Supplemental material

Supplemental Text1: Python-code of the simulation, comments in green.

```
#########################################################################
# Individual-based Polygenic Rapid Adaptation Simulations Script (iPRASS)#
# Markus Pfenninger, Mrz. 2017                                          #
# written in python3.4                                                  #
# primarily written to evaluate the influence of                        #
# different offspring sizes on the speed of                             #
# polygenic adaptation                                                  #
#########################################################################
#load necessary modules
from random import randint, random, gauss
import numpy as np
import math
from math import sqrt, pi, exp, sin
from scipy.stats import gamma, pearsonr
import copy
#####INPUT/OUTPUT MODULE
#import parameters from params.py
import params
# Define population sizes and new selective optimum
#adult population size N
N = int(params.N)
#factor by which the juvenile population is larger than the adult population (= the
clutch size)
x_juv = int(params.x_juv)
#number of neutral loci to simulate
neutl = int(params.neutl)
#number of loci to contribute to the phenotypic trait
nloc = int(params.nloc)
#generate distribution of phenotyic values from gamma distribution
alpha = float(params.alpha)
pheno = []
for i in range(0,nloc):
       pheno.append(float(gamma.pdf(i + 1,alpha)*10 +1))
print(pheno)
#set-off for new population mean in multiples of the initial s.d.
dev = float(params.dev)
#random phenotypic plasticity added as random variate from a distribution with mean
zero and s.d. as multiple of the initial mean
phenoplast = float(params.phenoplast)
#choice of selection mode
selmode = str(params.selmode)
s = float(params.s)
#number of replicates to run
replicates = int(params.replicates)
#field to sample the time to adaptation (or failure) for the replicates
adaptime = []
adaptime50 = []
adaptime95 = []
adapfail = 0
#fields to sample info on allelefrequency shifts and more
selectmean = []
selectmax = []
selectmin = []
driftmean = []
driftmax = []
driftmin = []
drift99 = []
detectloc = []
QTfix = []
neutfix = []
phenoexpl = []
corr = []
#preparing outputfile
outfname = "Directional_N" + str(N) +"_j"+ str(x_juv) + "_l" + str(nloc) + "_a" +
str(alpha) + "_d"+ str(dev)+ "_s" + str(selmode) + str(s) + ".out"
```

```
outf = open(outfname, "w")
outf.write("adult population size N: " + str(N) + "\nx_juv: " + str(x_juv) +
"\nnumber of loci: " + str(nloc) + "\nnumber of replicates: " + str(replicates) +
"\nalpha = " + str(alpha) + "\nmode of selection and coefficient\t" + str(selmode)
+ "\t" + str(s) + "\nfitness mean set-off: " + str(dev) + "\n")
#####END INPUT/OUTPUT MODULE
#####REPLICATION OF SIMULATION
for rep in range(0,replicates):
	k = 0
	milest50 = 0
	milest95 = 0
#####TRAIT GENETIC ARCHITECTURE MODULE
	#creation of new population
	#first the genetic architecture
	phenoval = [[] for _ in range(nloc)]
	allelefreq = [[] for _ in range(nloc)]
	neutrallfreq = [[] for _ in range(neutl)]
	for loc in range(0,nloc):
	# Define phenotypic values by drawing from a normal distribution with mean
and s.d. as from field pheno
	#and initial allele frequencies for each locus
	#2 alleles at each locus
		freq = 0
		for chr in range(0,2):
			val = gauss(pheno[loc],pheno[loc] / 4)
			phenoval[loc].append(float(val))
		freq = randint(100,900)/1000
		allelefreq[loc].append(freq)
	for loc in range(0,neutl):
		freq = 0
		freq = randint(100,900)/1000
		neutrallfreq[loc].append(freq)
	print("Allelic phenotype value:  ", phenoval)
	print("Allele frequencies quantitative trait:   ", allelefreq)
	print("Allele frequencies neutral markers:   ", neutrallfreq)
	#and now the neutral loci
#####END TRAIT GENETIC ARCHITECTURE MODULE
#####POPULATION CREATION
	population = [[] for _ in range(N*x_juv)]
	#meta field where all the information for an individual is stored in a
subfield

	#creating the initial juvenile population
	for ind in range (0,(N*x_juv)):
		#assigning a random identifier to the individual (for later
randomisation purposes)
		randID = randint(0,10*N*x_juv)
		population[ind].append(randID)
		#assign phenotypically relevant alleles to individual
		for loc in range(0,nloc):
			allele = allelefreq[loc][0]
			#drawing a genotype at locus1 from the allele frequency
distribution
			rand = randint(1,100)
