Step-by-Step Procedures for Constructing Gene Regulatory Networks Using RNA-Seq Data

Chen Chen, Min Ren, Douglas G. Crabill, Dabao Zhang

Fall 2018

Contents

1 Example Data

Caution: The codes should be modified to include genes which do not have cis-eQTL, so as to only identify their regulatory genes.

Caution: This example datasets are available in our sever `morgan:/work/example/tspls/`.

The gene regulatory network will be constructed on the basis of a set of gene expression data, and a set of genotype data in *Merlin* format (`.map` & `.ped` files) for a total of n individuals:

- Gene Expression Data (`GeneExpression/jpt.ge` & `GeneExpression/jpt.gpos`):
  - `jpt.ge` includes the sample IDs in the first column, and the rest is an $n \times p$ matrix with p genes for each of n individuals;
  - `jpt.gpos` is a gene annotation file, including four columns with the first one for *Gene Symbol*, the second one for *Chromosome No.*, the third for *Start Position*, and the last for *End Position*.
- Genotype Data (`Genotype/jpt.map` & `Genotype/jpt.ped`):
  - `jpt.ped` includes pedgree information, i.e., [family_ID individual_ID mother_ID father_ID gender phenotype] in the first six columns, followed by $2p$ columns with two columns for each of p SNPs.
  - `jpt.map` includes SNP location information with four columns, i.e., [chromosome SNP_name genetic_distance locus] for each of p SNPs.

Caution: For users who would like to analyze their data sets, the four files, i.e., `jpt.ge`, `jpt.gpos`, `jpt.map` and `jpt.ped`, should be replaced by their own datasets (with `.map` & `.ped` files in *Merlin* formats) accordingly. Put the first two inside the folder `GeneExpression`, and the last two inside the folder `Genotype`.

Caution: With more and more genotype data in *VCF* or *BCF* format, `Mega2` can be used to convert the files to *Merlin* format files. Or use the most recent version of `PLINK` to directly preprocess the files in *VCF* or *BCF* format.

Caution: As this example data set is for human, 22 autosome chromosomes (excluding the sex chromosome) are included in the analysis. When adapting the codes here to analyze other organisms, the number 22 should be changed accordingly.

Future Plan: We should prepare the following three files to help set the environment of network construction,

- `settings`: a file with two columns to store settings required for the analysis. The first column includes the name of the item, and the second column includes the value of the item. For example,

item	value	meaning (not in the file)
ge.file	jpt.ge	name of gene expression file
gpos.file	jpt.gpos	name of gene position file
ped.file	jpt.ped	name of `.ped` file (genotype)
map.file	jpt.map	name of `.map` file
nchr	22	# of chromosomes
mind	0.1	*PLINK* option
geno	0.1	*PLINK* option
hwe	0.0001	*PLINK* option
recode	A	*PLINK* option
alpha.cis	0.05	α-level for cis-eQTL
uncor.ncis	3	maximum # of cis-eQTL for each gene
uncor.r	0.5	maximum corr. coeff. b/w cis-eQTL of same gene
nperms	100	# of permutations
upstream	1000	upstream region to flank the genetic region
downstream	500	downstream region to flank the genetic region
...	...	

- `data.info`: a file with two columns to save some summary statistics on the data, which may be needed during the analysis. The first column includes the name of the item, and the second column includes the value of the item. These information are usually generated during the analysis. For example,

item	value	meaning (not in the file)
n	82	sample size
ngenes	18402	# of genes included in the analysis
...	...	

- `bt.settings`: a separate setting file with two columns to set up the environment for the bootstrap stage of the analysis. The first column includes the name of the item, and the second column includes the value of the item. For example,

item	value	meaning (not in the file)
nboots	100	# of bootstrap data sets
nnodes	500	# of available nodes in the server
ncores	16	# of available cores in each node
memory	64	memory size (Gb) of each node
walltime	4	walltime of the server
...	...	

Future Plan: A function `setoption` may be written to set up (default) settings, and modify the two files, settings and bt.settings. Another function getoption should be written to check a specific setting, or all default settings in settings and bt.settings.

2 Preprocessing Genotype Data

Input Files of This Section:

- Genotype/jpt.ped: includes pedgree information, i.e., [family_ID individual_ID mother_ID father_ID gender phenotype] in the first six columns, followed by $2p$ columns with two columns for each of p SNPs;
- Genotype/jpt.map: includes SNP location information with four columns, i.e., [chromosome SNP_name genetic_distance locus] for each of p SNPs.

Firstly, we will use PLINK to filter out individuals or SNPs on the basis of several summary statistic measures, including maximum per-person missing (–mind), maximum per-SNP missing (–geno), and hardy-Weinberg disequilibrium p-value (–hwe). The SNP genotypes will be output to pre-specified *clean_Genotype.map* and *clean_Genotype.ped*, and the numbers of minor alleles per person will be saved to the file *clean_Genotype.raw*.

```
cd /work/example/tspls/Genotype
plink --noweb --file jpt --mind 0.1 --geno 0.1 --hwe 0.0001 --recode --out clean_Genotype &
```

Caution: While the above assumes the additive coding (so only additive effects will be consided later on), an output file named *clean_Genotype.raw* can also be generated to consider both addtivie and dominant effects, assuming both additive and dominance coding with the following PLINK command line:

```
cd ./Genotype
plink --noweb --file jpt --mind 0.1 --geno 0.1 --hwe 0.0001 --recodeAD --out clean_Genotype &
```

Caution: The newer version PLINK (version 1.9 or later) is much faster than the old version (v1.07).

Caution: When the genotype data includes insertions and deletions (INDELs), the newer version PLINK may be used to either include or exclude them in the following files.

We need to split genotype data by chromosomes:

```
# Split the .map file
for i in `seq 1 22`
do
echo $i
awk '/^'"$i"'\t/ {print $0}' clean_Genotype.map > clean_Genotype_chr$i.map
done

# Create the file "clean_Genotype.sizes":
for i in `seq 1 22`
do
echo -n "$i "
n=`wc -l clean_Genotype_chr$i.map | awk '{print $1}'`
echo "$n"
done > clean_Genotype.sizes

# Split the .raw file:
nohup tail -n+2 clean_Genotype.raw > clean_Genotype.data &
nohup ./splitchr.pl clean_Genotype.data clean_Genotype.sizes 22 &
```

The output files are *clean_Genotype_chrXX.map* and *clean_Genotype_chrXX.data* (each row is a sample, each column is a SNP).

For sequencing data, if we still have missing values after the Mach-Admix imputation. We can just set all missing data for a given genotype to reference homozygous (0).

```
### Replace NA with 0
for i in {1..22}
do
    sed -i -e 's/NA/0/g' clean_Genotype_chr$i.data
done
```

Caution: The above codes assume that there are 22 chromosomes to be processed.

Then we can combine genotype data across all chromosomes.

```
### Combine genotype data
paste -d' ' $(find ./ -name "clean_Genotype_chr*.data" | sort -V) > imputed_Genotype.data

### Sample ID
awk '{print $2}' clean_Genotype.data > Genotype.sampleID

### Combine sample ID with genotype data
paste -d' ' Genotype.sampleID imputed_Genotype.data > Geno
```

Output Files of This Section:

- `Genotype/Geno`: each row is a sample and each column is a SNP, with the first column for Sample ID;
- `Genotype/clean_Genotype.map`: the corresponding map file;
- `Genotype/clean_Genotype_chr$i.map`: map file for i-the chromosome, with $i = 1, 2, \cdots$

3 Preprocessing Gene Expression Data

After obtaining a gene expression data, we should

- check whether the data has been normalized;
- impute missing values if there are any;
- make PC plot to detect potential patterns, e.g., population stratification, in the data, and remove these patterns.

One approach to impute missing values is `knnimpute(1)`, which replaces missing values in the data with the corresponding value from an individual which has the best matched values of the neighboring columns (i.e., genes). Also, if we want to do gene-based analysis but the expression data is probe-based, we need to take an average of expression values of probes corresponding the same gene. Some data sets may be more complicated and need further handling.

We also need a gene annotation file to includethe chromosome positions of all genes in the gene expression file. This annotation file should be organized to include four columns: Column 1 is *Gene Symbol*, Column 2 is *Chromosome No.*, Column 3 is *Start Position*, and Column 4 is *End Position*. The genes are put in the same order as in the expression file, and are better ordered according to their physical positions on chromosomes.

```
cd /work/example/tspls/GeneExpression
cp jpt.ge Gexp
cp jpt.gpos gene_pos
```

Output Files of This Section:

- `GeneExpression/Gexp`: the gene expression file of gene expression matrix with each row for a sample, the first column for Sample ID and each of rest columns for a gene;

- `GeneExpression/gene_pos`: the gene annotation file with chromosome positions of all genes, including four columns with the first one for *Gene Symbol*, the second one for *Chromosome No.*, the third for *Start Position*, and the last for *End Position*.

Caution: The genes in the above expression and annotation files should be in the same order, and be better ordered according to their physical positions on chromosomes.

Caution: *EIGENSTRAT* may be used to construct principal components to account for possible population stratification.

4 Matching Gene Expression Data with Genotype Data

Input Files of This Section:

- `GeneExpression/Gexp`: the gene expression file of gene expression matrix with each row for a sample, the first column for Sample ID and each of rest columns for a gene;
- `GeneExpression/gene_pos`: the gene annotation file with chromosome positions of all genes, including four columns with the first one for *Gene Symbol*, the second one for *Chromosome No.*, the third for *Start Position*, and the last for *End Position*;
- `Genotype/Geno`: each row is a sample and each column is a SNP, with the first column for Sample ID;
- `Genotype/clean_Genotype.map`: the corresponding map file.

We need to match the sample IDs of the gene expression data with the genotype data. Below are the Linux commands codes (using *Rscript*):

```
cd /work/example/tspls/

### Extract genotype data that match Gexp Sample ID
cut -d' ' -f1 ./GeneExpression/Gexp > GexpID
./codes/extractrows.pl GexpID ./Genotype/Geno > EGeno

### Extract gene expression data that match Geno Sample ID
cut -d' ' -f1 ./Genotype/Geno > GenoID
./codes/extractrows.pl GenoID ./GeneExpression/Gexp > EGexp

### Extracted Geno Sample ID
cut -d' ' -f1 EGeno > EGenoID

### Order extracted gene expression data by extracted Geno Sample ID in R
Rscript ./codes/orderge.R

### Rename matched gene expression data and genotype data
mv MGexp matched.Gexp #?: mv EGexp matched.Gexp
mv EGeno matched.Geno.data #?: mv MGeno matched.Geno
```

Note that the *Rscript* file `orderge.R` takes files `EGexp` and `EGenoID` to generate the file 'MGexp'.

Output Files of This Section:

- `matched.Gexp`: the gene expression file with matched sample IDs in the first column;
- `matched.Geno.data`: the genotype file with matched sample IDs in the first column.

5 Preprocess Genotype Data (Con't)

Input Files of This Section:

- `matched.Geno.data`: the genotype file with sample IDs in the first column matched to the gene expression file `matched.Gexp`;
- `Genotype/clean_Genotype.sizes`: to be clarified.

We split matched genotype data by chromosome:

```
### Split match.snpsdata file by chromesome
nohup ./codes/splitchr.pl matched.Geno.data ./Genotype/clean_Genotype.sizes 22 &

for i in {1..22}
do
  mv Genotype/clean_Genotype_chr$i.map matched.Geno_chr$i.map
done
```

The output files are *matched.Geno_chrXX.map* and *matched.Geno_chrXX.data* (each row is a sample, each column is a SNP). We can now summarize the genotype data by chromosome:

```
### Number of minor alleles by chromosome
#!/bin/sh

echo "#!/bin/sh" > qsub.sh
chmod +x qsub.sh

for i in {1..22}
do
  perl -pe 's/XXX/'$i'/g' < ./codes/genosum.m > genosum_chr$i.m
  echo "nohup matlab -nodisplay -nodesktop -nosplash < genosum_chr$i.m > genosum_chr$i.log &" >> qsub.s
done

sh qsub.sh

find . -name "matched.Geno.ma_chr*"|sort -V|xargs paste -d' ' >matched.Geno.ma

nohup matlab -nodisplay -nodesktop -nosplash < ./codes/filterma5.m > nohup.out
```

The last *MatLab* command filters out SNPs with less than 5 minor alleles, and out the remaining to `snps5.idx`.

```
### Map file for SNPs with >=5 minor alleles => "new.Geno.idx"
Rscript ./codes/snps5.R

### Convert new.Geno.idx to awk format: {print $ ,...$ }
sed -i '1s/,$//;1s/^/{print /;1s/$/}/' new.Geno.idx

### Genotype data for SNPs with >=5 minor alleles
nohup awk -f new.Geno.idx < matched.Geno.data > new.Geno &
```

The output file is "new.Geno" where each row is a sample and each column is a SNP. The first column should be Sample ID. The corresponding map file is "new.Geno.map"

Summarize the minor allele frequency for SNPs in `new.Geno` with output to `new.Geno.maf`:

```
awk 'END {print NR}' matched.Gexp > n
nohup matlab -nodisplay -nodesktop -nosplash < ./codes/masummary.m > nohup.out
```

Output Files of This Section:

- `new.Geno`: genotype file;
- `new.Geno.idx`: index of SNPs selected to `new.Geno`;
- `new.Geno.map`: map file;

- `new.Geno.maf`: each element is the minor allele frequency (MAF) for each SNP.

