataVec[i].trmVec[mj].value;
201                 if (solMethod=="direct"){
202                     lhs[ii][jj] += vi*Ri[ti][tj]*vj;
203                 }
204                 else {
205                     res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
206                     if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
207                 }
208             }
209         }
210     }
211 }
212
213 void MME::addGinv(){
214     for (unsigned i=0;i<covBlockVec.size();i++){
215         covBlockVec[i].addGinv();
216     }
217 }
218
219 void MME::initSetup(){
220     inputData();
221     calcStarts();

```

```

222     if(solMethod=="direct"){
223         lhs.resize(mmeSize,mmeSize,0.0);
224     }
225     else {
226         sol.resize(mmeSize,0.0);
227     }
228 }
229
230
231 void MME::getDirectSolution(){
232     solMethod = "direct";
233     initSetup();
234     calcWPW();
235     addGinv();
236     sol = lhs.ginv0()*rhs;
237 }
238
239 void MME::display(){
240
241     if (solMethod=="direct"){
242         cout << "LHS " << endl;
243         for (unsigned i = 0;i<mmeSize;i++){
244             for (unsigned j = 0;j<mmeSize;j++){
245                 cout << setw(5) << lhs[i][j] <<" ";
246             }
247             cout << endl;
248         }
249     }
250     cout << "RHS " << endl;
251     cout << rhs << endl;
252     for (unsigned i=0;i<modelTrmVec.size();i++){
253         cout << "Solutions for " << modelTrmVec[i].name
254             << ", Trait: " << modelTrmVec[i].depVarName << endl;
255         Recoder<string>::iterator it;
256         for (it=modelTrmVec[i].myRecoderPtr->begin();
257             it!=modelTrmVec[i].myRecoderPtr->end();
258             it++){
259             unsigned ii = modelTrmVec[i].start + it->second - 1;

```

```
260         cout << setw(10) << it->first << " " << sol[ii] << endl;
261     }
262 }
263 }
264
265 void MME::getJacobiSolution(double p){
266     solMethod = "jacobi";
267     initSetup();
268     mmeTimes(sol); // result goes into res, also creates rhs and diag
269     matvec::Vector<double> resid = (rhs - res);
270     tempSol = resid/diag + sol;
271     double diff = resid.sumsq();
272     unsigned iter = 0;
273     while(diff/mmeSize > .0000001 && ++iter<200){
274         sol = p*tempSol + (1-p)*sol;
275         cout <<"Iteration : "
276             << iter <<" diff = "
277             <<diff/mmeSize
278             << endl << endl;
279         mmeTimes(sol);
280         resid = (rhs - res);
281         tempSol = resid/diag + sol;
282         diff = resid.sumsq();
283     }
284     cout << resid << endl;
285 }
286
287 void MME::getCGSolution(){
288     solMethod = "cg";
289     initSetup();
290     mmeTimes(sol); // result goes into res, also creates rhs and diag
291     matvec::Vector<double> resid = (res-rhs);
292     matvec::Vector<double> d = resid;
293     double oldDiffSq;
294     double newDiffSq = resid.sumsq();
295     unsigned iter = 0;
296     while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
297         cout <<"Iteration : "
```

```

298         << iter <<" diff = "
299         <<newDiffSq/mmeSize
300         << endl << endl;
301     mmeTimes(d);
302     double alpha = -newDiffSq/(res*d).sum();
303     sol += alpha*d;
304     if (iter%10){
305         resid += alpha*res;
306     }
307     else {
308         mmeTimes(sol);
309         resid = (res-rhs);
310     }
311     oldDiffSq = newDiffSq;
312     newDiffSq = resid.sumSq();
313     double beta = -newDiffSq/oldDiffSq;
314     d = resid - beta*d;
315 }
316 cout << resid << endl;
317 }
318
319 void MME::getPCCGSolution(){
320     solMethod = "pccg";
321     initSetup();
322     mmeTimes(sol); // result goes into res, also creates rhs and diag
323     matvec::Vector<double> resid = (res - rhs);
324     matvec::Vector<double> d = resid/diag;
325     double oldDiffSq;
326     double newDiffSq = (resid*d).sum();
327     unsigned iter = 0;
328     while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
329         cout <<"Iteration : "
330             << iter <<" diff = "
331             <<newDiffSq/mmeSize
332             << endl << endl;
333         mmeTimes(d);
334         double alpha = -newDiffSq/(res*d).sum();
335         sol += alpha*d;

```



```

336     if (iter % 10){
337         resid += alpha*res;
338     }
339     else {
340         mmeTimes(sol);
341         resid = (res - rhs);
342     }
343     matvec::Vector<double> s = resid/diag;
344     oldDiffSq = newDiffSq;
345     newDiffSq = (resid*s).sum();
346     double beta = -newDiffSq/oldDiffSq;
347     d = s - beta*d;
348 }
349 cout << resid << endl;
350 }
351
352 void MME::mmeTimes(matvec::Vector<double>& x){
353     vec = &x;
354     calcWPW();
355     addGinv();
356 }
357
358 void CovBlock::buildModelTrmVec(string str){
359     string sep(" ");
360     Tokenizer modelTokens;
361     modelTokens.getTokens(str,sep);
362     unsigned nTokens = modelTokens.size();
363     unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
364     for (unsigned i=0;i<nTokens;i++){
365         for (unsigned j=0;j<numModelTrms;j++){
366             if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
367                 modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
368                 if (pedPtr){
369                     delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
370                     ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr
371                     = &pedPtr->coder;
372                 }
373             }

```