			if rand <= (100 * allele):
				population[ind].append(phenoval[loc][0])
			else:
				population[ind].append(phenoval[loc][1])
			rand = randint(1,100)
			if rand <= (100 * allele):
				population[ind].append(phenoval[loc][0])
			else:
				population[ind].append(phenoval[loc][1])
	#calculate the phenotypic value for the current individual
		phenotype_value = 0
		for loc in range(0,2*nloc):
			phenotype_value = phenotype_value + ((population[ind][loc +
1])/2)
		population[ind].append(phenotype_value)
		#assign neutral alleles to individual
```

```
		for loc in range(0,neutl):
			allele = neutrallfreq[loc][0]
			#drawing a genotype at locus1 from the allele frequency
distribution
			rand = randint(1,100)
			if rand <= (100 * allele):
				population[ind].append(1)
			else:
				population[ind].append(0)
			rand = randint(1,100)
			if rand <= (100 * allele):
				population[ind].append(1)
			else:
				population[ind].append(0)
	#determining the phenotypic value population parameter values

	mean = np.mean(population, axis=0)
	sd = np.std(population, axis=0)
	initial_mean = mean[nloc*2 + 1]
	#print("Mean:   " + str(mean[nloc*2 + 1]) + "  s.d.  " + str(sd[nloc*2 + 1]))

	# determining the deviation of each individual from the current phenotypic
optimum and record the value
	for i in range (0,(N*x_juv)):
		phenotype_value = phenotype_value + gauss(0,initial_mean*phenoplast)
		population[i].append(math.sqrt(np.square(population[ind][nloc*2 + 1] -
mean[nloc*2 + 1])))
	#Some stabilising selection:
	#the deviation from the optimum determines the probability to get to
reproduction
	i = 0
	j = 0
	l = 0
	while i <= N:
		p_reprod = population[j][nloc*2 + 1] / mean[nloc*2 + 1]
		randvar = randint(0,100)
		if p_reprod * 100 >= randvar:
			j = j + 1
			i = i + 1
		else:
			del population[j]
			j = j + 1
			l = l + 1
	del population[-((N*x_juv)-(N+l)):]
	print("pop size after initial reduction ", len(population))
	#determining the phenotypic value population parameter values for the adult
base population
	mean = np.mean(population, axis=0)
	sd = np.std(population, axis=0)
	generations = 0
#####INFERENCE OF ALLELE FREQUENCIES FOR QUANTIATIVE TRAIT
	for loc in range(0,nloc):
		Allele = 0
		for ind in range(0, len(population)):
			if str(population[ind][2* loc + 1]) == str(phenoval[loc][0]):
				Allele = Allele + 1
			if str(population[ind][2* loc + 2]) == str(phenoval[loc][0]):
				Allele = Allele + 1
		allelefreq[loc].append(Allele / (len(population)*2))
#####INFERENCE OF ALLELE FREQUENCIES FOR NEUTRAL MARKERS
	for loc in range(0,neutl):
		Allele = 0
		for ind in range(0, len(population)):
			if str(population[ind][(2* (nloc + loc)) + 2]) == "1":
				Allele = Allele + 1
			if str(population[ind][(2* (nloc + loc)) + 3]) == "1":
				Allele = Allele + 1
		neutrallfreq[loc].append(Allele / (len(population)*2))
	#thus, the initial base population is ready to roll
	current_mean = mean[nloc*2 + 1]
```

```
	initial_mean = mean[nloc*2 + 1]
	#setting the new fitness optimum
	newfit = (dev * sd[nloc*2 + 1]) + current_mean
#####OUTPUT INITIAL CONDITIONS TO FILE
	outf.write("replicate\t generation\t optimum\tphenotypic
mean\ts.d.\tallelefreqs\n")
	outf.write(str(rep) + "\t" + str(generations) + "\t" + str(mean[nloc*2 + 1])
+ "\t" + str(mean[nloc*2 + 1]) + "\t" + str(sd[nloc*2 + 1]) + "\t")
	for loc in range(0,nloc):
		outf.write(str(allelefreq[loc][k]) + "\t")
	outf.write("\t")
	for loc in range(0,neutl):
		outf.write(str(neutrallfreq[loc][k]) + "\t")
	outf.write("\n")
	initialquant = copy.deepcopy(allelefreq)
	initialneutr = copy.deepcopy(neutrallfreq)
	print("new fitness optimum", newfit)
	print("Mean:   " + str(mean[2 * nloc + 1]) + "  s.d.  " + str(sd[2 * nloc +
1]))
######DIRECTIONAL SELECTION MODULE
######REPRODUCTION
	#until the actual population mean is equal to or larger than the new fitness
optimum, repeat the process generation for generation
	k = 1
	while current_mean < newfit:
		generations = generations + 1
		previous_mean = current_mean
		print("generation: ", generations)
		#mate the individuals randomly
		#for this, copy the current population to a new array
		adults = copy.deepcopy(population)
		pair = 0
		no_pairs = int(N/2)
		#empty the current population
		population = [[] for _ in range(2*no_pairs*x_juv)]
		#pick randomly two individuals from the adults and delete them from
the list
		for pairs in range(0,no_pairs):
			rand1 = randint(0,len(adults)-1)