6 Separating SNPs based on MAF

Input Files of This Section:

- `new.Geno.map`: map file;
- `new.Geno.maf`: each element is the minor allele frequency (MAF) for each SNP;
- `new.Geno`: genotype file.

We first separate SNPs based on minor allele frequency:

```
### Separate SNPs based on minor allele frequency:
###     Rare variants: MAF<0.01
###     Low frequency variants: MAF>=0.01 & MAF<0.05
###     Common variants: MAF>=0.05
Rscript ./codes/grpsnps.R
### Output: rare.snps.idx, low.snps.idx, common.snps.idx,
###         rare.snps.map, low.snps.map, common.snps.map

### Convert snps.idx to awk format: {print $1,$2,...$n}

### Gnotype data
cut -d' ' -f2- new.Geno > Geno.data #remove column header

### add '{print ' to the begining and '}' to the end of *.snps.idx file
sed -i '1s/,$//;1s/^/{print /;1s/$/}/' rare.snps.idx
sed -i '1s/,$//;1s/^/{print /;1s/$/}/' low.snps.idx
sed -i '1s/,$//;1s/^/{print /;1s/$/}/' common.snps.idx

### Genotype data for rare, low-freq, and common variants
### What about no rare, or low-freq variants???
nohup awk -f rare.snps.idx < Geno.data 1> rare.Geno.data 2>err_rare &
nohup awk -f low.snps.idx < Geno.data 1>low.Geno.data 2>err_low &
nohup awk -f common.snps.idx < Geno.data 1>common.Geno.data 2>err_common &

### SNP position (chr#, SNP pos)
awk '{print $1,$4}' rare.Geno.map > rare.SNPpos
awk '{print $1,$4}' low.Geno.map > low.SNPpos
awk '{print $1,$4}' common.Geno.map > common.SNPpos
```

The output files are `rare.Geno.data`, `low.Geno.data`, and `common.Geno.data`, the corresponding map files are `rare.Geno.map`, `low.Geno.map`, and `common.Geno.map`, and the corresponding SNP position files are `rare.SNPpos`, `low.SNPpos`, and `common.SNPpos`.

Output Files of This Section:

- 'n': the total number of observations;
- `rare.Geno.data`;
- `low.Geno.data`;
- `common.Geno.data`;
- `rare.Geno.map`: map file of `rare.Geno.data`;
- `low.Geno.map`: map file of `low.Geno.data`;
- `common.Geno.map`: map file of `common.Geno.data`;
- `all.SNPpos`: SNP position file of all genes;

- `rare.SNPpos`: SNP position file of `rare.Geno.data`;
- `low.SNPpos`: SNP position file of `low.Geno.data`;
- `common.SNPpos`: SNP position file of `common.Geno.data`.

7 Cis-eQTL Analysis

Input Files of This Section:

- `matched.Gexp`: gene exrepssion file matched to the genotype file;
- `GeneExpression/gene_pos`: gene annotation file;
- `genopos`: gene position file;
- `rare.SNPpos`:
- `low.SNPpos`:
- `common.SNPpos`:

Caution: We need the information about upstream & downstream information before we move forward to the rest of this section.

Now we can do cis-eQTL analysis with the gene expression data and genotype data.

```
cd /work/example/tspls

### Gene expression data: remove column header
cut -d' ' -f2- matched.Gexp > Gexp.data

# Remove the first column
### Gene position: Col#1=chr#; Col#2=start pos; Col#3=end pos
cut -f2- ./GeneExpression/gene_pos > genepos
```

Caution: By default, `cut` assumes the file is delimited with *tab*, otherwise, the switch `-d` should be used as shown to handle `matched.Gexp`.

The gene expression data is ***`Gexp.data`****, and the corresponding gene position file is* ***`genepos`****.*

In the below, R is used to find indices of rare, low-frequency, and common cis-SNPs for each gene (we need to decide upstream and downstream regions to define cis-SNPs):

```
# Find index of all cis-SNPs for each gene (including upstream & downstream regions)
Rscript ./codes/all.cispair.R

# Find index of common cis-SNPs for each gene (including upstream & downstream regions)
Rscript ./codes/common.cispair.R

# Find index of low-freq cis-SNPs for each gene (including upstream & downstream regions)
Rscript ./codes/low.cispair.R

# Find index of rare cis-SNPs for each gene (including upstream & downstream regions)
Rscript ./codes/rare.cispair.R
```

The above codes generate `rare.cispair.idx`, `low.cispair.idx`, and `common.cispair.idx`, including the index of genes and their corresponding rare, low-frequency, and common cis-SNPs.

Caution: The file `rare.cispair.idx` or even `low.cispair.idx` may be empty, so the corresponding script files should be modified.

We then first count the number of genes under consideration (saved into `ngenes` which will be used by `cissummary.R`), and then summarize the number of rare, low-frequency, and common cis-SNPs as below:

```
awk '{if(NF>max) max=NF} END {print max}' Gexp.data > ngenes
Rscript ./codes/cissummary.R
```

The above command will output the following files:

- `all.cispair.num`: total number of cis-SNPs for each gene
- `common.cispair.num`: number of common cis-SNPs for each gene
- `low.cispair.num`: number of low-frequency cis-SNPs for each gene
- `rare.cispair.num`: number of rare cis-SNPs for each gene

Caution: Some of the files may not be able to be generated due to the corresponding empty `.idx` files.

In the following, we will conduct cis-eQTL analysis, and evaluate the association between the expression value of each gene and the genotype of its corresponding rare, low-frequency, and common cis-SNPs.

7.1 Common Cis-eQTL Analysis

Firstly, we calculate the p-values of common cis-eQTL.

```
### Calcualte p-values of common cis-eQTL
Rscript ./codes/common.ciseQTL.R
```

It will generate the file `common.pValue`, which includes the *p*-value for the association between each gene and its common cis-SNPs, where Column 1 is Gene Index, Column is SNP Index, and Column 3 is *p*-Value.

Secondly, we selet significant cis-eQTL with *p*-values over,say 0.05.

```
### Select significant common cis-eQTL (p-value<0.05)
Rscript ./codes/sig.commoncis.R
```

The results will be output to `common.sig.pValue_0.05`, which includes the *p*-value of each pair of gene and its marginally significant (*p*-Value$<$0.05) common cis-eQTL, where Column 1 is Gene Index, Column is SNP Index, and Column 3 is *p*-Value.

We then obtain genotype data for these significant common cis-eQTL as below:

```
### Obtain genotype for eQTL data
Rscript ./codes/common.eQTLdata.R
```

The results will be output to the following two files:

- `common.eQTL.data`: includes the genotype data for marginally significant common cis-eQTL;
- `new.common.sig.pValue_0.05`: includes the p-value of each pair of gene and its marginally significant common cis-eQTL, where Column 1 is Gene Index, Column is SNP Index in `common.eQTL.data`, and Column 3 is p-Value.

7.2 Low-Frequency Cis-eQTL Analysis

We use permutations to calculate the p-values and run the permutations in parallel. The size of sliding window is 50. A template R file `codes/low.ciseQTL.R` will be used.

```
#!/bin/sh

echo "#!/bin/sh" > qsub.sh
chmod +x qsub.sh

for i in {1..16}
do
```

```
  perl -pe 's/YYstartYY/'$i'*1000-999/e; s/YYendYY/'$i'*1000/e; s/YYY/'$i'/g' < ./codes/low.ciseQTL.R >
  echo "nohup R CMD BATCH low.ciseQTL$i.R &" >> qsub.sh
done

sh qsub.sh
```

Caution: i runs from 1 to 16 so that `low.ciseQTL$i.R` goes from 1 to 1,600, which should be modified in the future to allow it go with the data!

There are two types of output files:

- `low.ciseQTL.weightXX`: the first column is the index of collapsed group of SNPs (a gene may have more than one collapsed groups), and the second column is the weight of each SNP in its collapsed group (either 1 or -1);
- `low.theoPXX`: the first column is the index of gene, the second column is the index of collapsed group of SNPs, and the third column is the p-value.

We combine the weight and p-value for all genes:

```
cat $(find ./ -name 'low.ciseQTL.weight*' | sort -V) > low.ciseQTL.weight0
# low.ciseQTL.weight: Col 1~gene index, Col 2~SNP index,
#     Col 3~index of collapsed group of SNPs, Col 4~weight of SNP (1 or -1)
paste low.cispair.idx low.ciseQTL.weight0 > low.ciseQTL.weight

# low.theoP: Col 1~gene index, Col 2~index of collapsed group of SNPs, Col 3~p-value
cat $(find ./ -name 'low.theoP*' | sort -V) > low.theoP
```

There are two types of output files:

- `low.ciseQTL.weight`: the first column is the gene index, the second column is the SNP index, the third column is the index of collapsed SNP group, and the fourth column is the weight of each SNP in its collapsed group (either 1 or -1);
- `low.theoP`: the first column is the gene index, the second column is the index of collapsed SNP group, and the third column is the p-value.

We then select significant collapsed low-frequency cis-eQTL with p-value less than, say, 0.05.

```
### Select significant collapsed low-freq cis-eQTL (p<0.05)
Rscript ./codes/sig.lowcis.R
```

The results are output into the following files:

- `low.sig.pValue_0.05`: includes the p-value of each pair of gene and its marginally significant (p-Value$<$0.05) collapsed low-frequency cis-eQTL, where Column 1 is Gene Index, Column is SNP Index, and Column 3 is p-Value;
- `low.sig.weight_0.05`: includes the weight of collapsed SNPs for marginally significant cis-eQTL. The first column is the gene index, the second column is the SNP index, the third column is the index of collapsed SNP group, and the fourth column is the weight of each SNP in its collapsed group (either 1 or -1).

We obtain genotype data for collapsed low-frequency cis-eQTL as below:

```
### Obtain genotype for collapsed cis-eQTL of each gene
Rscript ./codes/low.eQTLdata.R
```

The results will be saved into `low.eQTL.data`, which includes the genotype data for marginally significant collapsed low-frequency cis-eQTL.

7.3 Rare Cis-eQTL Analysis

We use permutations to calculate the p-values and run the permutations in parallel. The size of sliding window is 100. A template R file `codes/rare.ciseQTL.R` will be used.

```
#!/bin/sh

echo "#!/bin/sh" > qsub.sh
chmod +x qsub.sh

for i in {1..16}
do
  perl -pe 's/YYstartYY/'$i'*1000-999/e; s/YYendYY/'$i'*1000/e; s/YYY/'$i'/g' < ./codes/rare.ciseQTL.R
  echo "nohup R CMD BATCH rare.ciseQTL$i.R &" >> qsub.sh
done

sh qsub.sh
```

Caution: i runs from 1 to 16 so that `rare.ciseQTL$i.R` goes from 1 to 1,600, which should be modified in the future to allow it go with the data!

There are two types of output files:

- `rare.ciseQTL.weightXX`: its first column is the index of collapsed SNP group, and the second column is the weight of each SNP in its collapsed group (either 1 or -1);
- `rare.theoPXX`: its first column is the gene index, the second column is the index of collapsed SNP group, and the third column is the p-value.

