saegus Documentation

Release 0.1.2

John J. Dougherty III

April 04, 2016

Contents:

CHAPTER

ONE

MAGIC POPULATION WITH SUBSET OF LOCI

The `standard_magic` population uses all 7386 loci from `hapmap3.txt`. However, if I use all 7386 loci and the triplet scheme this alters the GWAS results. I need to create a population which uses the loci which occupy integer-valued positions on the genetic map i.e. 1.0 cM, 2.0 cM. `standard_magic` has the loci at 0.6 cM, 0.8 cM, 1.0 cM, 1.2 cM so on and so forth.

Allele Effects and Triplets of Non-Recombining Loci

The `standard_magic` population assigns effects to an integer-valued position and the positions immediately upstream and downstream. When I use the subset of 1478 loci I will assign the alleles at those loci effects as **three independent** draws from an exponential distribution. The goal is to make `magic1478` somewhat comparable to `standard_magic`.

Strategy of Creating MAGIC1478

I am going to use the `nam_prefounders` and withdraw the integer-valued positions of each individual. This saves me the work of building up the population from the raw files.

Creating the simuPOP Population Object for MAGIC1478

A simuPOP `Population` requires the number of chromosomes and the number of loci per chromosome. Luckily I saved the absolute indexes of the integer-valued positions of the genetic map in a `shelve` instance.

```
misc_gmap = shelve.open('misc_gmap')
integral_valued_loci = misc_gmap['integral_valued_loci']
relative_integral_valued_loci = misc_gmap['relative_integral_valued_loci']
print(integral_valued_loci[:10])
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```

`relative_integral_valued_loci` is a `dict` which is keyed by absolute locus index. We will extract the chromosome of each absolute valued index and then use `collections.Counter` to count how many loci are on each chromosome.

```
print(relative_integral_valued_loci[4])
(1, -4.0)

integer_loci_per_chromosome = [relative_integral_valued_loci[abs_locus][0] for abs_locus in integral_

import collections as col
```

```
integer_valued_loci_per_chromosome = col.Counter(loci_per_chromosome)
print(integer_valued_loci_per_chromosome)

Counter({1.0: 210, 3.0: 164, 2.0: 161, 5.0: 157, 4.0: 152, 7.0: 139, 8.0: 138, 9.0: 132, 10.0: 113, 6

loci_per_chromosome = tuple(integer_valued_loci_per_chromosome.values())

print(loci_per_chromosome)
(210, 161, 164, 152, 157, 112, 139, 138, 132, 113)
```

So then I can simply do:

```
prefounders_1478 = sim.Population(size=26, ploidy=2, loci=loci_per_chromosome)
```

This initializes the population and now I can copy over genotypes at the integer valued positions from the prefounders7386 population.

```
prefounders7386 = sim.loadPopulation('nam_prefounders.pop')
for ind_7386, ind_1478 in zip(prefounders_7386.individuals(), prefounders_1478.individuals()):
   sub_alpha = [ind_7386.genotype(ploidy=0)[integer_position]
               for integer_position in integer_subset]
   sub_beta = [ind_7386.genotype(ploidy=1)[integer_position]
              for integer_position in integer_subset]
   genotype_1478 = sub_alpha + sub_beta
   ind_1478.setGenotype(genotype_1478)
```

Generating Standard MAGIC1478 from Prefounders1478

I followed the same mating and testing strategy to make a `standard_magic_1478.pop` file. I used the same founders as in `standard_magic` i.e. founders 1 through 8.

```
prefounders_1478 = sim.loadPopulation('prefounders_1478.pop')
```

Determine the Mating Pairs of Each Generation

I created a standardized MAGIC1478 population as I did with MAGIC7386. At each step I pre-determine the mating pairs and record them in `lists` which have the title `expected_x_mother_ids` and `expected_x_father_ids`. The expected parental id pairs are mated in order. The offspring have infoFields which record the ID of their mother and ID father.

After mating the offspring parental IDs are compared with the expected parental IDs. Below is an example of this mating-testing cycle.