```

374     }
375   }
376 }

378 void CovBlock::addGinv(void){
379   Vari = Var.inv();
380   unsigned n = modelTrmPtrVec.size();
381   for (unsigned i=0;i<n;i++){
382     ModelTerm* mtermiPtr = modelTrmPtrVec[i];
383     unsigned starti = mtermiPtr->start;
384     for (unsigned j=0;j<n;j++){
385       ModelTerm* mtermjPtr = modelTrmPtrVec[j];
386       unsigned startj = mtermjPtr->start;
387       if (pedPtr){
388         pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,
389                       starti,startj,Vari[i][j]);
390       }
391       else{
392         unsigned numLevels = mtermiPtr->nLevels();
393         for (unsigned k=0;k<numLevels;k++){
394           unsigned ii = starti + k;
395           unsigned jj = startj + k;
396           if (MME::solMethod=="direct") {
397             ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
398           }
399           else {
400             MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
401             if (ii==jj) MME::diag[ii] += Vari[i][j];
402           }
403         }
404       }
405     }
406   }
407 }

```

3.6.5 iterDataMSMME.cpp

```
1  #include <fstream>
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  #include <sstream>
6  #include <stdarg.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <map>
10 #include <matvec/doublematrix.h>
11 #include <matvec/vector.h>
12 #include <matvec/session.h>
13 #include "util.h"
14 #include "ped.h"
15 #include "mme1.h"
16
17
18 // model abstraction using MME class: animal model, iteration on data, PCCG
19
20
21 int main() {
22     try{
23         matvec::SESSION.initialize("matvec_trash");
24         Pedigree ped;
25         ped.inputPed("Data/additive.ped");
26         ped.displayPed();
27         MME mme;
28         matvec::doubleMatrix R;
29         R.resize(1,1);
30         R(1,1) = 1.0;
31         mme.R = R;
32         ModelTerm::myMMEPtr = &mme;
33         mme.fileName = "Data/additive2Tr.dat";
34         mme.putColNames("directAdditive y1 y2");
35         mme.putColTypes("CLASS DEP DEP");
36         mme.putModels("y1 = intercept directAdditive;");
```

```
37         mme.putVarCovMatrix("directAdditive",R,ped);
38         mme.getPCCGSolution();
39         mme.display();
40     }
41     catch (matvec::exception &ex) {
42         cerr << ex.what() << "\n";
43         exit(1);
44     }
45     catch (...) {
46         cerr << "other exceptions were caught\n";
47         exit(1);
48     }
49 }
50
```

Chapter 4

Marker Assisted BLUP

4.1 Single trait models

Consider the following univariate, mixed linear model:

$$y_i = \mathbf{x}_i\boldsymbol{\beta} + v_i^p + v_i^m + u_i + e_i, \quad (4.1)$$

where y_i is the phenotypic value for animal i , $\mathbf{x}_i\boldsymbol{\beta}$ 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 \mathbf{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 \mathbf{v} . Also, to setup the MME for this model, we need the inverse of $\text{Var}(\mathbf{v}) = \boldsymbol{\Sigma}_v$. This covariance matrix can be approximated by the recursive formula:

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

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 Σ_v of gametic covariances can be expressed in matrix notation. To do so, the rows and columns of Σ_v are ordered such that those for ancestors precede those for descendants. Suppose Σ_s is the gametic covariance matrix for individuals $1, 2, \dots, i-1$. This matrix can be expanded to include the covariances with v_i^m , for example, as

$$\Sigma_{s+1} = \begin{bmatrix} \Sigma_s & \Sigma_s \mathbf{q} \\ \mathbf{q}' \Sigma_s & \text{Var}(v_i^m) \end{bmatrix}, \quad (4.3)$$

where \mathbf{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, $\text{Var}(v_i^m) = \sigma_Q^2$ 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 Σ_v singular. For example, suppose individual 1 is the dam of 3 and $\Pr(O_Q(m, 3) = m | \mathbf{M}) = 0.0$ and $\Pr(O_Q(p, 3) = m | \mathbf{M}) = 0.9$. Then, the model for the phenotypic value of individual 3 should be written as

$$y_3 = \mathbf{x}_3 \boldsymbol{\beta} + v_3^p + v_1^p + u_3 + e_i, \quad (4.4)$$

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 Σ_v efficiently. Suppose that Σ_s^{-1} is the inverse of the sub matrix Σ_s defined previously, then the inverse of Σ_{s+1} is

$$\Sigma_{s+1}^{-1} = \begin{bmatrix} \Sigma_s^{-1} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} -\mathbf{q} \\ 1 \end{bmatrix} \left(\frac{1}{v_{ii}} \right) \begin{bmatrix} -\mathbf{q}' & 1 \end{bmatrix}, \quad (4.5)$$

where

$$v_{ii} = [\text{Var}(v_i^m) - \mathbf{q}' \Sigma_s \mathbf{q}]. \quad (4.6)$$

Note that \mathbf{q} has only two non-zero elements. Thus, (4.5) leads to an efficient algorithm, and the resulting inverse is very sparse. Also, because \mathbf{q} has only two non-zero elements, only four elements from Σ_v are needed to compute v_{ii} , and these elements can be obtained efficiently without constructing the entire Σ_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 \mathbf{v}_t denote the vector of random MQTL allele values for trait t . Then, the vector for trait t' is:

$$\mathbf{v}_{t'} = \mathbf{v}_t \frac{a_{t'}}{a_t}. \quad (4.7)$$

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 Σ_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 Σ_v has to be added to the position of the LHS corresponding to 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);