We combine the weight and p-value for all genes:

```
cat $(find ./ -name 'rare.ciseQTL.weight*' | sort -V) > rare.ciseQTL.weight0
# rare.ciseQTL.weight: Col 1~gene index, Col 2~SNP index,
#     Col 3~index of collapsed group of SNPs, Col 4~weight of SNP (1 or -1)
paste rare.cispair.idx rare.ciseQTL.weight0 > rare.ciseQTL.weight

# rare.theoP: Col 1~gene index, Col 2~index of collapsed group of SNPs, Col 3~p-value
cat $(find ./ -name 'rare.theoP*' | sort -V) > rare.theoP
```

The results are output into two files:

- `rare.ciseQTL.weight`: its first column is the gene index, the second column is the SNP index, the third column is the index of collapsed SNP group (a gene may have more than one collapsed groups), and the fourth column is the weight of each SNP in its collapsed group (with value 1 or -1);
- `rare.theoP`: The first column of low.theoPXX is the gene index, the second column is the index of collapsed SNP group, and the third column is the p-value.

We then select significant collapsed rare cis-eQTL with p-values less than, say 0.05.

```
### Select significant collapsed rare cis-eQTL (p<0.05)
Rscript ./codes/sig.rarecis.R
```

The results are output to the following two files:

- `rare.sig.pValue_0.05`: includes the p-value of each pair of gene and its marginally significant (p-Value$<$0.05) collapsed rare cis-eQTL, where Column 1 is Gene Index, Column is SNP Index, and Column 3 is p-Value;
- `rare.sig.weight_0.05`: includes the weight of collapsed SNPs for marginally significant cis-eQTL. The first column is the gene index, the second column is the SNP index, the third column is the index

of collapsed SNP group, and the fourth column is the weight of each SNP in its collapsed group (with value 1 or -1).

We obtain genotype data for collapsed rare cis-eQTL as below:

```
### Obtain genotype for collapsed rare cis-eQTL of each gene
Rscript ./codes/rare.eQTLdata.R
```

The results are output into the file `rare.eQTL.data`, which includes the genotype data for marginally significant collapsed rare cis-eQTL.

7.4 Combined Common, Low-Frequency, and Rare eQTL Results

```
paste -d' ' common.eQTL.data low.eQTL.data rare.eQTL.data > all.eQTL.data
```

"all.eQTL.data" is the genotype data for all marginally significant cis-eQTL.

```
### index of combined eQTL
Rscript ./codes/comb.sigcis.R
```

The results are output in to the following files:

- `net.Gexp.data`: is the expression data for genes with cis-eQTL;
- `net.genepos`: include the position for genes in `net.Gexp.data`;
- `all.sig.pValue_0.05`: includes the p-value of each pair of gene and its marginally significant (p-Value$<$0.05) cis-eQTL, where Column 1 is Gene Index (in `net.Gexp.data`), Column is SNP Index (in `all.Geno.data`), and Column 3 is p-Value.

7.5 Selecting Uncorrelated Cis-eQTL (Up to 3) for Each Gene

After we obtain significant cis-eQTL for each gene, the set of cis-eQTL for each gene should be filtered to control the pairwise correlation under r (e.g., 0.5), and then further filtered to keep up to q (i.e., 3) cis-eQTL which have the strongest association with the corresponding gene expression. Below are the commands to filter the set of cis-eQTL for each gene:

```
### correlation between cis-eQTLs of the same gene will be controlled under r=0.5
Rscript ./codes/uncor.sigcis.R
```

The result is save into the file `uncoryx_idx`, which includes the index of genes and the corresponding filtered set of cis-eQTL.

The expression values of these genes and the genotypes of the corresponding SNPs will be used in the following network analysis.

Note: If we are working with human data, we may need to adjust for covariates including age, gender, BMI, and the first ten PCs, and use the residual expression values for the network analysis.

Output Files of This Section:

- `uncoryx_idx`: includes the index of genes and the corresponding filtered set of cis-eQTL.

8 Network Analysis

Input Files of This Section:

- `net.Gexp.data`: Blaaaa;

- `all.Geno.data`: Blaaaa;
- `uncoryx_idx`: includes the index of genes and the corresponding filtered set of cis-eQTL.

First we need to organize the gene expression data and genotype data for network analysis:

```
### Organize gene expression and genotype data for network analysis
Rscript ./codes/netyx.R
```

The output file `nety` includess the gene expression data for network analysis, `netx` includes the genotype data for network analysis, and `netyx_idx` specifies the index of each gene and its corresponding cis-eQTL.

We then obtain the information of gene symbols and positions for all genes included in the following network analysis.

```
### Get information (symbol & position) of genes in networks data
Rscript ./codes/netgene.info.R
```

The reults are output into two files:

- `nety_geneinfo`: include annotation information for all genes involved in the network analysis;
- `nety_genesymbol`: include gene symbols of all genes involved in the network analysis.

Now we are going to use 2SPLS to conduct network analysis on the data. Below is the R file for network analysis:

```
### Construct gene regulatory network using 2SPLS
Rscript ./codes/tspls.R
```

Output Files of This Section:

- `adjacency_matrix`: the adjancency matrix for the estimated regulatory effects;
- `coefficient_matrix`: the coefficient matrix for the estimated regulatory effects;
- `network.RData`: working environment of R?

9 Bootstrap Analysis

Input Files of This Section:

- `***`: Blaaaa;
- `***`: Blaaaa.

To evaluate the reliability of constructed gene regulations, we generate N (i.e., 100) bootstrap data sets by randomly sampling the original data with replacement, and apply 2SPLS to each data set to infer the gene regulatory network. We can employ *rice* to run bootstrap analysis in parallel.

Below are the R codes for sampling with replacement:

Caution: Need to specify the sample size of the original data, i.e., n!

```
### btsamples.R
### Generate bootstrap data sets with replacement
#
Nb=100 #number of bootstrap data sets
set.seed(123)
for (i in 1:Nb){
  idx=sample.int(n,size=n,replace=T) #n is the sample size
  idxname=paste('IDX',as.character(i),sep='')
  write.table(idx,idxname,quote=F,col.names=F,row.names=F)
}
```

For large data sets, we need to split Stage 1 and Stage 2 and run each piece of job in parallel. Different data sets may require different splitting strategy for parallel computing. The splitting strategy is based on the memory consumption and computing time for each gene. For example, assume we have 6,000 genes in total and we want to run 100 bootstrap data sets. The first stage for each gene consumes about 3GB and takes about 1 minute.

We know that *rice* has 580 nodes in total. Each node has 20 cores and 64GB memory. The maximum walltime is 4 hours. Since 3GB *times* 20 < 64GB, memory is not an issue, so we can submit 20 jobs to each node. To make sure a job can be finished in 4 hours, we can split the 6,000 genes into 40 pieces, and run the analysis for 150 genes in each job. There are 200 (6000$times$100/150/20) job scripts in total.

After the first stage is done, we can combine the output files (i.e., the predicted values) from the first stage and use them for the second stage. Similar splitting strageties can be applied to the second stage.

Below is an example of running bootstrap analysis using the splitting stragety:

9.1 Stage 1:

Stage 1 takes the original expression values of genes as an input file and generates the predicted values of genes as output files.

We here will use the following two templeta files:

- `bts1template.R`: *R* template file for the first stage of the bootstrap;
- `bts1template.sh`: *Bash* template file for the first stage of the bootstrap.

We generate job scripts and submit jobs using the following Linux commands:

```
sh bts1genbatch.sh
sh bts1gensub.sh
sh qsub1.sh
```

After the first stage is done, we can combine the output files (i.e., the predicted values) from the first stage and use them for the second stage. Below are the commands:

```
for i in {1..100}
do
  paste -d' ' $(find ./ -name 'ypre'$i'_*' | sort -V) > ypre$i
done
```

9.2 Stage 2

Stage 2 takes the predicted values from Stage 1 as an input file and generates the adjacency matrix and coefficient matrix as output files.

We here will use the following two templeta files:

- `bts2template.R`: *R* template file for the second stage of the bootstrap;
- `bts2template.sh`: *Bash* template file for the second stage of the bootstrap.

We generate job scripts and submit jobs using the following Linux commands:

```
sh bts2genbatch.sh
sh bts2gensub.sh
sh qsub2.sh
```

After the second stage is done, we can combine the output files (i.e., the adjacency matrix and the coefficient matrix) from the second stage. Below are the commands:

```
for i in {1..100}
do
  cat $(find ./ -name 'AdjMat'$i'_*' | sort -V) > AdjMat$i
  cat $(find ./ -name 'CoeffMat'$i'_*' | sort -V) > CoeffMat$i
done
```

Finally, we can summarize the results from bootstrap analysis:

```
### Calculate frequence of each edge
Rscript ./codes/Afreq.R
```

Output Files of This Section:

- `Afreq`: includes the adjacency matrix with each element being the bootstrap frequency;
- `edgelist_thre`: includes the gene regulations for which the bootstrap frequency is larger than a pre-specified threshold, where Column 1 is Source Gene, Column 2 is Target Gene, and Column 3 is Bootstrap Frequency.

10 Network Visualization

Input Files of This Section:

- `***`: Blaaaa;
- `***`: Blaaaa.

We can visualize the constructed network in *Cytoscape*. The input file is *edgelist_thre*. We need to specify source gene, target gene, and bootstrap frequency for Cytoscape.

11 Gene Functional Annotation

Input Files of This Section:

- `***`: Blaaaa;
- `***`: Blaaaa.

We can employ *DAVID* to conduct gene-enrichment analysis of the genes involved in the constructed network. We just need to take the gene symbols as input.

12 Appendix: Required Software Packages and Scripts

Caution: All the scripts included here are available in our sever `morgan:/work/example/tspls/codes/`.

It is assumed to run the data analysis on `Bash`, a Unix shell with following packages installed,

- `Bash`
  - Use `bash` to activate if inactive;
- `PLINK` (testd on v1.07)
  - The newer version PLINK (v1.9 & later) is much faster;
  - The newer version PLINK (v1.9 & later) is able to process *VCF* & *BCF* format files, as well as *Merlin* format files.
- R
- PERL
  - `extractrows.pl`: what functions are implemented?

  - splitchr.pl: what functions are implemented?
- MATLAB (for matrix transpose)

Caution: All code files should put inside the folder codes.

12.1 splitchr.pl

The following is the *PERL* scripts in splitchr.pl:

```
#!/usr/bin/perl
# Arguments: datafile sizesfile #chromosomes
#
# The sizesfile should look like should show a chromosome number followed
# by a number of columns, like:
# 1 440212
# 2 473669
# ...
# 22 82139

if (!($ARGV[0] =~ /.*\.data$/)) {
        die "Filename must end in .data\n";
        exit;
}

$basefn = $ARGV[0];
$basefn =~ s/\.data$//;

open(DATAFILE, $ARGV[0]) || die "Couldn't open $ARGV[0]\n";
open(SIZES, $ARGV[1]) || die "Couldn't open $ARGV[1]\n";
while(<SIZES>) {
        chomp;
        ($chr, $n) = split;
        $size[$chr] = $n;
}
close(SIZES);

$endcol[0] = 5;
foreach $i (1..$ARGV[2]) {
        $startcol[$i] = $endcol[$i-1] + 1;
        $endcol[$i] = $startcol[$i] + $size[$i] - 1;
        #print "$i, $startcol[$i], $endcol[$i]\n";
        $fname = $basefn . "_chr" . $i . ".data";
        open($fh[$i], ">$fname") || die "Cannot open $fname for writing\n";
}

while(<DATAFILE>) {
        chomp;
        @f = split;
        foreach $i (1..$ARGV[2]) {
                $fhandle = $fh[$i];
                print $fhandle join(" ", @f[$startcol[$i]..$endcol[$i]]), "\n";
        }
}
```

```
foreach $i (1..$ARGV[2]) {
        close($fh[$i]);
}
```

12.2 extractrows.pl

The following is the *PERL* scripts in `extractrows.pl`:

```
#!/usr/bin/perl
# extractrows.pl file
open(GENELIST, $ARGV[0]) || die "Couldn't open $ARGV[0]\n";
while(<GENELIST>) {
chomp;
$genes{$_}++;
}
close(GENELIST);

open(MAP, $ARGV[1]) || die "Couldn't open $ARGV[1]\n";
while(<MAP>) {
($thisgene, @dummy2)= split;
if ($genes{$thisgene}) {
print $_;
}
}
```