```
first_sp_mothers = [random.choice(pop.indInfo('ind_id', 0)) for i in range(1000)]
second_sp_fathers = [random.choice(pop.indInfo('ind_id', 1)) for i in range(1000)]
third_sp_mothers = [random.choice(pop.indInfo('ind_id', 2)) for i in range(1000)]
fourth_sp_fathers = [random.choice(pop.indInfo('ind_id', 3)) for i in range(1000)]

expected_f_two_mother_ids = first_sp_mothers + third_sp_mothers
expected_f_two_father_ids = second_sp_fathers + fourth_sp_fathers
```

The expected parental IDs are written to disk using a `shelve` for post-comparison should it be necessary.

```
breeding_parameters['expected_f_two_mother_ids'] = expected_f_two_mother_ids
breeding_parameters['expected_f_two_father_ids'] = expected_f_two_father_ids
```

A `parent_chooser` is initiated which determines how offspring are created.

```
second_order_pc = breed.SecondOrderPairIDChooser(expected_f_two_mother_ids, expected_f_two_father_ids
```

Then mating occurs:

```
pop.evolve(
    matingScheme=sim.HomoMating(
        sim.PyParentsChooser(second_order_pc.snd_ord_id_pairs),
        sim.OffspringGenerator(ops=[
            sim.IdTagger(),
            sim.ParentsTagger(),
            sim.PedigreeTagger(),
            sim.Recombinator(rates=0.01)
        ],
            numOffspring=1),
        subPopSize=[2000],
    ),
    gen=1,
)
```

We organize mother and father IDs into `observed` lists and compare them to the expected IDs. We count the number of matches between expected and observed mother IDs and expected and observed father IDs. The number should be equal to the population size.

```
breeding_parameters['expected_f_two_mother_ids'] = expected_f_two_mother_ids
breeding_parameters['expected_f_two_father_ids'] = expected_f_two_father_ids

breeding_parameters['observed_f_two_mother_ids'] = observed_f_two_mother_ids
breeding_parameters['observed_f_two_father_ids'] = observed_f_two_father_ids

breeding_parameters['number_of_matches_f_two_mother_ids'] = sum(np.equal(expected_f_two_mother_ids, o
breeding_parameters['number_of_matches_f_two_father_ids'] = sum(np.equal(expected_f_two_father_ids, o

assert breeding_parameters['number_of_matches_f_two_father_ids'] == 2000, "Incorrect father IDs."

breeding_parameters['number_of_matches_f_two_mother_ids'] == 2000, "Incorrect mother IDs."
```

The function `np.equal()` checks if the IDs match by location so the order is preserved. Otherwise the script will crash via an `AssertionError`.

Parameter Set Stored Using `shelve`

I used the `shelve` module to store the parameters and entire mating history of `standard_magic`. I did the same thing for `magic_1478`.

```
m1478_sim_parameters = shelve.open('magic_1478_simulation_parameters')
m1478_sim_parameters['founders'] = founders
m1478_sim_parameters['number_of_replicates'] = 5
m1478_sim_parameters['prefounder_file_name'] = 'prefounders_1478.pop'
m1478_sim_parameters['alleles'] = magic1478_alleles
m1478_sim_parameters['operating_population_size'] = 2000
m1478_sim_parameters['recombination_rates'] = [0.01]*1478
m1478_sim_parameters.close()

m1478_trait_parameters = shelve.open('magic_1478_trait_parameters')
m1478_trait_parameters['number_of_qtl'] = 10
```

```
m1478_trait_parameters['allele_effect_parameters'] = 1
m1478_trait_parameters.close()
```

CHAPTER

TWO

GENERATING INPUT FOR TASSEL WITH MAGIC1478

This document records the exact steps I took to create a set of data which serves as input to TASSEL for mixed-linear-modeling. In this case we are simply performing random mating to create the population which we will analyze. Other times we will use a recurrently selected population for analysis.

Allele Effects for MAGIC1478

Our usual population uses all 7386 loci and assigns appropriate `recombination_rates` to create triplets of non-recombining loci. In an attempt to make MAGIC1478 and MAGIC7386 comparable I have assigned allele effects in MAGIC1478 as three independent draws for each allele at each locus. For example:

```
qtl = [2, 10, 20]
```

The alleles at loci `qtl` are:

```
alleles[2], alleles[10], alleles[20]