```



```
32     regCoeff(1) = 1.0;
33     regCoeff(2) = 0.9;
34     Q.regCoeff = regCoeff;
35     Q.variance = 0.25;
36     Q.MQTLNames = "pQ1 mQ1";
37     Q.display();
38
39     MME mme;
40
41     mme.fileName = "Data/additive2Tr.dat";
42     mme.putColNames("directAdditive y1 y2");
43     mme.putColTypes("CLASS          DEP DEP");
44     mme.putMQTL(Q);
45     mme.mergeMQTLLevelsBy("directAdditive");
46     mme.addColNames("pQ1 mQ1");
47     mme.addColTypes("MQTL MQTL");
48     string modelString = "y1 = intercept + directAdditive + pQ1 + mQ1;";
49     modelString        += "y2 = intercept + directAdditive + pQ1 + mQ1 ";
50     mme.putModels(modelString);
51
52     matvec::doubleMatrix Va;
53     Va.resize(2,2,0.0);
54     Va(1,1) = 1.0;
55     Va(1,2) = 0.0;
56     Va(2,1) = 0.0;
57     Va(2,2) = 1.0;
58     mme.putVarCovMatrix("directAdditive",Va,ped);
59
60     mme.putVarCovMatrix(Q);
61
62     matvec::doubleMatrix resVar;
63     resVar.resize(2,2,0.0);
64     resVar(1,1) = 1.0;
65     resVar(1,2) = 0.0;
66     resVar(2,1) = 0.0;
67     resVar(2,2) = 1.0;
68     mme.R = resVar;
69
```

```

70     mme.getDirectSolution();
71     mme.display();
72
73     }
74     catch (matvec::exception &ex) {
75         cerr << ex.what() << "\n";
76         exit(1);
77     }
78     catch (...) {
79         cerr << "other exceptions were caught\n";
80         exit(1);
81     }
82 }

```

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 transferred 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 first MQTL model term is calculated the usual way, and for all subsequent model terms the start value from the first model term is used.

```

180 void MME::calcStarts(){
181     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 member “regCoeff” in “MME::inputData” (lines 161–164).

In order to add Σ_v^{-1} to the LHS of the MME, the following “if block” was

added to CovBlock::addGinv:

```
489     if(myMQTLPtr){
490         myMQTLPtr->addGinv(ModelTerm::myMMEPtr->lhs,
491                             myMQTLPtr->myStart,
492                             myMQTLPtr->myStart,
493                             1.0/myMQTLPtr->variance);
494     }
495     return;
```

4.4 Listing of MAPLUP programs

4.4.1 Listing of MQTL.h

```
1  #ifndef MQTL_H
2  #define MQTL_H
3  #include <fstream>
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>
7  #include "ped.h"
8  #include "util.h"
9
10 using namespace std;
11
12 class QNode {
13 public:
14     unsigned mLevel, pLevel;
15     double   mPDQ,   pPDQ, f;
16
17     QNode(){
18         mLevel = 0;
19         pLevel = 0;
20         f=-1;
21     }
22 };
23
24 class MQTL:public vector<QNode*> {
```

```
25 public:
26     unsigned count;
27     MQTL(Pedigree& P){myPedPtr = &P; myStart = -1;}
28     static Pedigree* myPedPtr;
29     SparseCij SpCij;
30     string MQTLNames;
31     matvec::Vector<double> regCoeff;
32     Recoder<string> myRecoder;
33     double variance;
34     int myStart;
35
36     void inputPDQ(char *filename);
37     void generateMQTLLevels(void);
38     void codeMQTL(unsigned i);
39     void calcMQTLInbreeding(void);
40     double getMQTLrij(unsigned i,
41                       unsigned j,
42                       unsigned pi, unsigned pj);
43     void display(void);
44     string getString (unsigned i);
45     void codeLevels(void);
46     void addGinv(matvec::doubleMatrix& lhs,
47                unsigned startRow,
48                unsigned statCol,
49                double ratio);
50     void addtoGinv(unsigned pos[], double q[], double v,
51                  matvec::doubleMatrix& lhs,
52                  unsigned startRow,
53                  unsigned startCol,
54                  double ratio);
55 };
56
57 #endif
58
```

4.4.2 Listing of MQTL.cpp

```

1  #include "MQTL.h"
2  #include "mmeMQTL.h"
3  #include "ped.h"
4  #include <matvec/session.h>
5  using namespace std;
6
7  Pedigree* MQTL::myPedPtr=0;
8
9  void MQTL::inputPDQ(char *filename){
10     unsigned rec=0, rec1=0;
11     QNode *ptr;
12     double pPDQ, mPDQ;
13     string indStr;
14     ifstream pdqfile;
15     pdqfile.open(filename);
16     if(!pdqfile){
17         cerr << "Couldn't open " << filename << endl;
18         exit (-1);
19     }
20     resize(myPedPtr->size(),0);
21     while (pdqfile >> indStr >> pPDQ >> mPDQ ){
22         rec++;
23         if(rec==1000){
24             cout<<".";
25             cout.flush();
26             rec1 += rec;
27             rec = 0;
28         }
29         Pedigree::iterator pedIt = myPedPtr->find(indStr);
30         if (pedIt == myPedPtr->end()) {
31             cout << indStr << " in record " << rec1
32             << " of " << filename
33             <<" not found in pedigree file \n";
34             exit(1);
35         }
36         ptr = new QNode();

```

```

37         ptr->pPDQ = pPDQ;
38         ptr->mPDQ = mPDQ;
39         (*this)[pedIt->second->ind-1] = ptr;
40     }
41     pdqfile.close();
42     generateMQTLLevels();
43     calcMQTLInbreeding();
44 }
45
46 void MQTL::generateMQTLLevels(void){
47     MQTL::iterator it;
48     count = 0;
49     Pedigree::iterator pedIt;
50     for (unsigned i=0;i<size();i++){
51         if((*this)[i]==0){
52             if (myPedPtr->pedVector[i]->sire==0 &&
53                 myPedPtr->pedVector[i]->dam==0) {
54                 QNode *ptr = new QNode();
55                 ptr->pPDQ = -1.0;
56                 ptr->mPDQ = -1.0;
57                 (*this)[i] = ptr;
58             }
59             else {
60                 cout << myPedPtr->pedVector[i]->ind_str
61                     << " not in PDQ file \n";
62                 exit(-1);
63             }
64         }
65     }
66     for (unsigned i=0;i<size();i++){
67         if ((*this)[i]->mLevel==0) {
68             codeMQTL(i);
69         }
70     }
71     codeLevels();
72 }
73 void MQTL::codeMQTL(unsigned i){
74     if ((*this)[i]->mLevel) return;           // already coded

```