12.3 orderge.R

The following is the *R* scripts in `orderge.R`:

```
# orderge.R
### Order extracted gene expression data by extracted Geno Sample ID (R)
EGexp=read.table('EGexp');
EGexp=as.matrix(EGexp);
EGenoID=read.table('EGenoID');
EGenoID=as.matrix(EGenoID);
L=dim(EGenoID)[1];
idx=matrix(0,L,1);
for (i in 1:L){
  idx[i]=which(EGexp[,1]==EGenoID[i]);
}
MGexp=EGexp[idx,];
write.table(MGexp,'MGexp',quote=F,row.names=F,col.names=F)
```

12.4 genosum.m

The following is the *MatLab* scripts in `genosum.m`:

```
%%% genosum.m
%%% Calculate the minor allele counts for a chromosome
%
data=load('matched.Geno_chrXXX.data');
```

```
ma=sum(data,1);   %? ma=sum(data,2);
dlmwrite('matched.Geno.ma_chrXXX',ma,' ');
```

Note: This part of *MatLab* codes may be replaced by

```
nohup awk -F' ' '{for (i=1;i<=NF;i++) a[i] += $i} END{for (i=1;i<=NF;i++) print a[i]}' matched.Geno_chrX
```

12.5 filterma5.m

The following is the *MatLab* scripts in `filterma5.m`:

```
%%% filterma5.m
%%% Filter out SNPs with minor alleles less than 5
%
ma=load('matched.Geno.ma');
idx5=find(ma>=5);
dlmwrite('snps5.idx',idx5','delimiter',' ','precision','%.0f');
```

12.6 snps5.R

The following is the *R* scripts in `snps5.R`:

```
### snps5.R
### Generate a map file for SNPs with >=5 minor alleles
#
map=read.table("./Genotype/clean_Genotype.map")
map=as.matrix(map)
idx5=read.table("snps5.idx")
idx5=as.matrix(idx5)

map5=map[idx5,]
write.table(map5,"new.Geno.map",row.names=F,col.names=F,quote=F,sep=" ")

dataidx5=c(1,idx5+1)
write.table(paste0("$", dataidx5), "new.Geno.idx", row.names=F, col.names=F, quote=F, eol=",")
```

12.7 masummary.m

The following is the *MatLab* scripts in `masummary.m`:

```
%%% masummary.m
%%% Calculate minor allele frequency of each SNP
%
n = load('n'); %n is the sample size
ma=load('matched.Geno.ma');
idx5=find(ma>=5);
maf5=ma(idx5)/(n*2);

sum(maf5<0.01) %maf<0.01
sum(maf5>=0.01&maf5<0.05) %0.01=<maf<0.05
sum(maf5>=0.05) %maf>=0.05
```

```
dlmwrite('new.Geno.maf',maf5',' ');
```

Note: The transpose of a matrix may be implemented in *Python* as follows,

```
nohup python -c "import sys; print('\n'.join(' '.join(c) for c in zip(*(l.split() for l in sys.stdin.rea
```

12.8 grpsnps.R

The following is the *R* scripts in `grpsnps.R`:

```
### grpsnps.R
### Separate SNPs based on minor allele frequency:
###     Rare variants: MAF<0.01
###     Low frequency variants: MAF>=0.01 & MAF<0.05
###     Common variants: MAF>=0.05
map=read.table("new.Geno.map")
map=as.matrix(map)
maf=read.table("new.Geno.maf")
maf=as.matrix(maf)

rareidx=which(maf<0.01)
write.table(paste0("$", rareidx), "rare.snps.idx", row.names=F, col.names=F, quote=F, eol=",")
lowidx=which(maf>=0.01&maf<0.05)
write.table(paste0("$", lowidx), "low.snps.idx", row.names=F, col.names=F, quote=F, eol=",")
commonidx=which(maf>=0.05)
write.table(paste0("$", commonidx), "common.snps.idx", row.names=F, col.names=F, quote=F, eol=",")

raremap=map[rareidx,]
write.table(raremap,"rare.Geno.map",row.names=F,col.names=F,quote=F,sep=" ")
lowmap=map[lowidx,]
write.table(lowmap,"low.Geno.map",row.names=F,col.names=F,quote=F,sep=" ")
commonmap=map[commonidx,]
write.table(commonmap,"common.Geno.map",row.names=F,col.names=F,quote=F,sep=" ")
```

12.9 all.cispair.R

The following is the *R* scripts in `all.cispair.R`:

```
### all.cispair.R
### Find index of all cis-SNPs for each gene (including upstream & downstream regions)
###
# Load data
genepos=read.table('genepos'); # Col#1 is chr#; Col#2 is start pos; Col#3 is end pos;
SNPpos=read.table('all.SNPpos'); # Col#1 is chr#; Col#2 is SNP pos;
ly=dim(genepos)[1];

### Index of cis-SNP
cisSNP_idx=list();
for (i in 1:ly) { # the same chromosome, up 1Kb, down 1Kb
  cisSNP_idx[[i]]=which((SNPpos[,1]==genepos[i,1])&((genepos[i,2]-1000)<=SNPpos[,2])&
                        (SNPpos[,2]<=(genepos[i,3]+1000))) }

### Index of gene and its cis-SNP, Col#1 is index of gene, Col#2 is index of SNP
```

```
cispair_idx <- character(0)  # empty
for (i in 1:ly) {
  if (length(cisSNP_idx[[i]]>0))
    cispair_idx <- c(cispair_idx,paste(i,cisSNP_idx[[i]])) }

cispair_idx <- as.matrix(cispair_idx)
write.table(cispair_idx,"all.cispair.idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.10 common.cispair.R

The following is the *R* scripts in `common.cispair.R`:

```
### common.cispair.R
### Find index of common cis-SNPs for each gene (including upstream & downstream regions)
###
# Load data
genepos=read.table('genepos'); # Col#1 is chr#; Col#2 is start pos; Col#3 is end pos;
SNPpos=read.table('common.SNPpos'); # Col#1 is chr#; Col#2 is SNP pos;
ly=dim(genepos)[1];

### Index of cis-SNP
cisSNP_idx=list();
for (i in 1:ly) { # the same chromosome, up 1Kb, down 1Kb
  cisSNP_idx[[i]]=which((SNPpos[,1]==genepos[i,1])&((genepos[i,2]-1000)<=SNPpos[,2])&
                      (SNPpos[,2]<=(genepos[i,3]+1000))) }

### Index of gene and its cis-SNP, Col#1 is index of gene, Col#2 is index of SNP
cispair_idx <- character(0)  # empty
for (i in 1:ly) {
  if (length(cisSNP_idx[[i]]>0))
    cispair_idx <- c(cispair_idx,paste(i,cisSNP_idx[[i]])) }

cispair_idx <- as.matrix(cispair_idx)
write.table(cispair_idx,"common.cispair.idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.11 low.cispair.R

The following is the *R* scripts in `low.cispair.R`:

```
### low.cispair.R
### Find index of low-freq cis-SNPs for each gene (including upstream & downstream regions)
###
# Load data
genepos=read.table('genepos'); # Col#1 is chr#; Col#2 is start pos; Col#3 is end pos;
SNPpos=read.table('low.SNPpos'); # Col#1 is chr#; Col#2 is SNP pos;
ly=dim(genepos)[1];

### Index of cis-SNP
cisSNP_idx=list();
for (i in 1:ly) { # the same chromosome, up 1Kb, down 1Kb
  cisSNP_idx[[i]]=which((SNPpos[,1]==genepos[i,1])&((genepos[i,2]-1000)<=SNPpos[,2])&
                      (SNPpos[,2]<=(genepos[i,3]+1000))) }
```

```
### Index of gene and its cis-SNP, Col#1 is index of gene, Col#2 is index of SNP
cispair_idx <- character(0)  # empty
for (i in 1:ly) {
  if (length(cisSNP_idx[[i]]>0))
    cispair_idx <- c(cispair_idx,paste(i,cisSNP_idx[[i]])) }

cispair_idx <- as.matrix(cispair_idx)
write.table(cispair_idx,"low.cispair.idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.12 rare.cispair.R

The following is the *R* scripts in rare.cispair.R:

```
### rare.cispair.R
### Find index of rare cis-SNPs for each gene (including upstream & downstream regions)
###
# Load data
genepos=read.table('genepos'); # Col#1 is chr#; Col#2 is start pos; Col#3 is end pos;
SNPpos=read.table('rare.SNPpos'); # Col#1 is chr#; Col#2 is SNP pos;
ly=dim(genepos)[1];

### Index of cis-SNP
cisSNP_idx=list();
for (i in 1:ly) { # the same chromosome, up 1Kb, down 1Kb
  cisSNP_idx[[i]]=which((SNPpos[,1]==genepos[i,1])&((genepos[i,2]-1000)<=SNPpos[,2])&
                      (SNPpos[,2]<=(genepos[i,3]+1000))) }

### Index of gene and its cis-SNP, Col#1 is index of gene, Col#2 is index of SNP
cispair_idx <- character(0)  # empty
for (i in 1:ly) {
  if (length(cisSNP_idx[[i]]>0))
    cispair_idx <- c(cispair_idx,paste(i,cisSNP_idx[[i]])) }

cispair_idx <- as.matrix(cispair_idx)
write.table(cispair_idx,"rare.cispair.idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.13 cissummary.R

The following is the *R* scripts in cissummary.R:

```
### cissummary.R
### Number of cis-SNPs for each gene
#
idx=read.table("all.cispair.idx")
len=scan('ngenes',quiet=TRUE) #number of genes
cisnum=matrix(0,len,1)
for (i in 1:len){
  cisnum[i]=sum(idx[,1]==i)
  if((i%%1000)==0) print(i)
}
write.table(cisnum,"all.cispair.num",row.names=F,col.names=F,quote=F,sep=" ")
```

```
idx=read.table("common.cispair.idx")
cisnum=matrix(0,len,1)
for (i in 1:len){
  cisnum[i]=sum(idx[,1]==i)
  if((i%%1000)==0) print(i)
}
write.table(cisnum,"common.cispair.num",row.names=F,col.names=F,quote=F,sep=" ")

idx=read.table("low.cispair.idx")
cisnum=matrix(0,len,1)
for (i in 1:len){
  cisnum[i]=sum(idx[,1]==i)
  if((i%%1000)==0) print(i)
}
write.table(cisnum,"low.cispair.num",row.names=F,col.names=F,quote=F,sep=" ")

idx=read.table("rare.cispair.idx")
cisnum=matrix(0,len,1)
for (i in 1:len){
  cisnum[i]=sum(idx[,1]==i)
  if((i%%1000)==0) print(i)
}
write.table(cisnum,"rare.cispair.num",row.names=F,col.names=F,quote=F,sep=" ")
```

12.14 SumTest.R

The following is the *R* scripts in `SumTest.R`:

```
### SumTest.R
#
SumTest <- function(Y,X,alpha0){
    pv <- NULL
    beta <- NULL
    for(i in 1:ncol(X)){
       fit  <- lm(Y~X[,i])
       beta <- c(beta,fit$coefficients[-1])
       pv <- c(pv,as.numeric(summary(fit)$coefficients[,4][-1]))
    }

    Xg <- X
    Xgb <- Xg-mean(Xg)
    n <- nrow(Xgb)

    #U <- t(Xg) %*% (Y-mean(Y)) #score vector, needs to be modified for normal distribution
    #CovS <- mean(Y)*(1-mean(Y))*(t(Xgb) %*% Xgb)  #variance of score vector
    U <- (t(Xg) %*% (Y-mean(Y))) * (n-1) / as.numeric(t(Y-mean(Y)) %*% (Y-mean(Y))) #score vector
    CovS <- (t(Xgb) %*% Xgb) * (n-1) / as.numeric(t(Y-mean(Y)) %*% (Y-mean(Y))) #variance of score vect

    w <- rep(1, length(U))
    w[beta<0 & pv<alpha0] <- -1

    u <- sum(t(w)%*%U) #adaptive score vector
    v <- as.numeric(t(w) %*% CovS %*% (w)) #variance of adaptive score vector
    Tsum <- u/sqrt(v) #test statistic
    pTsum <- as.numeric( 1-pchisq(Tsum^2, 1) ) #p-value

    return(cbind(u,v,Tsum,pTsum,w))
}
```