([3, 1], [1, 3], [3, 0])
```

So then we assign an effect to each `allele` at each `locus` as such:

```
allele_effects = {}
for locus in qtl:
   allele_effects[locus] = {}
   for allele in alleles[locus]:
      allele_effects[locus][allele] = sum([random.expovariate(1) for i in range(3)])
```

```
allele_effects

{
   2: {1: 3.57, 3: 1.874},
   10: {1: 3.29, 3: 3.44},
   20: {0: 2.05, 3: 3.03},
}
```

I defined a function which generalizes assignment of allele effects *assign-any-allele-effects*

Random Mating MAGIC1478 for GWAS

```
def assign_additive_g(pop, qtl, allele_effects):
    """
    Calculates genotypic contribution ``g`` by summing the effect of each
    allele at each QTL triplet.
    """
    for ind in pop.individuals():
        genotypic_contribution = \
            sum([
                    allele_effects[locus][ind.genotype(ploidy=0)[locus]] +\
                    allele_effects[locus][ind.genotype(ploidy=1)[locus]]
                 for locus
                 in qtl])
        ind.g = genotypic_contribution
```

```
magic1478 = sim.loadPopulation('magic_1478.pop')
```

```
sim.tagID(magic1478, reset=False)

genetic_map = shelve.open('magic_1478_genetic_map')
history = shelve.open('magic_1478_history')
simulation = shelve.open('magic_1478_simulation_parameters')
trait = shelve.open('magic_1478_trait_parameters')

locus_1478_names = list(range(1478))
pos_1478_column = list(range(1478))

breed_magic_1478 = breed.MAGIC(magic1478, simulation['recombination_rates'])
breed_magic_1478.interim_random_mating(3, 2000)
```

```
Initiating interim random mating for 3 generations.
Generation: 3
Generation: 4
Generation: 5
```

Determining G and P in the 1478 Population

```
ae = assign_allele_effects(simulation['alleles'], trait['qtl'], 3, random.expovariate, 1)
```

```
ae
{2: {1: 1.6039383268614498, 3: 2.795016834003455},
 10: {1: 3.3259920171422936, 3: 3.1695014054478565},
 20: {0: 2.4204478909872953, 3: 4.269861858273051}}
```

```
assign_additive_g(magic1478, qtl, ae)
```

```
def calculate_error_variance(pop, heritability):
    """
    Calculates the parameter ``epsilon`` to be used as the variance
    of the error distribution. The error distribution generates noise
    found in real experiments.
    """
    variance_of_g = np.var(pop.indInfo('g'))
    epsilon = variance_of_g*(1/heritability - 1)
    pop.dvars().epsilon = epsilon

def phenotypic_effect_calculator(pop):
    """
    Simulate measurement error by adding random error to genotypic
    contribution.
    """
    for ind in pop.individuals():
        ind.p = ind.g + random.normalvariate(0, pop.dvars().epsilon)
```

```
heritability = 0.7
calculate_error_variance(magic1478, heritability)
print(magic1478.dvars().epsilon)
phenotypic_effect_calculator(magic1478)
```

```
0.849336297482
```

```
trait['heritability'] = heritability
trait['epsilon'] = magic1478.dvars().epsilon
trait['qtl'] = qtl
trait['allele_effects'] = ae
trait['g'] = list(magic1478.indInfo('g'))
trait['p'] = list(magic1478.indInfo('p'))
```

```
trait.close()
```

```
print(np.var(pop.indInfo('p')), np.mean(pop.indInfo('p')))
```

```
np.var(pop.indInfo('p'))
```

```
import collections as col
```

```
Design = col.namedtuple("Design", ["genetic", "history", "simulation", "trait"])
```

```
trait[]
```

```
trait['allele_effects'] = allele_effects
trait['qtl'] = qtl
```

```
trait['heritability'] = heritability
trait['epsilon'] = pop.dvars().epsilon
trait['g'] = list(pop.indInfo('g'))
trait['p'] = list(pop.indInfo('p'))
```

```
trait.close()
```

```
trait = shelve.open('magic_1478_trait_parameters')
```

```
history.close()
```

```
def calc_error_variance(pop, heritability, *args, **kwargs):
    operators.CalculateErrorVariance(heritability, *args, **kwargs).apply(pop)
```