```
75     int dam    = myPedPtr->pedVector[i]->dam;
76     int sire   = myPedPtr->pedVector[i]->sire;
77
78     if (sire==0) {
79         (*this)[i]->pLevel = ++count;
80     }
81     else {
82         codeMQTL(sire-1);
83
84         if((*this)[i]->pPDQ==1){
85             (*this)[i]->pLevel = (*this)[sire-1]->mLevel;
86         }
87         else if ((*this)[i]->pPDQ==0){
88             (*this)[i]->pLevel = (*this)[sire-1]->pLevel;
89         }
90         else {
91             (*this)[i]->pLevel = ++count;
92         }
93     }
94     if (dam==0) {
95         (*this)[i]->mLevel = ++count;
96     }
97     else {
98         codeMQTL(dam-1);
99
100        if((*this)[i]->mPDQ==1){
101            (*this)[i]->mLevel = (*this)[dam-1]->mLevel;
102        }
103        else if ((*this)[i]->mPDQ==0){
104            (*this)[i]->mLevel = (*this)[dam-1]->pLevel;
105        }
106        else {
107            (*this)[i]->mLevel = ++count;
108        }
109    }
110 }
```

```
111
112 void MQTL::codeLevels(void){
113     for (unsigned i=1;i<=count;i++){
114         string str = getString(i);
115         myRecorder.code(str);
116     }
117 }
118
119 string MQTL::getString(unsigned i){
120     ostringstream outputStrStream(ostringstream::out);
121     outputStrStream << i;
122     return outputStrStream.str();
123 }
124
125
126 void MQTL::display(){
127     for (unsigned i=0;i<size();i++){
128         cout << setw(10) << myPedPtr->pedVector[i]->ind_str
129             << setw(5) << (*this)[i]->pLevel
130             << setw(5) << (*this)[i]->mLevel
131             << setw(10) << (*this)[i]->f
132             << endl;
133     }
134 }
135
136 void MQTL::calcMQTLInbreeding(void){
137     SpCij.clear();
138     for (unsigned i=0;i<size();i++){
139         if(myPedPtr->pedVector[i]->sire==0 ||
140            myPedPtr->pedVector[i]->dam==0){
141             (*this)[i]->f= 0.0;
142         }
143         else{
144             unsigned ind = i+1;
145             (*this)[i]->f = getMQTLrij(ind,ind,0,1);
146         }
147     }
148 }
```

```
149
150
151 double MQTL::getMQTLrij(unsigned i,
152                          unsigned j,
153                          unsigned pi, unsigned pj){
154     if (i==0 || j==0){
155         return 0.0;
156     }
157     unsigned vi = pi ? (*this)[i-1]->pLevel : (*this)[i-1]->mLevel;
158     unsigned vj = pj ? (*this)[j-1]->pLevel : (*this)[j-1]->mLevel;
159     float x = SpCij.retrieve_cij(vi,vj);
160     if(x != -1.0) {
161         return x;
162     }
163     unsigned old, young, oldp, youngp;
164     if(i < j){
165         old = i;
166         oldp = pi;
167         young = j;
168         youngp = pj;
169     }
170     else if(j < i){
171         old = j;
172         oldp = pj;
173         young = i;
174         youngp = pi;
175     }
176     else if(pi==pj) {
177         return 1.0;
178     }
179     else {
180         old = i;
181         oldp = pi;
182         young = j;
183         youngp = pj;
184     }
185     unsigned y_parent;
186     double PDQym, PDQyp;
```

```

187     if (youngp==1){
188         y_parent = myPedPtr->pedVector[young-1]->sire;
189         PDQym = (*this)[young-1]->pPDQ;
190     }
191     else{
192         y_parent = myPedPtr->pedVector[young-1]->dam;
193         PDQym    = (*this)[young-1]->mPDQ;
194     }
195     PDQyp = 1.0 - PDQym;
196     x = PDQym*getMQTLrij(old,y_parent,oldp,0) +
197         PDQyp*getMQTLrij(old,y_parent,oldp,1);
198     SpCij.put_cij(vi,vj,x);
199     return x;
200
201 void MQTL::addGinv(matvec::doubleMatrix& lhs,
202                 unsigned startRow,
203                 unsigned startCol,
204                 double ratio){
205     MQTL::iterator it;
206     unsigned maxLevel=0;
207     double q[3];
208     unsigned pos[3];
209     double v;
210     for (unsigned i=0;i<size();i++){
211         if ((*this)[i]->pLevel > maxLevel){
212             if (myPedPtr->pedVector[i]->sire==0){ // founder
213                 pos[0] = 0;
214                 pos[1] = 0;
215                 pos[2] = (*this)[i]->pLevel;
216                 q[0]   = 0;
217                 q[1]   = 0;
218                 q[2]   = 1.0;
219                 v = 1.0;
220                 addtoGinv(pos,q,v,lhs,startRow,startCol,ratio);
221             }
222             else { // not founder
223                 unsigned dad = myPedPtr->pedVector[i]->sire;
224                 pos[0] = (*this)[dad-1]->mLevel;

```

```

225         pos[1] = (*this)[dad-1]->pLevel;
226         pos[2] = (*this)[i]->pLevel;
227         q[0]   = -(*this)[i]->pPDQ;
228         q[1]   = -(1+q[0]);
229         q[2]   = 1.0;
230         v     = 1 - q[0]*q[0]
231               - q[1]*q[1]
232               - 2*q[0]*q[1]*(*this)[dad-1]->f;
233         addtoGinv(pos,q,1.0/v,lhs,startRow,startCol,ratio);
234     }
235     maxLevel = pos[2];
236 }
237 if ((*this)[i]->mLevel > maxLevel){
238     if (myPedPtr->pedVector[i]->dam==0) { // founder
239         pos[0] = 0;
240         pos[1] = 0;
241         pos[2] = (*this)[i]->mLevel;
242         q[0]   = 0;
243         q[1]   = 0;
244         q[2]   = 1.0;
245         v     = 1.0;
246         addtoGinv(pos,q,v,lhs,startRow,startCol,ratio);
247     }
248     else { // not founder
249         unsigned mom = myPedPtr->pedVector[i]->dam;
250         pos[0] = (*this)[mom-1]->mLevel;
251         pos[1] = (*this)[mom-1]->pLevel;
252         pos[2] = (*this)[i]->mLevel;
253         q[0]   = -(*this)[i]->mPDQ;
254         q[1]   = -(1+q[0]);
255         q[2]   = 1.0;
256         v     = 1 - q[0]*q[0]
257               - q[1]*q[1]
258               - 2*q[0]*q[1]*(*this)[mom-1]->f;
259         addtoGinv(pos,q,1.0/v,lhs,startRow,startCol,ratio);
260     }
261     maxLevel = pos[2];
262 }

```