12.15 rSumTest.R

The following is the *R* scripts in `rSumTest.R`:

```
### rSumTest.R
#
rSumTest <- function(Y,X,k,alpha0){
    Yg <- Y
    n <- length(Yg)

    u0 <- v0 <- pv0 <- NULL
    for(sim in 1:k){
       set.seed(sim)
       pidx <- sample.int(n,size=n,replace=FALSE)
       Y <- Yg[pidx]

       fit0 <- SumTest(Y,X,alpha0)
       u0 <- c(u0, fit0[1,1])
       v0 <- c(v0, fit0[1,2])
       pv0 <- c(pv0,as.numeric(fit0[1,4]))
    }

    x <- (u0-mean(u0))^2/var(u0)
    a <- sqrt(var(x)/2)
    b <- mean(x)-a

    return(cbind(rep(mean(u0),k), rep(var(u0),k) ,rep(a,k), rep(b,k), pv0))
}
```

12.16 common.ciseQTL.R

The following is the *R* scripts in `common.ciseQTL.R`:

```
### common.ciseQTL.R
### Calculate p-values of all common cis-eQTL
#
y=read.table('Gexp.data')
x=read.table('common.Geno.data')
y=as.matrix(y)
x=as.matrix(x)
y=scale(y)
x=scale(x)
idx=read.table('common.cispair.idx')
len=dim(idx)[1]

p=matrix(0,len,1)
```

```r
for (i in 1:len){
  if(any(is.nan(x[,idx[i,2]])))
    p[i] <- 1
  else {
    fit=lm(y[,idx[i,1]]~x[,idx[i,2]])
    p[i]=summary(fit)$coefficients[2,4]
  }
  if((i%%100)==0) print(i)
}

p=cbind(idx,p) # Col 1 is index of gene, Col 2 is index of its cis-SNP, Col 3 is p-value
write.table(p,"common.pValue",row.names=F,col.names=F,quote=F,sep=" ")
```

12.17 sig.commoncis.R

The following is the *R* scripts in `sig.commoncis.R`:

```r
### sig.commoncis.R
### Select significant common cis-eQTL (p-value<0.05)
#
pval=read.table("common.pValue")
names(pval)=c("y","x","p")
sigp=pval[pval$p<0.05,] # significant cis-eQTL
lyx=nrow(sigp) # number of gene&cis-eQTL pairs
y=unique(sigp$y)
ly=length(y) # number of genes that have cis-eQTL
x=unique(sigp$x)
lx=length(x) # number of SNPs that are cis-eQTL of certain genes

tablex=as.data.frame(table(sigp$x))
uniqx=tablex[tablex$Freq==1,1]
lux=length(uniqx) #number of SNPs that are cis-eQTL of only one gene

ind=matrix(0,ly,1)
for (i in 1:ly){
  ind[i]=sum(sigp$x[sigp$y==unique(sigp$y)[i]] %in% uniqx)
}
uniqy=y[ind!=0]
luy=length(uniqy) # number of genes that have at least one unique cis-eQTL

### p-value for genes that have at least one unique cis-eQTL
uniqsigp=sigp[sigp$y%in%uniqy,]
write.table(uniqsigp,"common.sig.pValue_0.05",row.names=F,col.names=F,quote=F,sep=" ")
```

12.18 common.eQTLdata.R

The following is the *R* scripts in `common.eQTLdata.R`:

```r
### common.eQTLdata.R
### Obtain genotype for eQTL data
#
data=read.table("common.Geno.data")
```

```r
data=as.matrix(data)
sigp=read.table("common.sig.pValue_0.05")
names(sigp)=c("y","x","p")

uniqx=unique(sigp$x)

eQTL=data[,uniqx]
write.table(eQTL,"common.eQTL.data",row.names=F,col.names=F,quote=F,sep=" ")

lx=nrow(sigp)
newx=matrix(0,lx,1)
for (i in 1:lx){
  newx[i]=which(uniqx==sigp$x[i])
}
sigp$x=newx
write.table(sigp,"new.common.sig.pValue_0.05",row.names=F,col.names=F,quote=F,sep=" ")
```

12.19 `low.ciseQTL.R`

The following is the *R* scripts in `low.ciseQTL.R`:

```r
### low.ciseQTL.R
### Calculate p-values of low-freq cis-eQTL via permutation
#
y=read.table('Gexp.data')
x=read.table('low.Geno.data')
idx=read.table('low.cispair.idx')
y=as.matrix(y)
x=as.matrix(x)
idx=as.matrix(idx)
yidx=unique(idx[,1])
ylen=length(yidx)

alpha0=0.1 #the cut-off value;
           #if the marginal p-value of an SNP is less than alpha0 and
           #its marginal regression coefficient is negative,
           #we flip its coding;
B=100 #the number of permutations

source("./codes/SumTest.R")
source("./codes/rSumTest.R")

w <- NULL
aSumP <- NULL
theoP <- NULL
for (i in YYstartYY:YYendYY){
    Y <- y[,yidx[i]]
    X <- x[,idx[idx[,1]==yidx[i],2]]
    Y <- as.matrix(Y)
    X <- as.matrix(X)

    ng <- ncol(X)
    wsize <- 50
    nw <- ng %/% wsize

    if (nw==0){
        Xs <- X[,1:ng]
        Xs <- as.matrix(Xs)
        fit <- SumTest(Y,Xs,alpha0)
        u <- fit[1,1]
        v <- fit[1,2]
        pv <- fit[1,4] #p-value
        w < rbind(w, cbind(rep(1,ng), fit[,5])) #weight of X

        fit0 <- rSumTest(Y,X,B,alpha0)
        u0 <- fit0[1,1]
        v0 <- fit0[1,2]
        a <- fit0[1,3]
        b <- fit0[1,4]
        pv0 <- fit0[,5]

        aSumP <- rbind(aSumP, c(yidx[i], 1, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
        theoP <- rbind(theoP, c(yidx[i], 1, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
    } else{
        for (j in 1:nw){
            Xs <- X[,((j-1)*wsize+1):(j*wsize)]
            Xs <- as.matrix(Xs)
            fit <- SumTest(Y,Xs,alpha0)
            u <- fit[1,1]
            v <- fit[1,2]
            pv <- fit[1,4] #p-value
            w < rbind(w, cbind(rep(j,wsize), fit[,5])) #weight of X

            fit0 <- rSumTest(Y,X,B,alpha0)
            u0 <- fit0[1,1]
            v0 <- fit0[1,2]
            a <- fit0[1,3]
            b <- fit0[1,4]
            pv0 <- fit0[,5]

            aSumP <- rbind(aSumP, c(yidx[i], j, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
            theoP <- rbind(theoP, c(yidx[i], j, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
        }

        if (ng%%wsize!=0){
            j <- nw+1
            Xs <- X[,(nw*wsize+1):ng]
            Xs <- as.matrix(Xs)
            fit <- SumTest(Y,Xs,alpha0)
            u <- fit[1,1]
            v <- fit[1,2]
            pv <- fit[1,4] #p-value
            w < rbind(w, cbind(rep(j,ng%%wsize), fit[,5])) #weight of X

            fit0 <- rSumTest(Y,X,B,alpha0)
            u0 <- fit0[1,1]
            v0 <- fit0[1,2]
            a <- fit0[1,3]
            b <- fit0[1,4]
            pv0 <- fit0[,5]

            aSumP <- rbind(aSumP, c(yidx[i], j, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
            theoP <- rbind(theoP, c(yidx[i], j, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
        }
    }

    if((i%%100)==0) print(i)
}

write.table(w,"low.ciseQTL.weightYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of collapsed SNPs for a gene, Col 2: weight of collapsed SNPs
write.table(aSumP,"low.aSumPYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of gene, Col 2: index of collapsed SNPs for a gene, Col 3: p-value
write.table(theoP,"low.theoPYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of gene, Col 2: index of collapsed SNPs for a gene, Col 3: p-value
```

12.20 sig.lowcis.R

The following is the *R* scripts in sig.lowcis.R:

```
### sig.lowcis.R
### Select significant collapsed low-freq cis-eQTL (p<0.05)
#
pval=read.table("low.theoP")
names(pval)=c("y","cx","p")
sigp=pval[pval$p<0.05,] # significant cis-eQTL
lyx=nrow(sigp) #number of gene&collapsed cis-eQTL pairs
y=unique(sigp$y)
ly=length(y) #number of genes that have cis-eQTL

### check whether genes have unique collapsed cis-eQTL
weight <- read.table("low.ciseQTL.weight")
names(weight) <- c("y","x","cx","w")
sigw <- weight[paste(weight$y,weight$cx)%in%paste(sigp$y,sigp$cx),]
sex=matrix(0,ly,3) #start and end SNP of collapsed cis-eQTL for a gene
sex[,1]=y
for (i in 1:ly){
  sex[i,2]=sigw$x[which(sigw$y==y[i])[1]]
  sex[i,3]=sigw$x[which(sigw$y==y[i])[sum(sigw$y==y[i])]]
}
lx=nrow(unique(sex[,2:3]))
# make sure each gene has a unique collapsed cis-eQTL!

### save p-values and weights for significant collapsed cis-eQTL
sigp$cx <- 1:lyx
write.table(sigp,"low.sig.pValue_0.05",row.names=F,col.names=F,quote=F,sep=" ")
write.table(sigw,"low.sig.weight_0.05",row.names=F,col.names=F,quote=F,sep=" ")
```

12.21 low.eQTLdata.R

The following is the *R* scripts in `low.eQTLdata.R`:

```
### low.eQTLdata.R
### Obtain genotype for collapsed cis-eQTL of each gene
#
x <- read.table("low.Geno.data")
x <- as.matrix(x)
sigw <- read.table("low.sig.weight_0.05")
names(sigw) <- c("y","x","cx","w")

n <- nrow(x)
uniqyx <- unique(paste(sigw$y,sigw$cx))
lyx <- length(uniqyx)
eQTL <- matrix(0,n,lyx)

for (i in 1:lyx){
  widx <- which(paste(sigw$y, sigw$cx)==uniqyx[i])
  eQTL[,i] <- as.matrix(x[,sigw$x[widx]]) %*% sigw$w[widx]
}

write.table(eQTL,"low.eQTL.data",row.names=F,col.names=F,quote=F,sep=" ")
```

12.22 rare.ciseQTL.R

The following is the *R* scripts in `rare.ciseQTL.R`:

```
### rare.ciseQTL.R
### Calculate p-values of rare cis-eQTL via permutation
#
y=read.table('Gexp.data')
x=read.table('rare.Geno.data')
idx=read.table('rare.cispair.idx')
y=as.matrix(y)
x=as.matrix(x)
idx=as.matrix(idx)
yidx=unique(idx[,1])
ylen=length(yidx)

alpha0=0.1 #the cut-off value;
           #if the marginal p-value of an SNP is less than alpha0 and
           #its marginal regression coefficient is negative,
           #we flip its coding;
B=100 #the number of permutations

source("./codes/SumTest.r")
source("./codes/rSumTest.r")

w <- NULL
aSumP <- NULL
theoP <- NULL
for (i in YYstartYY:YYendYY){
    Y <- y[,yidx[i]]
    X <- x[,idx[idx[,1]==yidx[i],2]]
    Y <- as.matrix(Y)
    X <- as.matrix(X)

    ng <- ncol(X)
    wsize <- 100
    nw <- ng %/% wsize

    if (nw==0){
        Xs <- X[,1:ng]
        Xs <- as.matrix(Xs)
        fit <- SumTest(Y,Xs,alpha0)
        u <- fit[1,1]
        v <- fit[1,2]
        pv <- fit[1,4] #p-value
        w <- rbind(w, cbind(rep(1,ng), fit[,5])) #weight of X

        fit0 <- rSumTest(Y,X,B,alpha0)
        u0 <- fit0[1,1]
        v0 <- fit0[1,2]
        a <- fit0[1,3]
        b <- fit0[1,4]
        pv0 <- fit0[,5]

        aSumP <- rbind(aSumP, c(yidx[i], 1, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
        theoP <- rbind(theoP, c(yidx[i], 1, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
    } else{
        for (j in 1:nw){
            Xs <- X[,((j-1)*wsize+1):(j*wsize)]
            Xs <- as.matrix(Xs)
            fit <- SumTest(Y,Xs,alpha0)
            u <- fit[1,1]
            v <- fit[1,2]
            pv <- fit[1,4] #p-value
            w <- rbind(w, cbind(rep(j,wsize), fit[,5])) #weight of X

            fit0 <- rSumTest(Y,X,B,alpha0)
            u0 <- fit0[1,1]
            v0 <- fit0[1,2]
            a <- fit0[1,3]
            b <- fit0[1,4]
            pv0 <- fit0[,5]

            aSumP <- rbind(aSumP, c(yidx[i], j, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
            theoP <- rbind(theoP, c(yidx[i], j, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
        }

        if (ng%%wsize!=0){
            j <- nw+1
            Xs <- X[,(nw*wsize+1):ng]
            Xs <- as.matrix(Xs)
            fit <- SumTest(Y,Xs,alpha0)
            u <- fit[1,1]
            v <- fit[1,2]
            pv <- fit[1,4] #p-value
            w <- rbind(w, cbind(rep(j,ng%%wsize), fit[,5])) #weight of X

            fit0 <- rSumTest(Y,X,B,alpha0)
            u0 <- fit0[1,1]
            v0 <- fit0[1,2]
            a <- fit0[1,3]
            b <- fit0[1,4]
            pv0 <- fit0[,5]

            aSumP <- rbind(aSumP, c(yidx[i], j, sum(pv>pv0)/length(pv0)) ) #permutation-based p value
            theoP <- rbind(theoP, c(yidx[i], j, as.numeric(1 - pchisq(abs(((u-u0)^2/v0-b)/a),1)) ) )
        }
    }

  if((i%%100)==0)  print(i)
}

write.table(w,"rare.ciseQTL.weightYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of collapsed SNPs for a gene, Col 2: weight of collapsed SNPs
write.table(aSumP,"rare.aSumPYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of gene, Col 2: index of collapsed SNPs for a gene, Col 3: p-value
write.table(theoP,"rare.theoPYYY",row.names=F,col.names=F,quote=F,sep=" ")
#Col 1: index of gene, Col 2: index of collapsed SNPs for a gene, Col 3: p-value
```