```
for magic_rep in multi_std_pop.populations():
    calc_error_variance(magic_rep, 0.7)
```

```
multi_g = {0: list(multi_std_pop.population(0).indInfo('g')),
           1: list(multi_std_pop.population(1).indInfo('g'))}
```

```
multi_p = {0: list(multi_std_pop.population(0).indInfo('p')),
           1: list(multi_std_pop.population(1).indInfo('p'))}
```

```
multi_g[1] == multi_g[0]
```

```
for magic_rep in multi_std_pop.populations():
    pheno_calc(magic_rep, 0.05)
```

```
trait_parameter_storeage = shelve.open("magic_random_trait_parameters")
trait_parameter_storeage['triplet_qtl'] = triplet_qtl
trait_parameter_storeage['allele_effects'] = allele_effects
trait_parameter_storeage['epsilon'] = epsilon_reps
trait_parameter_storeage['g'] = multi_g
trait_parameter_storeage['p'] = multi_p
trait_parameter_storeage.close()
```

```
import importlib as imp
```

```
for i in range(2):
    sim.stat(multi_std_pop.population(i), alleleFreq=sim.ALL_AVAIL, vars=[''])
```

```
imp.reload(analyze)
```

```
pop.dvars().qtl = qtl
pop.dvars().allele_effects = allele_effects
```

```
alleles = simrams['alleles']
```

```
alleles
```

```
sim
```

```
pop
```

```
simupop.stat(pop, alleleFreq=simupop.ALL_AVAIL)
```

```
frq = analyze.Frq(magic1478)
```

```
af = frq.allele_frequencies(magic1478, alleles, list(range(1478)))
```

```
minor_alleles = np.array([af['minor', 'alleles'][locus] for locus in range(1478)])
```

```
ties = [locus for locus in range(magic1478.totNumLoci())
        if af['minor', 'alleles'][locus] == af['major', 'alleles'][locus]]

for st in ties:
    af['major', 'alleles'][st] = list(magic1478.dvars().alleleFreq[st])[0]
    af['minor', 'alleles'][st] = list(magic1478.dvars().alleleFreq[st])[1]
major_minor_allele_conflicts = sum(np.equal(list(af['minor', 'alleles'].values()),
                                            list(af['major', 'alleles'].values())))
```

```
pca = analyze.PCA(magic1478, range(magic1478.totNumLoci()), frq)
minor_ac = pca.calculate_count_matrix(magic1478, af['minor', 'alleles'],
                                      'minor_allele_count.txt')
eigendata = pca.svd(magic1478, minor_ac)
```

```
simulation['snp_to_integer'] = snp_to_integer
simulation['integer_to_snp'] = integer_to_snp
```

```
individual_names = {}
for ind in magic1478.individuals():
    individual_names[ind.ind_id] = str(ind.ind_id)
```

```
gwas = analyze.GWAS(magic1478, individual_names, locus_1478_names, pos_1478_column)
hmap = gwas.hapmap_formatter(integer_to_snp, 'simulated_hapmap.txt')
phenos = gwas.trait_formatter('phenotype_vector.txt')
kinship_matrix = gwas.calc_kinship_matrix(minor_ac, af, 'kinship_matrix.txt')
pop_struct_matrix = gwas.population_structure_formatter(eigendata, 'structure_matrix.txt')
#pd.DataFrame(multipop.population(i).dvars().statistics).to_csv(prefix + 'means_and_vars.txt', sep='
analyze.generate_tassel_gwas_configs('',
                                     '',
                                     '',
                                     '',
                                     'sim_mlm_gwas_pipeline.xml')
```