```

263     }
264 }
265
266 void MQTL::addtoGinv(unsigned pos[], double q[], double v,
267                    matvec::doubleMatrix& lhs,
268                    unsigned startRow,
269                    unsigned startCol,
270                    double ratio){
271     double vi,vj;
272     unsigned ii,jj;
273     for (unsigned i=0;i<3;i++){
274         if(pos[i]){
275             ii = startRow + pos[i] - 1;
276             vi = q[i]*v;
277             for (unsigned j=0;j<3;j++){
278                 if(pos[j]) {
279                     vj = q[j];
280                     jj = startCol + pos[j] - 1;
281                     if(MME::solMethod=="direct"){
282                         lhs[ii][jj] += ratio*vi*vj;
283                     }
284                     else{
285                         MME::res[ii] += ratio*vi*vj * (*MME::vec)[jj];
286                         if(ii==jj) MME::diag[ii] += ratio*vi*vj;
287                     }
288                 }
289             }
290         }
291     }
292 }

```

4.4.3 Listing of mmeMQTL.h

```

1 #ifndef MME_H
2 #define MME_H
3 #include <fstream>
4 #include <iostream>
5 #include <iomanip>

```

```
6 #include <string>
7 #include <sstream>
8 #include <stdarg.h>
9 #include <stdlib.h>
10 #include <cmath>
11 #include <map>
12 #include <matvec/doublematrix.h>
13 #include <matvec/vector.h>
14 #include <matvec/session.h>
15 #include "util.h"
16 #include "ped.h"
17 #include "MQTL.h"
18
19
20 // classes for multiple trait, mixed models with iteration on data
21
22 using namespace std;
23
24 class TermData{
25 public:
26     double value;
27     unsigned level;
28 };
29
30 class DataNode{
31 public:
32     vector<TermData > trmVec;
33     vector<double>     depVec;
34 };
35
36 class MME;
37
38 class ModelTerm{
39 public:
40     unsigned start;
41     unsigned trait;
42     string name;
43     string depVarName;
```

```

44     static MME *myMMEPtr;
45     Recoder<string>* myRecoderPtr;
46     MQTL* myMQTLPtr;
47     bool secondMQTLEffect;
48     vector<unsigned> factors;
49
50     unsigned code(string str){return myRecoderPtr->code(str);}
51     unsigned nLevels(){return myRecoderPtr->size();}
52     void putFactors(string str);
53     string  getTermString();
54     unsigned getTermLevel (){
55         return code(getTermString());
56     }
57     double  getTermValue ();
58 };
59
60 class CovBlock {
61 public:
62     vector<ModelTerm*> modelTrmPtrVec;
63     matvec::doubleMatrix Var, Vari;
64     Pedigree* pedPtr;
65     MQTL* myMQTLPtr;
66     CovBlock(void){pedPtr = 0;}
67     CovBlock(string str, matvec::doubleMatrix V){
68         Var = V;
69         buildModelTrmVec(str);
70     }
71     CovBlock(string str, matvec::doubleMatrix V, Pedigree &P){
72         Var = V;
73         pedPtr = &P;
74         myMQTLPtr = 0;
75         buildModelTrmVec(str);
76     }
77     CovBlock(MQTL &mQTL){
78         pedPtr = 0;
79         myMQTLPtr = &mQTL;
80     }
81     void buildModelTrmVec(string str);

```



```

82         void addGinv(void);
83     };
84
85
86     class MME {
87     private:
88         void putModel(string str);
89     public:
90         static string solMethod;
91         static matvec::Vector<double> *vec, diag, res;
92
93         string fileName;
94         Tokenizer colType;
95         Tokenizer colName;
96         Tokenizer depVar;
97         Tokenizer colData;
98         unsigned numCols;
99         vector <ModelTerm> modelTrmVec;
100        vector <CovBlock> covBlockVec;
101        vector <DataNode> dataVec;
102        vector<MQTL*> MQTLVec;
103        unsigned numTerms, numTraits;
104        unsigned mmeSize;
105        matvec::doubleMatrix lhs, R, Ri;
106        matvec::Vector<double> rhs, sol, tempSol;
107
108        void putColNames(string str);
109        void putColTypes(string str);
110        void putModels(string str);
111        void putVarCovMatrix(string str, matvec::doubleMatrix V, Pedigree &P){
112            CovBlock covBlock(str,V,P);
113            covBlockVec.push_back(covBlock);
114        }
115        void putVarCovMatrix(string str, matvec::doubleMatrix V){
116            CovBlock covBlock(str,V);
117            covBlockVec.push_back(covBlock);
118        }
119        void putVarCovMatrix(MQTL &Q){

```

```

120             CovBlock covBlock(Q);
121             covBlockVec.push_back(covBlock);
122         }
123         void initSetup();
124         void inputData();
125         void displayData();
126         static double getDouble(string& Str);
127         void calcStarts();
128         void getDirectSolution();
129         void getJacobiSolution(double p);
130         void getCGSolution();
131         void getPCCGSolution();
132
133         void mmeTimes(matvec::Vector<double>& x);
134         void calcWPW();
135         void addGinv();
136         void display();
137
138         void putMQTL(MQTL& Q){MQTLVec.push_back(&Q);}
139         void mergeMQTLLevelsBy(string mergeStr);
140         void addColNames(string str);
141         void addColTypes(string str);
142         void putMQTLStuffInModelTerms(MQTL& mQTL);
143         void processMQTL(void);
144     };
145 #endif
146

```

4.4.4 Listing of mmeMQTL.cpp

