BayesC0

Simulating Genotypes and Phenotypes

In [31]:
```
using(Distributions)
```

In [2]:
```
nObs     = 100
nMarkers = 1000
X = sample([0,1,2],(nObs,nMarkers))
α  = randn(nMarkers)
a  = X*α
stdGen = std(a)
a = a/stdGen
y = a + randn(nObs)
saveAlpha = α
nothing
```

Centering Genotype Covariates

In [3]:
```
meanXCols = mean(X,1)
X = X - ones(nObs,1)*meanXCols;
```

Priors

In [4]:
```
seed            =   10    # set the seed for the random number generator
chainLength     =  2000   # number of iterations
probFixed       =    0    # parameter "pi" the probability SNP effect is z
dfEffectVar     =    4    # hyper parameter (degrees of freedom) for locus
nuRes           =    4    # hyper parameter (degrees of freedom) for resic
varGenotypic    =    1    # used to derive hyper parameter (scale) for loc
varResidual     =    1    # used to derive hyper parameter (scale) for loc
scaleVar        =   varGenotypic*(dfEffectVar-2)/dfEffectVar    # scale fa
scaleRes        =   varResidual*(nuRes-2)/nuRes                 # scale fa
nothing
```

Function for Sampling Marker Effects

In [5]:
```
function get_column(X,nRows,j)
    indx = 1 + (j-1)*nRows
    ptr = pointer(X,indx)
    pointer_to_array(ptr,nRows)
end
```

Out[5]:
```
get_column (generic function with 1 method)
```

In [6]:
```
xpx = [(X[:,i]'X[:,i])[1]::Float64 for i=1:nMarkers]
xArray = Array(Array{Float64,1},nMarkers)
for i=1:nMarkers
    xArray[i] = get_column(X,nObs,i)
end
```

In [7]:
```
typeof(xArray[1])
```

Out[7]:
```
Array{Float64,1}
```

Computing the adjusted right-hand-side efficiently

We want to compute:

$$rhs = \mathbf{X}_j'(\mathbf{y}_{corr} + \mathbf{X}_j\alpha_j)$$

This is more efficiently obtained as

$$rhs = \mathbf{X}_j'\mathbf{y}_{corr} + \mathbf{X}_j'\mathbf{X}_j\alpha_j,$$

using the diagonals of $\mathbf{X}'\mathbf{X}$ that have already been computed (line 4 of the function below).

In [19]:
```
 1 function sampleEffects!(nMarkers,xArray,xpx,yCorr,α,meanAlpha,vare,var
 2     nObs = size(X,1)
 3     for j=1:nMarkers
 4         rhs::Float64 = dot(xArray[j],yCorr) + xpx[j]*α[j]
 5         lhs::Float64      = xpx[j] + vare/varEffects
 6         invLhs::Float64   = 1.0/lhs
 7         mean::Float64     = invLhs*rhs
 8         oldAlpha::Float64 = α[j]
 9         α[j] = mean + randn()*sqrt(invLhs*vare)
10         BLAS.axpy!(oldAlpha-α[j],xArray[j],yCorr)
11     end
12     nothing
13 end
```

Out[19]:
```
sampleEffects! (generic function with 1 method)
```

Function for BayesC0

The intercept is sampled first and the sampleEffects! function is called to sample the marker effects

In [10]:
```
chi1=Chisq(nObs+nuRes)
chi2=Chisq(dfEffectVar+nMarkers)

function BayesC0!(numIter,nMarkers,X,xpx,yCorr,mu,meanMu,α,meanAlpha,vare,
    for i=1:numIter
        # sample residula variance
        vare = (dot(yCorr,yCorr)+nuRes*scaleRes)/rand(chi1)

        # sample intercept
        yCorr  = yCorr+mu
        rhs    = sum(yCorr)
        invLhs = 1.0/(nObs)
        mean   = rhs*invLhs
        mu     = mean + randn()*sqrt(invLhs*vare)
        yCorr  = yCorr - mu
        meanMu = meanMu + (mu - meanMu)/i

        # sample effects
        sampleEffects!(nMarkers,xArray,xpx,yCorr,α,meanAlpha,vare,varEffec
        meanAlpha = meanAlpha + (α - meanAlpha)/i

        #sameple locus effect variance
        varEffects = (scaleVar*dfEffectVar + dot(α,α))/rand(chi2)

        if (i%1000)==0
            yhat = meanMu+X*meanAlpha
            resCorr = cor(a,yhat)
            println ("Correlation of between true and predicted breeding v
        end
    end
end
```

Out[10]: BayesC0! (generic function with 1 method)

Run BayesC0

In [30]:
```
meanMu    = 0
meanAlpha = zeros(nMarkers)

#initial valus
vare = 1
varEffects = 1
mu = mean(y)
yCorr = y - mu
alpha = fill(0.0,nMarkers)

#run it
@time BayesC0!(chainLength,nMarkers,X,xpx,yCorr,mu,meanMu,alpha,meanAlpha,
```

```
Correlation of between true and predicted breeding value: 0.77452987300536
Correlation of between true and predicted breeding value: 0.77472194735639
elapsed time: 0.213988087 seconds (53211392 bytes allocated, 12.66% gc tim
```

Compare Runtime with R Implementation

In [18]:
```
;Rscript RBayesC0/BayesC0.R
```

```
   user  system elapsed
 50.936   1.524  52.569
```