```
simulation.close()
```

```
hmap = gwas.hapmap_formatter(integer_to_snp, 'simulated_hapmap.txt')
```

```
def pre_GWAS_grinder(multi_pop, founder_alleles, info_prefix, triplet_qtl, allele_effects):
    for i, pop_rep in enumerate(multi_pop.populations()):
        pop_rep_id = str(pop_rep.dvars().rep)
        prefix = info_prefix + str(pop_rep_id) + '_'
        qtl = triplet_qtl[i][1:-1:3]
        pop_rep.dvars().qtl = qtl
        pop_rep.dvars().triplet_qtl = triplet_qtl[i]
        pop_rep.dvars().allele_effects = allele_effects[i]
        frq = analyze.Frq(pop_rep, )
        af = frq.allele_frequencies(pop_rep, alleles, range(pop_rep.totNumLoci()))

        ties = [locus for locus in range(pop_rep.totNumLoci())
                if af['minor', 'alleles'][locus] == af['major', 'alleles'][locus]]

        for st in ties:
            af['major', 'alleles'][st] = list(pop_rep.dvars().alleleFreq[st])[0]
            af['minor', 'alleles'][st] = list(pop_rep.dvars().alleleFreq[st])[1]
        major_minor_allele_conflicts = sum(np.equal(list(af['minor', 'alleles'].values()),
                                                    list(af['major', 'alleles'].values())))

        assert major_minor_allele_conflicts == 0, "There is a tie in at least one locus."

        pca = analyze.PCA(pop_rep, range(pop_rep.totNumLoci()), frq)


        minor_ac = pca.calculate_count_matrix(pop_rep, af['minor', 'alleles'],
                                              prefix + 'minor_allele_count.txt')

        eigendata = pca.svd(pop_rep, minor_ac)


        individual_names = {ind.ind_id: info_prefix + pop_rep_id +'_G' +
                            str(int(ind.generation)) +
                            '_I'+str(int(ind.ind_id))
                            for ind in pop_rep.individuals()}
        gwas = analyze.GWAS(pop_rep, individual_names, locus_names, pos_column)
        hmap = gwas.hapmap_formatter(integer_to_snp, prefix + 'simulated_hapmap.txt')
        phenos = gwas.trait_formatter(prefix + 'phenotype_vector.txt')
        kinship_matrix = gwas.calc_kinship_matrix(minor_ac, af, prefix + 'kinship_matrix.txt')
        pop_struct_matrix = gwas.population_structure_formatter(eigendata, prefix + 'structure_matrix
        #pd.DataFrame(multipop.population(i).dvars().statistics).to_csv(prefix + 'means_and_vars.txt
        analyze.generate_tassel_gwas_configs('',
                                             '',
                                             '',
                                             prefix,
                                             'sim_mlm_gwas_pipeline.xml')
```

```
rd_sample.indInfo('ind_id')
```

```
for st in ties:
    af['major', 'alleles'][st] = list(pop_rep.dvars().alleleFreq[st])[0]
    af['minor', 'alleles'][st] = list(pop_rep.dvars().alleleFreq[st])[1]
major_minor_allele_conflicts = sum(np.equal(list(af['minor', 'alleles'].values()),
                                            list(af['major', 'alleles'].values())))

assert major_minor_allele_conflicts == 0, "There is a tie in at least one locus."
```

```
pre_GWAS_grinder(multi_std_pop, alleles, 'magic_rdm_mating_', triplet_qtl, allele_effects)
```

```
for i, magic_rep in enumerate(multi_std_pop.populations()):

    magic_rep_id = str(magic_rep.dvars().rep)
    prefix = 'magic_RM_L10_H07_R' + str(meta_rep_id) + '_'

    qtl = triplet_qtl[i][1:-1:3]

    meta_rep.dvars().qtl = qtl
    meta_rep.dvars().triplet_qtl = triplet_qtl[i]
    meta_rep.dvars().allele_effects = allele_effects[i]


    frq = analyze.Frq(meta_rep, triplet_qtl[i], alleles, allele_effects[i])
    af = frq.allele_frequencies(meta_rep, range(meta_rep.totNumLoci()))


    #qtalleles = frq.rank_allele_effects(meta_rep, triplet_qtl[i], alleles, allele_effects[i])
    ties = [locus for locus in range(meta_rep.totNumLoci())
            if af['minor', 'alleles'][locus] == af['major', 'alleles'][locus]]

    for st in ties:
        af['major', 'alleles'][st] = list(meta_rep.dvars().alleleFreq[st])[0]
        af['minor', 'alleles'][st] = list(meta_rep.dvars().alleleFreq[st])[1]
    major_minor_allele_conflicts = sum(np.equal(list(af['minor', 'alleles'].values()),
                                                list(af['major', 'alleles'].values())))

    assert major_minor_allele_conflicts == 0, "There is a tie in at least one locus."