```

1  /*
2  *  mme1.cpp
3  *  C++WkShp
4  *
5  *  Created by Rohan Fernando on 5/6/05.
6  *  Copyright 2005, All rights reserved.
7  *
8  */

```

```
9
10 #include "mmeMQTL.h"
11
12 MME* ModelTerm::myMMEPtr;
13 string MME::solMethod;
14 matvec::Vector<double> *MME::vec, MME::diag, MME::res;
15
16 void ModelTerm::putFactors(string str){
17     Tokenizer tokens;
18     string sep("*");
19     tokens.getTokens(str,sep);
20     factors.clear();
21     for (unsigned i=0;i<tokens.size();i++){
22         unsigned factorIndex = myMMEPtr->colName.getIndex(tokens[i]);
23         if (factorIndex == -1){
24             cerr <<"Independent Variable "
25                 << tokens[i]
26                 << " not in list of column names \n";
27             exit(-1);
28         }
29         else {
30             factors.push_back(factorIndex);
31         }
32     }
33 }
34
35
36 string ModelTerm::getTermString(){
37     unsigned numFactors = factors.size();
38     string trmStr;
39     unsigned factorIndex = factors[0];
40     if(myMMEPtr->colType[factorIndex]=="COV"){
41         trmStr = myMMEPtr->colName[factorIndex];
42     }
43     else {
44         trmStr = myMMEPtr->colData[factorIndex];
45     }
46     for (unsigned i=1;i<numFactors;i++){
```

```
47         factorIndex = factors[i];
48         if(myMMEPtr->colType[factorIndex]=="COV"){
49             trmStr += "*" + myMMEPtr->colName[factorIndex];
50         }
51         else{
52             trmStr += "*" + myMMEPtr->colData[factorIndex];
53         }
54     }
55     return trmStr;
56 }
57
58 double ModelTerm::getTermValue(){
59     unsigned numFactors = factors.size();
60     double value = 1.0;
61     for (unsigned i=0;i<numFactors;i++){
62         unsigned factorIndex = factors[i];
63         if (myMMEPtr->colType[factorIndex]=="COV"){
64             string covStr = myMMEPtr->colData[factorIndex];
65             value *= MME::getDouble(covStr);
66         }
67     }
68     return value;
69 }
70
71 void MME::putColNames(string str){
72     string sep(" ");
73     str = "intercept " + str;
74     colName.getTokens(str,sep);
75     numCols = colName.size();
76 }
77
78 void MME::putColTypes(string str){
79     string sep(" ");
80     str = "CLASS " + str;
81     colType.getTokens(str,sep);
82     if (numCols!=colType.size()){
83         cerr <<"number of column names and column types do not match\n";
84         exit (-1);

```

```
85     }
86     unsigned n = 0;
87     for (unsigned i=0;i<numCols;i++){
88         if (colType[i] == "DEP") {
89             depVar.push_back(colName[i]);
90             n++;
91         }
92     }
93     numTraits = n;
94 }
95
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
109 void MME::putModel(string str){
110     string sep(" =+");
111     Tokenizer modelTokens;
112     modelTokens.getTokens(str,sep);
113     unsigned nTokens = modelTokens.size();
114     int depVarIndex = colName.getIndex(modelTokens[0]);
115     if (depVarIndex == -1){
116         cerr << "Dependent Variable "
117             << modelTokens[0]
118             << " not in list of column names \n";
119         exit (-1);
120     }
121     ModelTerm modelTrm;
122     modelTrm.secondMQTLEffect = false;
```

```
123     modelTrm.myMQTLPtr = 0;
124     modelTrm.depVarName = modelTokens[0];
125     modelTrm.trait = depVar.getIndex(modelTokens[0]);
126     for (unsigned i=1;i<nTokens;i++){
127         modelTrm.myRecoderPtr = new Recoder<string>;
128         modelTrm.name = modelTokens[i];
129         modelTrm.putFactors(modelTrm.name);
130         modelTrmVec.push_back(modelTrm);
131     }
132 }
133
134 void MME::inputData(){
135     DataNode dataNode;
136     numTerms = modelTrmVec.size();
137     dataNode.trmVec.resize(numTerms);
138     dataNode.depVec.resize(numTraits);
139     ifstream datafile;
140     datafile.open(fileName.c_str());
141     if(!datafile) {
142         cerr << "Couldn't open data file: " << fileName << endl;
143         exit (-1);
144     }
145     unsigned linewidth = 1024;
146     char *line = new char [linewidth];
147     string sep(" ");
148     while (datafile.getline(line,linewidth)){
149         string inputStr(line);
150         inputStr = "--- " + inputStr;
151         colData.getTokens(inputStr,sep);
152         unsigned j=0;
153         for (unsigned i=0;i<numCols;i++){
154             if (colType[i]=="DEP") {
155                 dataNode.depVec[j++] = getDouble(colData[i]);
156             }
157         }
158     }
159     for (unsigned i=0;i<numTerms;i++){
160         dataNode.trmVec[i].level = modelTrmVec[i].getTermLevel();
```