12.23 sig.rarecis.R

The following is the *R* scripts in `sig.rarecis.R`:

```
### sig.rarecis.R
### Select significant collapsed rare cis-eQTL (p<0.05)
#
pval=read.table("rare.theoP")
names(pval)=c("y","cx","p")
sigp=pval[pval$p<0.05,] # significant cis-eQTL
lyx=nrow(sigp) #number of gene&collapsed cis-eQTL pairs
y=unique(sigp$y)
ly=length(y) #number of genes that have cis-eQTL

### check whether all genes have unique collapsed cis-eQTL
weight <- read.table("rare.ciseQTL.weight")
names(weight) <- c("y","x","cx","w")
sigw <- weight[paste(weight$y,weight$cx)%in%paste(sigp$y,sigp$cx),]
sex=matrix(0,ly,3) #start and end SNP of collapsed cis-eQTL for a gene
sex[,1]=y
for (i in 1:ly){
  sex[i,2]=sigw$x[which(sigw$y==y[i])[1]]
  sex[i,3]=sigw$x[which(sigw$y==y[i])[sum(sigw$y==y[i])]]
}
lx=nrow(unique(sex[,2:3]))
# make sure all genes have at least one unique collapsed cis-eQTL
```

```
### save p-values and weights for significant collapsed cis-eQTL
sigp$cx <- 1:lyx
write.table(sigp,"rare.sig.pValue_0.05",row.names=F,col.names=F,quote=F,sep=" ")
write.table(sigw,"rare.sig.weight_0.05",row.names=F,col.names=F,quote=F,sep=" ")
```

12.24 rare.eQTLdata.R

The following is the *R* scripts in `rare.eQTLdata.R`:

```
### rare.eQTLdata.R
### Obtain genotype for collapsed rare cis-eQTL of each gene
#
x <- read.table("rare.Geno.data")
x <- as.matrix(x)
sigw <- read.table("rare.sig.weight_0.05")
names(sigw) <- c("y","x","cx","w")

n <- nrow(x)
uniqyx <- unique(paste(sigw$y,sigw$cx))
lyx <- length(uniqyx)
eQTL <- matrix(0,n,lyx)

for (i in 1:lyx){
  widx <- which(paste(sigw$y, sigw$cx)==uniqyx[i])
  eQTL[,i] <- as.matrix(x[,sigw$x[widx]]) %*% sigw$w[widx]
}

write.table(eQTL,"rare.eQTL.data",row.names=F,col.names=F,quote=F,sep=" ")
```

12.25 comb.sigcis.R

The following is the *R* scripts in `comb.sigcis.R`:

```
### comb.sigcis.R
### index of combined eQTL
#
common=read.table("new.common.sig.pValue_0.05")
low=read.table("low.sig.pValue_0.05")
rare=read.table("rare.sig.pValue_0.05")

# number of common cis-eQTL:
n1<-as.numeric(system("head -1 common.eQTL.data | tr ' ' '\n' | wc -l",intern=T))
# number of collapsed low-freq cis-eQTL:
n2<-as.numeric(system("head -1 low.eQTL.data | tr ' ' '\n' | wc -l",intern=T))
# number of collapsed rare cis-eQTL:
n3<-as.numeric(system("head -1 rare.eQTL.data | tr ' ' '\n' | wc -l",intern=T))
low[,2]=(n1+1):(n1+n2)
rare[,2]=(n1+n2+1):(n1+n2+n3)
all=rbind(common,low,rare)

### obtain expression data for genes with eQTL
data=read.table("Gexp.data")
```

```
data=as.matrix(data)

uniqy=unique(all[,1])

netdata=data[,uniqy]
write.table(netdata,"net.Gexp.data",row.names=F,col.names=F,quote=F,sep=" ")

### gene name and gene position
pos=read.table("genepos")
pos=as.matrix(pos)

uniqy=unique(all[,1])

netname=name[uniqy,]
netpos=pos[uniqy,]
write.table(netpos,"net.genepos",row.names=F,col.names=F,quote=F,sep=" ")

### index of gene in net.Gexp.data
ly=nrow(all)
newy=matrix(0,ly,1)
for (i in 1:ly){
  newy[i]=which(uniqy==all[i,1])
}
all[,1]=newy

### new index of gene and eQTL
length(unique(all[,1])) #number of genes
length(unique(all[,2])) #number of SNPs
nrow(all) #number of gene-eQTL pairs
write.table(all,"all.sig.pValue_0.05",row.names=F,col.names=F,quote=F,sep=" ")
```

12.26 uncor.sigcis.R

The following is the *R* scripts in uncor.sigcis.R:

```
### uncor.sigcis.R
### correlation between cis-eQTLs of the same gene will be controlled under r=0.5
#
r=0.5;

y <- read.table("net.Gexp.data"); #gene expression data
x=read.table("all.eQTL.data"); #genotype data
res=read.table("all.sig.pValue_0.05"); #Col 1: gene index, Col 2:cis-eQTL index, Col 3: p-value
y=as.matrix(y);
x=as.matrix(x);
res=as.matrix(res);

y_idx=res[,1]; #index of gene
uniqy_idx=unique(y_idx); #index of unique gene
ly=length(uniqy_idx);
x_idx=list();
for (i in 1:ly){
  x_idx[[i]]=res[which(res[,1]==uniqy_idx[i]),2] #index of cis-eQTL for each gene
```

```
}
pval=list();
for (i in 1:ly){
  pval[[i]]=res[which(res[,1]==uniqy_idx[i]),3] #p value of cis-eQTL for each gene
}

uncorx_idx0=list(); #index of selected cis-eQTL (up to 3) for each gene
#idx1: index of the most significant cis-eQTL for each gene in x_idx (in 1st run)
#idx2: index of the most significant cis-eQTL for each gene in x_idx1 (in 2nd run)
#idx3: index of the most significant cis-eQTL for each gene in x_idx2 (in 3rd run)
#uncorx_idx1: index of the most significant cis-eQTL for each gene (in 1st run)
#uncorx_idx2: index of the most significant cis-eQTL for each gene (in 2nd run)
#uncorx_idx3: index of the most significant cis-eQTL for each gene (in 3rd run)
x_cor1=list(); #correlation between cis-eQTL for each gene (after 1st run)
x_cor2=list(); #correlation between cis-eQTL for each gene (after 2nd run)
x_idx1=list(); #index of cis-eQTL for each gene (after 1st run)
x_idx2=list(); #index of cis-eQTL for each gene (after 2nd run)
pval1=list(); #p value of cis-eQTL (after 1st run)
pval2=list(); #p value of cis-eQTL (after 2nd run)

########################################
### index of selected cis-eQTL (up to 3) for each gene

uncorx_idx=NULL; #list of all selected cis-eQTL

i=1

if (length(x_idx[[i]])==1){
  uncorx_idx0[[i]]=x_idx[[i]]; #index of the first selected cis-eQTL
} else{
  #index of the most significant cis-eQTL in x_idx (in 1st run):
  idx1=which(pval[[i]]==min(pval[[i]]))[1];
  uncorx_idx1=x_idx[[i]][idx1]; #index of the most significant cis-eQTL (in 1st run)
  x_cor1[[i]]=cor(x[,x_idx[[i]]]); #correlation between cis-eQTL (after 1st run)
  x_idx1[[i]]=x_idx[[i]][abs(x_cor1[[i]][idx1,])<r]; #index of cis-eQTL (after 1st run)
  pval1[[i]]=pval[[i]][abs(x_cor1[[i]][idx1,])<r]; #result of cis-eQTL (after 1st run)

  #if gene has no cis-eQTL after first run
  if (length(x_idx1[[i]])==0){
    uncorx_idx0[[i]]=uncorx_idx1; #index of the first selected cis-eQTL
  } else if (length(x_idx1[[i]])==1){ #if gene has one cis-eQTL after first run
    uncorx_idx0[[i]]=c(uncorx_idx1,x_idx1[[i]]); #index of the first&second selected cis-eQTL
  } else{
    #index of the most significant cis-eQTL in x_idx1 (in 2nd run):
    idx2=which(pval1[[i]]==min(pval1[[i]]))[1];
    uncorx_idx2=x_idx1[[i]][idx2]; #index of the most significant cis-eQTL (in 2nd run)
    x_cor2[[i]]=cor(x[,x_idx1[[i]]]); #correlation between cis-eQTL (after 2nd run)
    x_idx2[[i]]=x_idx1[[i]][abs(x_cor2[[i]][idx2,])<r]; #index of cis-eQTL (after 2nd run)
    pval2[[i]]=pval1[[i]][abs(x_cor2[[i]][idx2,])<r]; #result of cis-eQTL (after 2nd run)

    #if gene has no cis-eQTL after 2nd run
    if (length(x_idx2[[i]])==0){
      uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2); #index of the first&second selected cis-eQTL
    } else if (length(x_idx2[[i]])==1){
      uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2,x_idx2[[i]]); #index of all three selected cis-eQTL
    } else{
      #index of the most significant cis-eQTL in x_idx2 (in 3rd run):
      idx3=which(pval2[[i]]==min(pval2[[i]]))[1];
      uncorx_idx3=x_idx2[[i]][idx3]; #index of the most significant cis-eQTL (in 3rd run)
      uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2,uncorx_idx3); #index of all three selected cis-eQTL
    }
  }
}
uncorx_idx=c(uncorx_idx,uncorx_idx0[[i]]) #list of all selected cis-eQTL
print(i)


for (i in 2:ly){
  corr=apply(as.matrix(x[,x_idx[[i]]]),2,function(xx) cor(xx,x[,uncorx_idx]))
  xind=(colSums(corr>=r)==0)
  #index of cis-eQTL for i-th gene which are uncorrelated (corr<r) with previously selected cis-eQTL:
  x_idx[[i]]=x_idx[[i]][xind==1]
  #result of cis-eQTL for i-th gene which are uncorrelated (corr<r) with previously selected cis-eQTL:
  pval[[i]]=pval[[i]][xind==1]

  if (length(x_idx[[i]])<=1){
    uncorx_idx0[[i]]=x_idx[[i]]; #index of the first selected cis-eQTL
  } else{
    #index of the most significant cis-eQTL in x_idx (in 1st run):
    idx1=which(pval[[i]]==min(pval[[i]]))[1];
    uncorx_idx1=x_idx[[i]][idx1]; #index of the most significant cis-eQTL (in 1st run)
    x_cor1[[i]]=cor(x[,x_idx[[i]]]); #correlation between cis-eQTL (after 1st run)
    x_idx1[[i]]=x_idx[[i]][abs(x_cor1[[i]][idx1,])<r]; #index of cis-eQTL (after 1st run)
    pval1[[i]]=pval[[i]][abs(x_cor1[[i]][idx1,])<r]; #result of cis-eQTL (after 1st run)

    #if gene has no cis-eQTL after first run
    if (length(x_idx1[[i]])==0){
      uncorx_idx0[[i]]=uncorx_idx1; #index of the first selected cis-eQTL
    } else if (length(x_idx1[[i]])==1){ #if gene has one cis-eQTL after first run
      uncorx_idx0[[i]]=c(uncorx_idx1,x_idx1[[i]]); #index of the first&second selected cis-eQTL
    } else{
      #index of the most significant cis-eQTL in x_idx1 (in 2nd run):
      idx2=which(pval1[[i]]==min(pval1[[i]]))[1];
      uncorx_idx2=x_idx1[[i]][idx2]; #index of the most significant cis-eQTL (in 2nd run)
      x_cor2[[i]]=cor(x[,x_idx1[[i]]]); #correlation between cis-eQTL (after 2nd run)
      x_idx2[[i]]=x_idx1[[i]][abs(x_cor2[[i]][idx2,])<r]; #index of cis-eQTL (after 2nd run)
      pval2[[i]]=pval1[[i]][abs(x_cor2[[i]][idx2,])<r]; #result of cis-eQTL (after 2nd run)

      #if gene has no cis-eQTL after 2nd run
      if (length(x_idx2[[i]])==0){
        uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2); #index of the first&second selected cis-eQTL
      } else if (length(x_idx2[[i]])==1){
        uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2,x_idx2[[i]]); #index of all three selected cis-eQTL
      } else{
        #index of the most significant cis-eQTL in x_idx2 (in 3rd run):
        idx3=which(pval2[[i]]==min(pval2[[i]]))[1];
        uncorx_idx3=x_idx2[[i]][idx3]; #index of the most significant cis-eQTL (in 3rd run)
        uncorx_idx0[[i]]=c(uncorx_idx1,uncorx_idx2,uncorx_idx3); #index of all three selected cis-eQTL
      }
    }
  }
  uncorx_idx=c(uncorx_idx,uncorx_idx0[[i]]) #list of all selected cis-eQTL
  if((i%%1000)==0) print(i)
}

uncory_idx=list(); #index of gene with selected cis-eQTL
for (i in 1:ly){
  uncory_idx[[i]]=rep(uniqy_idx[i],length(uncorx_idx0[[i]]))
}

uncoryx_idx=paste(unlist(uncory_idx),uncorx_idx) #index of gene and its selected cis-eQTL
uncoryx_idx=as.matrix(uncoryx_idx)
write.table(uncoryx_idx,"uncoryx_idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.27 netyx.R

The following is the *R* scripts in `netyx.R`:

```
### netyx.R
### Organize gene expression and genotype data for network analysis
#
y=read.table("net.Gexp.data"); #gene expression data
x=read.table("all.Geno.data"); #genotype data

uncoryx_idx=read.table("uncoryx_idx")
uncoryx_idx=as.matrix(uncoryx_idx)
uncory_idx=uncoryx_idx[,1] #index of gene
uncorx_idx=uncoryx_idx[,2] #index of cis-eQTL

uniqy_idx=unique(uncoryx_idx[,1]) #unique index of gene
uniqx_idx=unique(uncoryx_idx[,2]) #unique index of cis-eQTL
write.table(uniqy_idx,"uniqy_idx",row.names=F,col.names=F,quote=F,sep=" ")
write.table(uniqx_idx,"uniqx_idx",row.names=F,col.names=F,quote=F,sep=" ")

nety=y[,uniqy_idx] #network gene expression data
netx=x[,uniqx_idx] #network cis-eQTL data
write.table(nety,"nety",row.names=F,col.names=F,quote=F,sep=" ")
write.table(netx,"netx",row.names=F,col.names=F,quote=F,sep=" ")

lx=length(uncorx_idx)
netyx_idx=matrix(0,lx,2)
for (i in 1:lx){
  netyx_idx[i,1]=which(uniqy_idx==uncory_idx[i]) #index of gene in network gene data
  netyx_idx[i,2]=which(uniqx_idx==uncorx_idx[i]) #index of cis-eQTL in network cis-eQTL data
}
write.table(netyx_idx,"netyx_idx",row.names=F,col.names=F,quote=F,sep=" ")
```

12.28 netgene.info.R

The following is the *R* scripts in `netgene.info.R`:

```
### netgene.info.R
### Get information (symbol & position) of genes in networks data
#
gene_pos=read.table("net.genepos")
uniqy_idx=read.table("uniqy_idx")
gene_pos=as.matrix(gene_pos)
uniqy_idx=as.matrix(uniqy_idx)
nety_geneinfo=gene_pos[uniqy_idx,]
write.table(nety_geneinfo,"nety_geneinfo",row.names=F,col.names=F,quote=F,sep=" ")

nety_genesymbol=gene_pos[uniqy_idx,1]
write.table(nety_genesymbol,"nety_genesymbol",row.names=F,col.names=F,quote=F,sep=" ")
```

12.29 iclars.R

Here are the *R* scripts which computes the information-criteria-optimal regression coefficients for lasso (using AIC, BIC, or EBIC). The function `iclars()` will be used by `ebicalasso()`.

```
# Computes the IC-optimal regression coefficients for lasso: AIC, BIC, or EBIC
#
iclars <- function(X, y, ic.lars=c("bic","ebic","aic"), gamma.ebic=NULL, use.Gram=TRUE, normalize=TRUE,
{
  eps <- .Machine$double.eps

  ic.lars <- ic.lars[1]
  n <- length(y)
  if (use.Gram == TRUE) {
    type <- "covariance"
  }
  if (use.Gram == FALSE) {
    type <- "naive"
  }
  fit <- glmnet(X, y, family = "gaussian", standardize = normalize,
               type.gaussian = type, intercept = intercept)
  lambda <- fit$lambda

  # extract coefficients at all values of lambda,  including the intercept
  coef.beta <- rbind(fit$a0,as.matrix(fit$beta))

  if( (ic.lars=="ebic")&&(is.null(gamma.ebic)) )
    gamma.ebic <- min(1,max(1+eps-log(n)/(2*log(dim(x)[2])),0))

  obj <- deviance(fit) + switch(ic.lars,
           bic = log(n)*fit$df,
           aic = 2*fit$df,
           ebic =log(n)*fit$df + 2*gamma.ebic*log(choose(dim(x)[2],fit$df)))

  lambda.opt <- lambda[which.min(obj)]

  coefficients <- predict(fit, type = "coefficients", s = lambda.opt)
  inter <- coefficients[1]
  coefficients <- coefficients[-1]
  names(coefficients) <- 1:ncol(X)
  object <- list(lambda=lambda, ic=obj, lambda.opt=lambda.opt,
                 intercept=inter, coefficients=coefficients)
  invisible(object)
}
```

12.30 ebicalasso.R

The function `ebicallaso()` implments the adaptive LASSO with the tuning parameter chosen via the extended BIC. Note that, the initial parameters are choosen by LASSO, whose tuning parameter may be chosen with other information criteria (BIC by default), see `iclars.R` for details.

```
# Adaptive LASSO with the tuning parameter chosen by EBIC
#
ebicalasso <- function(X, y, gamma.ebic=NULL, ic.lars=c("bic","ebic","aic"), use.Gram=TRUE, both=TRUE,
{
  eps <- .Machine$double.eps

  ic.lars <- ic.lars[1]
  colnames(X) <- 1:ncol(X)
  n <- length(y)
  lassofit <- iclars(X, y, ic.lars=ic.lars, gamma.ebic=gamma.ebic,
                     use.Gram=use.Gram, normalize=TRUE, intercept=intercept)
  coefficients.lasso <- lassofit$coefficients
  intercept.lasso <- lassofit$intercept

  lambda <- lassofit$lambda
  lambda.lasso <- lassofit$lambda.opt
  coefficients.alasso <- coefficients.lasso
  lambda.alasso <- 0
  intercept.alasso <- intercept.lasso
  if (use.Gram == TRUE) {
    type <- "covariance"
  }
  if (use.Gram == FALSE) {
    type <- "naive"
  }

  if (both == TRUE) {
    weights <- 1/abs(coefficients.lasso[abs(coefficients.lasso)>0])
    names(coefficients.alasso) <- 1:ncol(X)
    if (length(weights) > 1) {
      if( is.null(gamma.ebic) )
        gamma.ebic <- min(1,max(1+eps-log(n)/(2*log(length(weights))),0))

      XX <- X[, names(weights), drop = FALSE]
      XX <- scale(XX, center = FALSE, scale = weights)

      fit <- glmnet(XX, y, type.gaussian=type, standardize=FALSE,
                    intercept=intercept)
      # extract coefficients at all values of lambda,  including the intercept
      coef.beta <- rbind(fit$a0,as.matrix(fit$beta))

      obj <- deviance(fit) + log(n)*fit$df + 2*gamma.ebic*log(choose(dim(XX)[2],fit$df))
      lambda.alasso <- fit$lambda[which.min(obj)]

      coefficients <- predict(fit, type="coefficients", s=lambda.alasso)
      inter <- coefficients[1]

      intercept.alasso <- coefficients[1]
      coefficients.alasso[names(weights)] <- coefficients[-1]/weights
    }
  }

  return(list(lambda.lasso=lambda.lasso, lambda.alasso=lambda.alasso,
              intercept.lasso=intercept.lasso, intercept.alasso=intercept.alasso,
              coefficients.lasso=coefficients.lasso, coefficients.alasso=coefficients.alasso))
}
```

12.31 tspls.R

Caution: This files and related files should be modified to include genes which do not have cis-eQTL, so as to only identify their regulatory genes.

The following is the *R* scripts in `tspls.R`:

```
### tspls.R
### Two-Stage Penalized Least Squares
#
### Load data
y=read.table("nety") #expression data for genes with cis-eQTL
y=as.matrix(y)
x=read.table("netx") #cis-eQTL data
x=as.matrix(x)
netyx_idx=read.table("netyx_idx") #Col 1 is index of gene, Col 2 is index of corresponding cis-eQTL
netyx_idx=as.matrix(netyx_idx)
py=dim(y)[2]
px=dim(x)[2]
N=dim(y)[1]

library(MASS)
library(parcor)
library(SIS)
source("iclars.R")
source("ebicalasso.R")

### Index of instruments for each gene
y_idx=unique(netyx_idx[,1]) #unique index of gene
IV_idx=list()
for (i in 1:py){
  #index of cis-eQTL (IV) for each gene
  IV_idx[[i]]=netyx_idx[which(netyx_idx[,1]==y_idx[i]),2]
}
```

```r
# number of remaining IVs for ISIS screening
nsis = floor(N/log(N))

### Stage 1: Prediction for Y using X with prescreened instruments.
### Ridge Regression; CV to select # of components
lambda=matrix(0,py,1)
coeff=matrix(0,px,py)
ypre=matrix(0,N,py)
lambdaseq=seq(0,0.1,0.001)
for(i in 1:py) {
    sis_fit <- SIS(x, y[,i], family = "gaussian", iter = TRUE, nsis = nsis, penalty = "lasso")
    sis_screened_set <- sis_fit$ix0
    fit=lm.ridge(y[,i]~x[,sis_screened_set],lambda=lambdaseq)
    lambda[i]=lambdaseq[which.min(fit$GCV)]
    coeffi=fit$coef[,which.min(fit$GCV)]
    ypre[,i]=scale(x[,sis_screened_set],center=T,scale=fit$scales)%*%coeffi+mean(y[,i])
    print(i)
}