    #af_table = frq.allele_frq_table(meta_rep, meta_rep.numSubPop(), af,
    #                                                    recombination_rates, genetic_map)
    #qtaf_table = analyze.qt_allele_table(meta_rep, qtalleles, allele_effects[i], 10)

    #af_table.to_csv(various_simulation_info_prefix + prefix + 'allele_frequency_table.txt', sep=',',
    #qtaf_table.to_csv(various_simulation_info_prefix + prefix + 'qt_allele_info.txt', sep=',', inde
    #del af_table, qtaf_table



    pca = analyze.PCA(meta_rep, range(meta_rep.totNumLoci()), frq)


    minor_ac = pca.calculate_count_matrix(meta_rep, af['minor', 'alleles'],
                                          prefix + 'minor_allele_count.txt')

    eigendata = pca.svd(meta_rep, minor_ac)


    individual_names = {ind.ind_id: 'rs_L10_H07_R'+ meta_rep_id +'_G' +
                        str(int(ind.generation)) +
                        '_I'+str(int(ind.ind_id))
                        for ind in meta_rep.individuals()}

    ind_names_for_gwas[meta_rep_id] = individual_names

    #meta_rep.save(populations_prefix + prefix + 'metapopulation.pop')

#     names_filename = prefix + 'individual_names.yaml'
#     with open(names_filename, 'w') as name_stream:
#         yaml.dump(individual_names, name_stream)





    gwas = analyze.GWAS(meta_rep, individual_names, locus_names, pos_column)
    hmap = gwas.hapmap_formatter(integer_to_snp, prefix + 'simulated_hapmap.txt')
    phenos = gwas.trait_formatter(prefix + 'phenotype_vector.txt')
    kinship_matrix = gwas.calc_kinship_matrix(minor_ac, af, prefix + 'kinship_matrix.txt')
    pop_struct_matrix = gwas.population_structure_formatter(eigendata, prefix + 'structure_matrix.txt
    pd.DataFrame(multipop.population(i).dvars().statistics).to_csv(prefix + 'means_and_vars.txt', sep
    analyze.generate_tassel_gwas_configs('',
                                         '',
                                         '',
                                         prefix,
                                         'sim_mlm_gwas_pipeline.xml')
```

```
minor_af_vector = np.zeros(7386)
minor_af_vector[:] = [meta_rep.dvars(0).alleleFreq[locus][af['minor', 'alleles', 0][locus]]
                      for locus in range(meta_rep.totNumLoci())]

minor_alleles = np.zeros((7386), dtype=np.int8)
major_alleles = np.zeros((7386), dtype=np.int8)
minor_alleles[:] = [af['minor', 'alleles', 0][locus]
                      for locus in range(meta_rep.totNumLoci())]
major_alleles[:] = [af['major', 'alleles', 0][locus]
                       for locus in range(meta_rep.totNumLoci())]

minor_ae = np.zeros(7386)
major_ae = np.zeros(7386)
for locus in qtl:
    minor_ae[locus] = allele_effects[0][locus][minor_alleles[locus]]
    major_ae[locus] = allele_effects[0][locus][major_alleles[locus]]
```