```

161         if (modelTrmVec[i].myMQTLPtr){
162             unsigned tr = modelTrmVec[i].trait;
163             dataNode.trmVec[i].value = modelTrmVec[i].myMQTLPtr->regCoeff[tr];
164         }
165         else {
166             dataNode.trmVec[i].value = modelTrmVec[i].getTermValue();
167         }
168     }
169     dataVec.push_back(dataNode);
170 }
171 }
172
173 double MME::getDouble(string& Str) {
174     istringstream inputStrStream(Str.c_str());
175     double val;
176     inputStrStream >> val;
177     return val;
178 }
179
180 void MME::calcStarts(){
181     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 }
206
207 void MME::calcWPW(){
208     unsigned ii,jj,ti,tj;
209     double vi,vj,tr_value;
210     if(solMethod!="direct"){
211         diag.resize(mmeSize,0.0);
212         res.resize(mmeSize,0.0);
213     }
214     rhs.resize(mmeSize,0.0);
215     Ri = R.inv();
216     for (unsigned i=0;i<dataVec.size();i++){
217         for (unsigned mi=0;mi<numTerms;mi++){
218             ii = modelTrmVec[mi].start + dataVec[i].trmVec[mi].level - 1;
219             ti = modelTrmVec[mi].trait;
220             vi = dataVec[i].trmVec[mi].value;
221             for (unsigned k=0;k<numTraits;k++){
222                 rhs[ii] += vi*Ri[ti][k]*dataVec[i].depVec[k];
223             }
224             for (unsigned mj=0;mj<numTerms;mj++){
225                 jj = modelTrmVec[mj].start + dataVec[i].trmVec[mj].level - 1;
226                 tj = modelTrmVec[mj].trait;
227                 vj = dataVec[i].trmVec[mj].value;
228                 if (solMethod=="direct"){
229                     lhs[ii][jj] += vi*Ri[ti][tj]*vj;
230                 }
231                 else {
232                     res[ii] += vi*Ri[ti][tj]*vj * (*vec)[jj];
233                     if(ii==jj) diag[ii] += vi*Ri[ti][tj]*vj;
234                 }
235             }
236         }

```



```
237     }
238 }
239
240 void MME::addGinv(){
241     for (unsigned i=0;i<covBlockVec.size();i++){
242         covBlockVec[i].addGinv();
243     }
244 }
245
246 void MME::initSetup(){
247     inputData();
248     calcStarts();
249     if(solMethod=="direct"){
250         lhs.resize(mmeSize,mmeSize,0.0);
251     }
252     else {
253         sol.resize(mmeSize,0.0);
254     }
255 }
256
257
258 void MME::getDirectSolution(){
259     solMethod = "direct";
260     initSetup();
261     calcWPW();
262     addGinv();
263     sol = lhs.ginv0()*rhs;
264 }
265
266 void MME::display(){
267     if (solMethod=="direct"){
268         cout << "LHS " << endl;
269         for (unsigned i = 0;i<mmeSize;i++){
270             for (unsigned j = 0;j<mmeSize;j++){
271                 cout << setw(8) << setprecision (3)
272                     << setiosflags (ios::right | ios::fixed)
273                     << lhs[i][j] <<" ";
274             }

```

```

275         cout << endl;
276     }
277 }
278 cout << "RHS " << endl;
279 cout << rhs << endl;
280 for (unsigned i=0;i<modelTrmVec.size();i++){
281     cout << "Solutions for " << modelTrmVec[i].name
282         << ", Trait: " << modelTrmVec[i].depVarName << endl;
283     Recorder<string>::iterator it;
284     for (it=modelTrmVec[i].myRecorderPtr->begin();
285         it!=modelTrmVec[i].myRecorderPtr->end();
286         it++){
287         unsigned ii = modelTrmVec[i].start + it->second - 1;
288         cout << setw(10) << it->first << " " << sol[ii] << endl;
289     }
290 }
291 }
292
293 void MME::getJacobiSolution(double p){
294     solMethod = "jacobi";
295     initSetup();
296     mmeTimes(sol); // result goes into res, also creates rhs and diag
297     matvec::Vector<double> resid = (rhs - res);
298     tempSol = resid/diag + sol;
299     double diff = resid.sumsq();
300     unsigned iter = 0;
301     while(diff/mmeSize > .0000001 && ++iter<200){
302         sol = p*tempSol + (1-p)*sol;
303         cout <<"Iteration : "
304             << iter <<" diff = "
305             <<diff/mmeSize << endl << endl;
306         mmeTimes(sol);
307         resid = (rhs - res);
308         tempSol = resid/diag + sol;
309         diff = resid.sumsq();
310     }
311     cout << resid << endl;
312 }

```

```
313
314 void MME::getCGSolution(){
315     solMethod = "cg";
316     initSetup();
317     mmeTimes(sol); // result goes into res, also creates rhs and diag
318     matvec::Vector<double> resid = (res-rhs);
319     matvec::Vector<double> d = resid;
320     double oldDiffSq;
321     double newDiffSq = resid.sumsq();
322     unsigned iter = 0;
323     while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
324         cout <<"Iteration : "
325             << iter
326             <<" diff = "
327             <<newDiffSq/mmeSize << endl << endl;
328         mmeTimes(d);
329         double alpha = -newDiffSq/(res*d).sum();
330         sol += alpha*d;
331         if (iter%10){
332             resid += alpha*res;
333         }
334         else {
335             mmeTimes(sol);
336             resid = (res-rhs);
337         }
338         oldDiffSq = newDiffSq;
339         newDiffSq = resid.sumsq();
340         double beta = -newDiffSq/oldDiffSq;
341         d = resid - beta*d;
342     }
343     cout << resid << endl;
344 }
345
346 void MME::getPCCGSolution(){
347     solMethod = "pccg";
348     initSetup();
349     mmeTimes(sol); // result goes into res, also creates rhs and diag
350     matvec::Vector<double> resid = (res - rhs);
```