### Stage 2: Adaptive lasso to identify network structure \gamma_i

# transform the problem into adaptive lasso problem
ynew=matrix(0,N,py)
iy=list()
cy=list()

x=scale(x)
y=scale(y)
ypre=scale(ypre)
for(i in 1:py) {
    cb=IV_idx[[i]]
    xi=x[,cb]
    Pi=diag(N)-xi%*%solve(t(xi)%*%xi)%*%t(xi)
    ynew[,i]=Pi%*%y[,i]
    ynew[,i]=scale(ynew[,i])
    yprenewi=Pi%*%ypre
    yprenewi=scale(yprenewi)
    tmpy=yprenewi[,-i]
    output<-ebicalasso(tmpy,ynew[,i]))
    #iy[[i]]<-sort(which(!output[[6]]==0))
    #cy[[i]]<-output[[6]]
    iy[[i]]<-which(!output[[6]]==0)
    cy[[i]]<-output[[6]][iy[[i]]]
    print(i)
}

### Adjacency matrix
B=matrix(0,py,(py-1))
A=matrix(0,py,py)
for (i in 1:py) {
  B[i,iy[[i]]]=1
  A[i,-i]=B[i,] #A[i,j]==1 <=> j regulates i
}
```

```
### Coefficient matrix
D=matrix(0,py,(py-1))
C=matrix(0,py,py)
for (i in 1:py) {
    D[i,iy[[i]]]=cy[[i]]
    C[i,-i]=D[i,]
}

### Save results and all objects
write.matrix(A,file="adjacency_matrix")
write.matrix(C,file="coefficient_matrix")
save.image(file="network.RData")
```

12.32 bts1template.R

The following is the *R* scripts in `bts1template.R`:

```
### bts1template.R
### Template R file for Stage 1 of bootstrap
#
### Load data
Y=read.table("nety") #expression data for genes with cis-eQTL
Y=as.matrix(Y)
X=read.table("netx") #cis-eQTL data
X=as.matrix(X)
netyx_idx=read.table("netyx_idx") #Col 1 is index of gene, Col 2 is index of corresponding cis-eQTL
netyx_idx=as.matrix(netyx_idx)
Y=Y[,YYfirstYY:YYlastYY]
N=dim(Y)[1]
py=dim(Y)[2]
px=dim(X)[2]

library(MASS)
library(SIS)

### Index of instruments for each gene
y_idx=unique(netyx_idx[,1]) #unique index of gene
y_idx=y_idx[YYfirstYY:YYlastYY]
IV_idx=list()
for (i in 1:py){
    IV_idx[[i]]=netyx_idx[which(netyx_idx[,1]==y_idx[i]),2] #index of cis-eQTL (IV) for each gene
}

### Bootstrap
idx=read.table('IDXXXbsXX')
idx=as.matrix(idx)
y=Y[idx,]
x=X[idx,]

### Stage 1: prediction for y using x (after removing constant columns)

### remove constant columns in x
```

```
ncx=x[,which(apply(x,2,sd)!=0)]
ncpx=dim(ncx)[2]

# number of retained IVs after ISIS
nsis = floor(N/log(N))

### Ridge Regression; CV to select # of components
lambda=matrix(0,py,1)
coeff=matrix(0,ncpx,py)
ypre=matrix(0,N,py)
lambdaseq=seq(0,0.1,0.001)
for (i in 1:py) {
  sis_fit <- SIS(ncx, y[,i], family = "gaussian", iter = TRUE, nsis = nsis, penalty = "lasso")
  sis_screened_set <- sis_fit$ix0
  fit=lm.ridge(y[,i]~ncx[,sis_screened_set],lambda=lambdaseq)
  coeffi=fit$coef[,which.min(fit$GCV)]
  ypre[,i]=scale(ncx[,sis_screened_set],center=T,scale=fit$scales)%*%coeffi+mean(y[,i])
  print(i)
}

### save results and all objects
write.table(ypre,file='ypreXXbsXX_YYfirstYY-YYlastYY',row.names=F,col.names=F,quote=F,sep=" ")
```

12.33 bts1template.sh

The following is the *Bash* scripts in bts1template.sh:

```
#!/bin/bash
#PBS -q standby
#PBS -l nodes=1:ppn=20
#PBS -l walltime=4:00:00

cd $PBS_O_WORKDIR
unset DISPLAY

module load utilities
module load r
export OPENBLAS_NUM_THREADS=1
export MKL_NUM_THREADS=1

module load parafly
ParaFly -c paramsXXX.txt -CPU 20 -failed_cmds rerunXXX.txt
```

12.34 bts1genbatch.sh

Below is the file bts1genbatch.sh for generating the batch R files using bts1template.R:

```
#!/bin/sh

for i in {1..100}
do
    for j in {1..40}
    do
        A=`expr $j \* 150 - 149`
        B=`expr $j \* 150`
        perl -pe 's/XXbsXX/'$i'/e; s/YYfirstYY/'$A'/e; s/YYlastYY/'$B'/e' < ./codes/bts1template.R > bs
        echo 'R CMD BATCH bs'$i'_'$A-$B'.R' >> params.txt
    done
done
```

12.35 bts1gensub.sh

Below is the `bts1gensub.sh` file for generating the batch sub files using `bts1template.sh`:

```
#!/bin/sh

echo "#!/bin/sh" > qsub1.sh
chmod +x qsub1.sh

for i in {1..200}
do
    A=`expr $i \* 20 - 19`
    B=`expr $i \* 20`
    awk "NR >= $A && NR <= $B {print}" < params.txt > params$i.txt
    perl -pe 's/XXX/'$i'/g' < ./codes/bts1template.sh > sub$i.sh
    echo "qsub sub$i.sh" >> qsub1.sh
done
```

12.36 bts2template.R

The following is the *R* scripts in `bts2template.R`:

```
### bts2template.R
### Template R file for Stage 2 of bootstrap:
###    variable selection for Y(-i) hat with its own instrument
#
### Load data
Y=read.table("nety") #expression data for genes with cis-eQTL
X=read.table("netx") #cis-eQTL data
Y=as.matrix(Y)
X=as.matrix(X)
netyx_idx=read.table("netyx_idx") #Col 1 is index of gene, Col 2 is index of corresponding cis-eQTL
netyx_idx=as.matrix(netyx_idx)
Y=Y[,YYfirstYY:YYlastYY]
py=dim(Y)[2]
px=dim(X)[2]
N=dim(Y)[1]

library(MASS)
library(parcor)
library(matrixcalc)
source("iclars.R")
source("ebicalasso.R")

### Index of instruments for each gene
```

```r
y_idx=unique(netyx_idx[,1]) #unique index of gene
y_idx=y_idx[YYfirstYY:YYlastYY]
IV_idx=list()
for (i in 1:py){
    IV_idx[[i]]=netyx_idx[which(netyx_idx[,1]==y_idx[i]),2] #index of cis-eQTL (IV) for each gene
}

### Bootstrap
idx=read.table('IDXXXbsXX')
idx=as.matrix(idx)
y=Y[idx,]
x=X[idx,]

### Predicted y
ypre=read.table("ypreXXbsXX")
ypre=as.matrix(ypre)
nypre=dim(ypre)[2]

### Screening and variable selection for y(-i) hat, its instrument is always in the model
ynew=matrix(0,N,py)
iy=list()
cy=list()

for (i in 1:py) {
  ### transform the problem into adaptive lasso problem
  cb=IV_idx[[i]] #index of cis-eQTL for the i-th y in ncy
  xi=x[,cb]
  xi=as.matrix(xi)
  xi=xi[,which(apply(xi,2,sd)!=0)] #remove constant columns
  xi=as.matrix(xi)
  xi=scale(xi)
  xi=as.matrix(xi)
  nxi=dim(xi)[2]
  while (nxi>1 & is.singular.matrix(t(xi)%*%xi)){
    xi=xi[,-1]
    nxi=nxi-1
  } #remove one column if t(xi)%*%xi is singular
  #xi=xi[,!duplicated(round(abs(xi),8),MARGIN=2)] #remove one of two columns which have perfect linear
  #two columns which have perfect linear correlation have the same absolute value after standardization

  if(nxi>0){
    Pi=diag(N)-xi%*%solve(t(xi)%*%xi)%*%t(xi)
    y[,i]=scale(y[,i])
    ynew[,i]=Pi%*%y[,i]
    ynew[,i]=scale(ynew[,i])
    yprenewi=Pi%*%ypre
    yprenewi=scale(yprenewi)
    tmpy=yprenewi[,-(YYfirstYY-1+i)]
    output<-ebicalasso(tmpy,ynew[,i])
    iy[[i]]<-sort(which(!output[[6]]==0))
    cy[[i]]<-output[[6]]
    print(i)
  }
}

### Adjacency matrix
B=matrix(0,py,(nypre-1))
A=matrix(0,py,nypre)
for (i in 1:py) {
  B[i,iy[[i]]]=1
  A[i,-i]=B[i,] #A[i,j]==1 <=> j regulates i
}

### Coefficient matrix
D=matrix(0,py,(nypre-1))
C=matrix(0,py,nypre)
for (i in 1:py) {
    D[i,iy[[i]]]=cy[[i]]
    C[i,-i]=D[i,]
}

### Save results and all objects
write.table(A,'AdjMatXXbsXX_YYfirstYY-YYlastYY',row.names=F,col.names=F,quote=F,sep=" ")
write.table(C,'CoeffMatXXbsXX_YYfirstYY-YYlastYY',row.names=F,col.names=F,quote=F,sep=" ")
```

12.37 bts2template.sh

Below is the template file `bts2template.sh`:

```
#!/bin/bash
#PBS -q standby
#PBS -l nodes=1:ppn=20
#PBS -l walltime=4:00:00

cd $PBS_O_WORKDIR
unset DISPLAY

module load utilities
module load r
export OPENBLAS_NUM_THREADS=1
export MKL_NUM_THREADS=1

module load parafly
ParaFly -c paramsXXX.txt -CPU 20 -failed_cmds rerunXXX.txt
```

12.38 bts2genbatch.sh

Below is the `bts2genbatch.sh` file for generating the batch R files using `bts2template.R`:

```
#!/bin/sh

for i in {1..100}
do
    for j in {1..40}
    do
```

```
        A=`expr $j \* 150 - 149`
        B=`expr $j \* 150`
        perl -pe 's/XXbsXX/'$i'/e; s/YYfirstYY/'$A'/e; s/YYlastYY/'$B'/e' < ./codes/bts2template.R > bs
        echo 'R CMD BATCH bs'$i'_'$A-$B'.R' >> params.txt
    done
done
```

12.39 bts2gensub.sh

Below is the `bts2gensub.sh` file for generating the batch sub files using `bts2template.sh`:

```
#!/bin/sh

echo "#!/bin/sh" > qsub2.sh
chmod +x qsub2.sh

for i in {1..200}
do
    A=`expr $i \* 20 - 19`
    B=`expr $i \* 20`
    awk "NR >= $A && NR <= $B {print}" < params.txt > params$i.txt
    perl -pe 's/XXX/'$i'/g' < ./codes/bts2template.sh > sub$i.sh
    echo "qsub sub$i.sh" >> qsub2.sh
done
```

12.40 Afreq.R

Below is the file `Afreq.R` to calculate the frequency of each edge of the network:

```
### Afreq.R
### Calculate frequence of each edge
#
Afiles=list.files(pattern='AdjMat')
A=lapply(Afiles,read.table)
N=length(A)
N

Afreq=0
for (i in 1:N) {
    Afreq=Afreq+as.matrix(A[[i]]/N)
    print(i)
}
write.table(Afreq,'Afreq',row.names=F,col.names=F)

########################################################
### Number of edges with bootstrap frequency > thre
Afreq=read.table('Afreq')
threlist=c(0.8,0.9,0.95,1)

for (thre in threlist){
  A=(Afreq>=thre)
  print(sum(A))
}
```

```
########################################################
### Edge list with gene symbol
genesym=read.table("nety_genesymbol")
genesym=as.matrix(genesym)

for (thre in threlist){
  A=(Afreq>=thre)

  edgeidx=which(A!=0,arr.ind=T)

  target=genesym[edgeidx[,1],]
  source=genesym[edgeidx[,2],]
  edgefreq=Afreq[edgeidx]

  edgelist_genesymbol=cbind(source,target,edgefreq) #Col 1: source, Col 2: target, Col 3: frequency

  write.table(edgelist_genesymbol,paste0("edgelist_",thre),row.names=F,col.names=c("source","target","f
}
```

Caution: The UNIX editor *vim* may be used to create and edit a file.

Caution: When `*.pl` files are created, make sure that those files are executable. For example, to make `splitchr.pl` excutable, run the following command in *Bash*:

```
chmod ugo+x splitchr.pl
```