```
avg_locus_effects = minor_af_vector*minor_ae + (1-minor_af_vector)*major_ae
```

Allele Effects Using any Number of QTL and any Distribution

I defined a function which allows the user to define allele effects for any number of `qtl`, any number of `alleles` at those loci and as random draws from any distribution.

`assign_allele_effects`(*alleles*, *qtl*, *distribution_function*, **distribution_function_parameters*, *multiplicity=1*)

Parameters

- **`alleles`** – Dictionary of alleles at each locus
- **`qtl`** – Loci which contribute to a quantitative trait
- **`distribution_function`** – Function used to determine allele effects as random draws
- **`distribution_function_parameters`** – Parameters required for the distribution function
- **`multiplicity`** – Number of independent random draws from the distribution_function to assign as alelle effects.

What do I have so far.

CHAPTER

THREE

ANALYZING RESULTS OF GWAS WITH TASSEL

We use the mixed-linear modeling method implemented in TASSEL. The MLM in TASSEL requires three files at minimum with the option for a fourth.

- genotypes in *hapmap* format
- kinship matrix
- phenotypes
- population structure matrix (optional)

`saegus` generates all four files. In this particular case the file names are:

- `simulated_hapmap.txt`
- `phenotype_vector.txt`
- `kinship_matrix.txt`
- `structure_matrix.txt`

For the sake of simplicity I placed them in same directory as the TASSEL executable file: `sTASSEL.jar`. Running an analysis with four files using the TASSEL command line interface requires typing a very long string of code. Fortunately TASSEL allows the user to create a `config` file in `xml` format. A `config` file provides a much clearer description of how TASSEL is processing the input. Below is the exact file I used to run the current analysis.

```xml
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<TasselPipeline>
    <fork1>
        <h>simulated_hapmap.txt</h>
    </fork1>
    <fork2>
        <t>phenotype_vector.txt</t>
    </fork2>
    <fork3>
        <q>structure_matrix.txt</q>
    </fork3>
    <fork4>
        <k>kinship_matrix.txt</k>
    </fork4>
    <combine5>
        <input1/>
        <input2/>
        <input3/>
        <intersect/>
    </combine5>
    <combine6>
        <input5/>
        <input4/>
        <mlm/>
        <mlmCompressionLevel>
            None
        </mlmCompressionLevel>
        <export>gwas_out_</export>
    </combine6>
    <runfork1/>
    <runfork2/>
    <runfork3/>
    <runfork4/>
</TasselPipeline>
```

The output from this analysis is a set of three files. The file names are determined by `<export>gwas_out_</export>` so that the exact file names are:

- `gwas_out_1.txt`
- `gwas_out_2.txt`
- `gwas_out_3.txt`

Given this config file and the input files we use a `.cmd` file to run the analysis in TASSEL:

```
for %%f in (sim_gwas_pipeline.xml) do (
   echo %%~nf
   run_pipeline.bat -configFile "%%f"
   )
```

The file we are primarily interested in is `gwas_out_2.txt`. This is the file which records the values and p-values of all the statistical tests. The goal is to import the `gwas_out_2.txt` into R and determine the Q values using the false discovery rate to control for false-positives.

Dataset Specifics

This is some useful information about the data-set used in the TASSEL MLM:

Population Size 2000

QTL [2, 10, 20]

```
Allele Effects

{2: {1: 1.6039383268614498, 3: 2.795016834003455},
 10: {1: 3.3259920171422936, 3: 3.1695014054478565},
 20: {0: 2.4204478909872953, 3: 4.269861858273051}}
```

QVALUES in R

We will follow Jim's tutorial to use the `qvalue` package in R; however, I have found that the function we want to use `qvalue()` does not handle missing data i.e. `NaN`. Because I am more proficient with `python` than `R` I used the `python pandas` package to convert all `NaN` p-values into values of 1.0

For example a sample of the P-values of `gwas_out_2.txt` are:

- 0.4968

- 5.6091E-28
- NaN
- 0.6236
- 0.16525

If we use the `qvalue()` function directly it will result in an error. Instead I use the values:

- 0.4968
- 5.6091E-28
- 1.0
- 0.6236
- 0.16525

The edited file name is `edited_gwas_out_2.txt`. I use these commands to obtain the q-values.

```
results_header = scan("edited_gwas_out_2.txt", what="character", nlines=1, sep="\t")
gwas_results = read.table("edited_gwas_out_2.txt", header=F, row.names=NULL, skip=2)
colnames(gwas_results) = results_header