In [32]:
```
;cat RBayesC0/BayesC0.R
```

```
#   This code is for illustrative purposes and not efficient for large pro
#   Real life data analysis (using the same file formats) is available at
#   bigs.ansci.iastate.edu/login.html based on GenSel cpp software impleme
#
#               Rohan Fernando      (rohan@iastate.edu)
#               Dorian Garrick      (dorian@iastate.edu)
#               copyright August 2012

# Parameters
setwd("RBayesC0")
seed            =   10    # set the seed for the random number generator
chainLength     =  2000    # number of iterations
dfEffectVar     =    4    # hyper parameter (degrees of freedom) for locus
nuRes           =    4    # hyper parameter (degrees of freedom) for resid
varGenotypic    =    1    # used to derive hyper parameter (scale) for loc
varResidual     =    1    # used to derive hyper parameter (scale) for res
windowSize      =   10    # number of consecutive markers in a genomic win
outputFrequency =   100    # frequency for reporting performance and for c

markerFileName         = "genotypes.dat"
trainPhenotypeFileName = "trainPhenotypes.dat"
testPhenotypeFileName  = "testPhenotypes.dat"
```

```
set.seed(seed)

genotypeFile         = read.table(markerFileName, header=TRUE)
trainPhenotypeFile   = read.table(trainPhenotypeFileName, skip=1)[,1:2]
testPhenotypeFile    = read.table(testPhenotypeFileName,  skip=1)[,1:2]
commonTrainingData   = merge(trainPhenotypeFile, genotypeFile, by.x=1, by.
ype
commonTestData       = merge(testPhenotypeFile,  genotypeFile, by.x=1, by.
ype


remove(genotypeFile)                                                  # Free
remove(trainPhenotypeFile)                                            # Free
remove(testPhenotypeFile)                                             # Free
animalID = unname(as.matrix(commonTrainingData[,1]))                  # Firs
y        = commonTrainingData[, 2]                                    # Seco
Z        = commonTrainingData[, 3: ncol(commonTrainingData)]          # Rema
Z        = unname(as.matrix((Z + 10)/10));                            # Reco
markerID = colnames(commonTrainingData)[3:ncol(commonTrainingData)] # Reme
remove(commonTrainingData)

testID = unname(as.matrix(commonTestData[,1]))                    # First fi
yTest        = commonTestData[, 2]                                # Second f
ZTest        = commonTestData[, 3: ncol(commonTestData)]          # Remainin
ZTest        = unname(as.matrix((ZTest + 10)/10));                # Recode g
remove(commonTestData)

nmarkers = ncol(Z)                                                # number o
nrecords = nrow(Z)                                                # number o

# center the genotype matrix to accelerate mixing
markerMeans = colMeans(Z)                                  # compute the mean f
Z = t(t(Z) - markerMeans)                                  # deviate covariate
p = markerMeans/2.0                                        # compute frequency
mean2pq = mean(2*p*(1-p))                                  # compute mean genot

varEffects  = varGenotypic/(nmarkers*mean2pq)              # variance of locus
                                                           #(e.g. Fernando et a
192-195)
scaleVar    = varEffects*(dfEffectVar-2)/dfEffectVar; # scale factor for l
scaleRes    = varResidual*(nuRes-2)/nuRes              # scale factor for r

numberWindows = nmarkers/windowSize                        # number of genomic
numberSamples = chainLength/outputFrequency                # number of samples


alpha           = array(0.0, nmarkers) # reserve a vector to store sampled
meanAlpha       = array(0.0, nmarkers) # reserve a vector to accumulate th
modelFreq       = array(0.0, nmarkers) # reserve a vector to store model f
```

```
mu              = mean(y)              # starting value for the location p
meanMu          = 0                    # reserve a scalar to accumulate th
geneticVar      = array(0,numberSamples) # reserve a vector to store sampl
                                       # reserve a matrix to store sampled
windowVarProp   = matrix(0,nrow=numberSamples,ncol=numberWindows)
sampleCount     = 0                    # initialize counter for number of



# adjust y for the fixed effect (ie location parameter)
ycorr = y - mu


ZPZ=t(Z)%*%Z
zpz=diag(ZPZ)

ptime=proc.time()
# mcmc sampling
for (iter in 1:chainLength){

# sample residual variance
        vare = ( t(ycorr)%*%ycorr + nuRes*scaleRes )/rchisq(1,nrecords + n

# sample intercept
        ycorr = ycorr + mu                    # Unadjust y for the previou
        rhs    = sum(ycorr)                   # Form X'y
        invLhs = 1.0/nrecords                 # Form (X'X)-1
        mean = rhs*invLhs                     # Solve (X'X) mu = X'y
        mu = rnorm(1,mean,sqrt(invLhs*vare))  # Sample new location parame
        ycorr = ycorr - mu                    # Adjust y for the new sampl
        meanMu = meanMu + mu                  # Accumulate the sum to comp

# sample effect for each locus
        for (locus in 1:nmarkers){

                rhs=t(Z[,locus])%*%ycorr +zpz[locus]*alpha[locus]
                mmeLhs = zpz[locus] + vare/varEffects
                invLhs = 1.0/mmeLhs                                  # In
                mean = invLhs*rhs                                    # So
                oldAlpha=alpha[locus]
                alpha[locus]= rnorm(1,mean,sqrt(invLhs*vare))        # Sa
                ycorr = ycorr + Z[,locus]*(oldAlpha-alpha[locus]);
                meanAlpha[locus] = meanAlpha[locus] + alpha[locus];   # Ac
        }

        # sample the common locus effect variance
        varEffects = ( scaleVar*dfEffectVar + sum(alpha^2) )/rchisq(1,dfEf

}
```

```
proc.time()-ptime
```