```
351         matvec::Vector<double> d = resid/diag;
352         double oldDiffSq;
353         double newDiffSq = (resid*d).sum();
354         unsigned iter = 0;
355         while(newDiffSq/mmeSize > .000001 && ++iter<2*mmeSize){
356             cout <<"Iteration : "
357                 << iter <<" diff = "
358                 <<newDiffSq/mmeSize
359                 << endl << endl;
360             mmeTimes(d);
361             double alpha = -newDiffSq/(res*d).sum();
362             sol += alpha*d;
363             if (iter % 10){
364                 resid += alpha*res;
365             }
366             else {
367                 mmeTimes(sol);
368                 resid = (res - rhs);
369             }
370             matvec::Vector<double> s = resid/diag;
371             oldDiffSq = newDiffSq;
372             newDiffSq = (resid*s).sum();
373             double beta = -newDiffSq/oldDiffSq;
374             d = s - beta*d;
375         }
376         cout << resid << endl;
377     }
378
379     void MME::mmeTimes(matvec::Vector<double>& x){
380         vec = &x;
381         calcWPW();
382         addGinv();
383     }
384
385     void MME::mergeMQTLLevelsBy(string mergeStr){
386         unsigned numColsInFile = numCols - 1;
387         int indexMergeStr = colName.getIndex(mergeStr) - 1;
388         if(indexMergeStr == -1){
```

```

389     cerr << mergeStr <<" not in Column Names \n";
390     exit(-1);
391 }
392 ifstream datafile;
393 ofstream outfile;
394 string outFileName = fileName + ".mrgMQTL";
395 datafile.open(fileName.c_str());
396 if(!datafile) {
397     cerr << "Couldn't open data file: " << fileName << endl;
398     exit (-1);
399 }
400 outfile.open(outFileName.c_str());
401 if(!outfile) {
402     cerr << "Couldn't open file: " << outFileName << endl;
403     exit (-1);
404 }
405 size_t linewidth = 1024;
406 char *line = new char [linewidth];
407 string sep(" ");
408 unsigned lineNumber=0;
409 while (datafile.getline(line,linewidth)){
410     lineNumber++;
411     string inputStr(line);
412     colData.getTokens(inputStr,sep);
413     unsigned n = colData.size();
414     if (n != numColsInFile){
415         cerr << "Line " << lineNumber << " of data file has " << n <<" columns" << endl;
416         cerr << numColsInFile <<" expected " << endl;
417         exit(-1);
418     }
419     string mergeVar = colData[indexMergeStr];
420     int mergeID = MQTLVec[0]->myPedPtr->coder.code(mergeVar);
421     if (mergeID>MQTLVec[0]->myPedPtr->size()){
422         cerr << "Line " << lineNumber << " of data file has merge string "<<mergeVar
423         cerr << "which is not in Pedigree file\n";
424     }
425     outfile << inputStr << " ";
426     for (unsigned i=0;i<MQTLVec.size();i++){

```

```
427         outfile << (*MQTLVec[i])[mergeID-1]->pLevel << " "
428                 << (*MQTLVec[i])[mergeID-1]->mLevel << " ";
429     }
430     outfile << endl;
431 }
432 datafile.close();
433 outfile.close();
434 fileName = outFileNames;
435 }
436
437 void MME::addColNames(string str){
438     string sep(" ");
439     Tokenizer tokens;
440     tokens.getTokens(str,sep);
441     Tokenizer::iterator it;
442     for(it=tokens.begin();it!=tokens.end();it++){
443         colName.push_back(*it);
444     }
445     numCols = colName.size();
446 }
447
448 void MME::addColTypes(string str){
449     string sep(" ");
450     Tokenizer tokens;
451     tokens.getTokens(str,sep);
452     Tokenizer::iterator it;
453     for(it=tokens.begin();it!=tokens.end();it++){
454         colType.push_back(*it);
455     }
456     if (numCols!=colType.size()){
457         cerr <<"number of column names and column types do not match\n";
458         exit (-1);
459     }
460 }
461
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 }
481
482 void CovBlock::buildModelTrmVec(string str){
483     string sep(" ");
484     Tokenizer modelTokens;
485     modelTokens.getTokens(str,sep);
486     unsigned nTokens = modelTokens.size();
487     unsigned numModelTrms = ModelTerm::myMMEPtr->modelTrmVec.size();
488     for (unsigned i=0;i<nTokens;i++){
489         for (unsigned j=0;j<numModelTrms;j++){
490             if (modelTokens[i]==ModelTerm::myMMEPtr->modelTrmVec[j].name){
491                 modelTrmPtrVec.push_back(&(ModelTerm::myMMEPtr->modelTrmVec[j]));
492                 if (pedPtr){
493                     delete ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr;
494                     ModelTerm::myMMEPtr->modelTrmVec[j].myRecoderPtr = &pedPtr->code1;
495                 }
496             }
497         }
498     }
499 }
500
501 void CovBlock::addGinv(void){
502     if(myMQTLPtr){

```

```

503     myMQTLPtr->addGinv(ModelTerm::myMMEPtr->lhs,
504                       myMQTLPtr->myStart,
505                       myMQTLPtr->myStart,
506                       1.0/myMQTLPtr->variance);
507     return;
508 }
509 Vari = Var.inv();
510 unsigned n = modelTrmPtrVec.size();
511 for (unsigned i=0;i<n;i++){
512     ModelTerm* mtermiPtr = modelTrmPtrVec[i];
513     unsigned starti = mtermiPtr->start;
514     for (unsigned j=0;j<n;j++){
515         ModelTerm* mtermjPtr = modelTrmPtrVec[j];
516         unsigned startj = mtermjPtr->start;
517         if (pedPtr){
518             pedPtr->addAinv(ModelTerm::myMMEPtr->lhs,starti,startj,Vari[i][j]);
519         }
520
521         else{
522             unsigned numLevels = mtermiPtr->nLevels();
523             for (unsigned k=0;k<numLevels;k++){
524                 unsigned ii = starti + k;
525                 unsigned jj = startj + k;
526                 if (MME::solMethod=="direct") {
527                     ModelTerm::myMMEPtr->lhs[ii][jj] += Vari[i][j];
528                 }
529                 else {
530                     MME::res[ii] += Vari[i][j] * (*MME::vec)[jj];
531                     if (ii==jj) MME::diag[ii] += Vari[i][j];
532                 }
533             }
534         }
535     }
536 }
537 }
538
539

```