pvalues = gwas_results$p
library(qvalue)
qobj = qvalue(p = pvalues)
qobj$qvalues
qvalues_of_magic1478_results = data.frame(qobj$qvalues)
write.table(qvalues_of_magic1478_results, "qvalues_of_magic1478.txt", sep="\t")
```

We use the `qqunif()` function in R to produce the quantile-quantile plot of the p-values.

TASSEL detected two of the three QTL: 2 and 20. The Q-values are 4.1451249e-25 and 1.606586e-73 respectively.

Two observations that are immediately obvious:

1. there is almost no difference between allele effects at locus 10
2. locus 10 is very close to 2 and 20 which might mute its already small effect

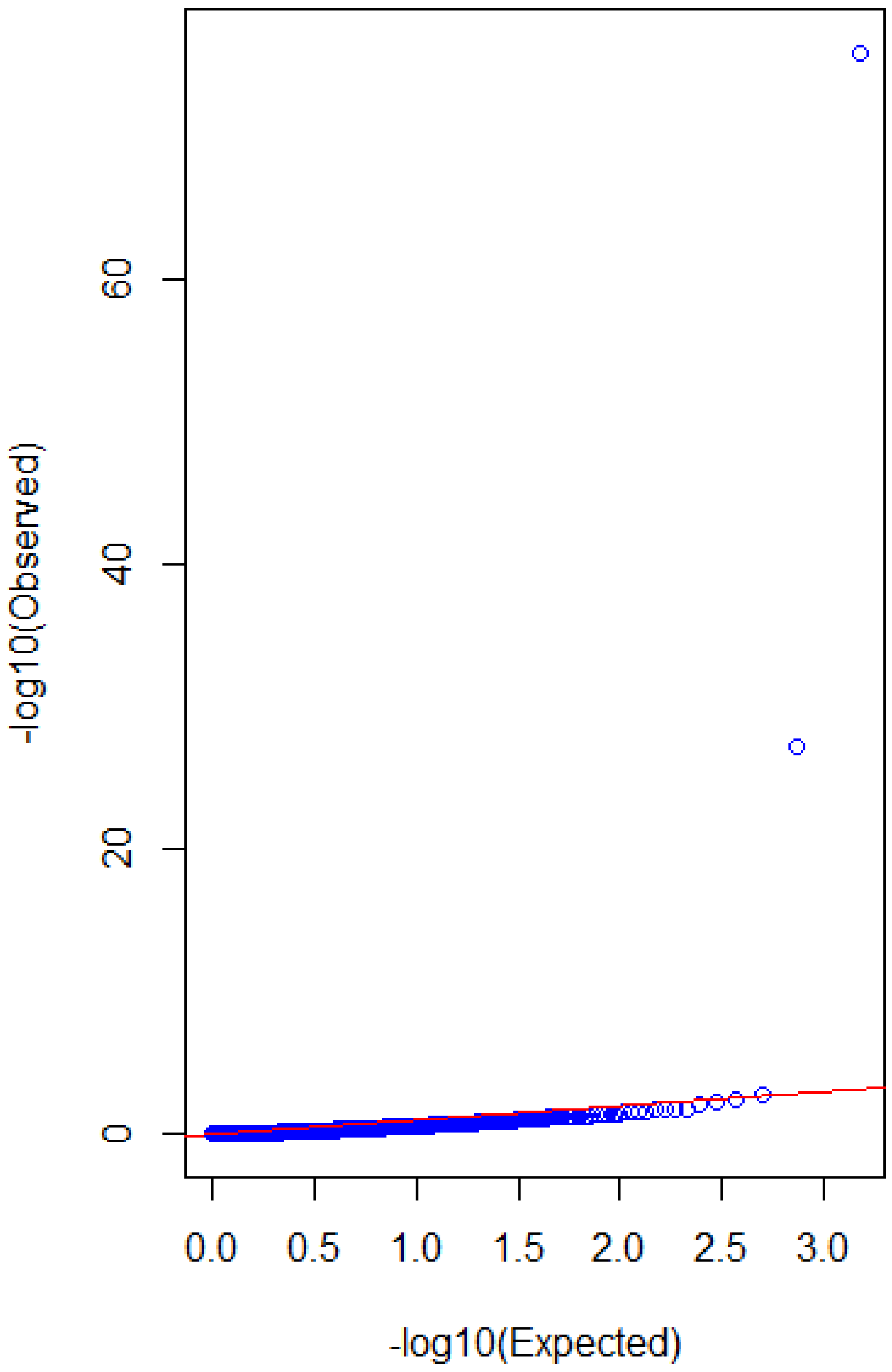

Fig. 3.1: Quantile-Quantile plot.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search