			indiv1 = adults[rand1]
			del adults[rand1]
			rand2 = randint(0,len(adults)-1)
			indiv2 = adults[rand2]
			del adults[rand2]
			#now they produce 2*x_juv offspring
			for ind in range (0 + (2* x_juv * pair), (2 * x_juv * pair) + 2
* x_juv):
				randID = randint(0,10*N*x_juv)
				population[ind].append(randID)
				#quantitative trait
				for loc in range(0,nloc):
					#random draw of gametes at each locus
					coin = randint(0,1)
					if coin == 0:
						population[ind].append(indiv1[2*loc + 1])
					else:
						population[ind].append(indiv1[2*loc + 2])
					coin = randint(0,1)
					if coin == 0:
						population[ind].append(indiv2[2*loc + 1])
					else:
						population[ind].append(indiv2[2*loc + 2])
				#calculate phenotype value
				phenotype_value = 0
				for loc in range(0,2*nloc):
					phenotype_value = phenotype_value +
((population[ind][loc + 1])/2)
					phenotype_value = phenotype_value +
gauss(0,initial_mean*phenoplast)
				population[ind].append(phenotype_value)
```

```
                        #neutral markers
                        for loc in range(nloc,nloc + neutl):
                              #random draw of gametes at each locus
                              coin = randint(0,1)
                              if coin == 0:
                                    population[ind].append(indiv1[2*loc + 2])
                              else:
                                    population[ind].append(indiv1[2*loc + 3])
                              coin = randint(0,1)
                              if coin == 0:
                                    population[ind].append(indiv2[2*loc + 2])
                              else:
                                    population[ind].append(indiv2[2*loc + 3])

      population[ind].append((math.sqrt(np.square(phenotype_value - newfit))))
                  pair = pair + 1
            #mix population by sorting by random index
            population.sort(key=lambda x: x[0])
####DETERMINISTIC SOFT SELECTION MODULE
            if selmode == "d":
                  #let deterministic selection do its beneficial work (only the
best survive and even there not all)
                  population.sort(key=lambda x: x[(2*(nloc+neutl))+2])
                  #Then, remove everything but the first N individuals
                  del population[-(len(population)-N):]
                  print("population reduced to adult size ", len(population))

            #record the result and proceed
            mean = np.mean(population, axis=0)
            sd = np.std(population, axis=0)
            current_mean = mean[2 * nloc + 1]

            #check for adaptation milestones
            if (1 - (newfit - mean[2 * nloc + 1]) / (newfit - initial_mean)) >=
0.5:
                  milest50 = milest50 + 1
                  if milest50 == 1:
                        adaptime50.append(generations)
            if (1 - (newfit - mean[2 * nloc + 1]) / (newfit - initial_mean)) >=
0.95:
                  milest95 = milest95 + 1
                  if milest95 == 1:
                        adaptime95.append(generations)

            print("Mean:   " + str(mean[2 * nloc + 1]) + "  s.d.  " + str(sd[2 *
nloc + 1]))
            print("deviation of population mean from fitness optimum ", newfit -
mean[2 * nloc + 1])
            print("proportion of adaptive walk: ",(1 - (newfit - mean[2 * nloc +
1]) / (newfit - initial_mean)))
#####INFERENCE OF ALLELE FREQUENCIES FOR QUANTITATIVE TRAIT
            for loc in range(0,nloc):
                  Allele = 0
                  for ind in range(0, len(population)):
                        if str(population[ind][2* loc + 1]) ==
str(phenoval[loc][0]):
                              Allele = Allele + 1
                        if str(population[ind][2* loc + 2]) ==
str(phenoval[loc][0]):
                              Allele = Allele + 1
                  allelefreq[loc].append(Allele / (len(population)*2))
#####INFERENCE OF ALLELE FREQUENCIES FOR NEUTRAL MARKERS
            for loc in range(0,neutl):
                  Allele = 0
                  for ind in range(0, len(population)):
                        if str(population[ind][(2* (nloc + loc)) + 2]) == "1":
                              Allele = Allele + 1
                        if str(population[ind][(2* (nloc + loc)) + 3]) == "1":
                              Allele = Allele + 1
                  neutrallfreq[loc].append(Allele / (len(population)*2))
```

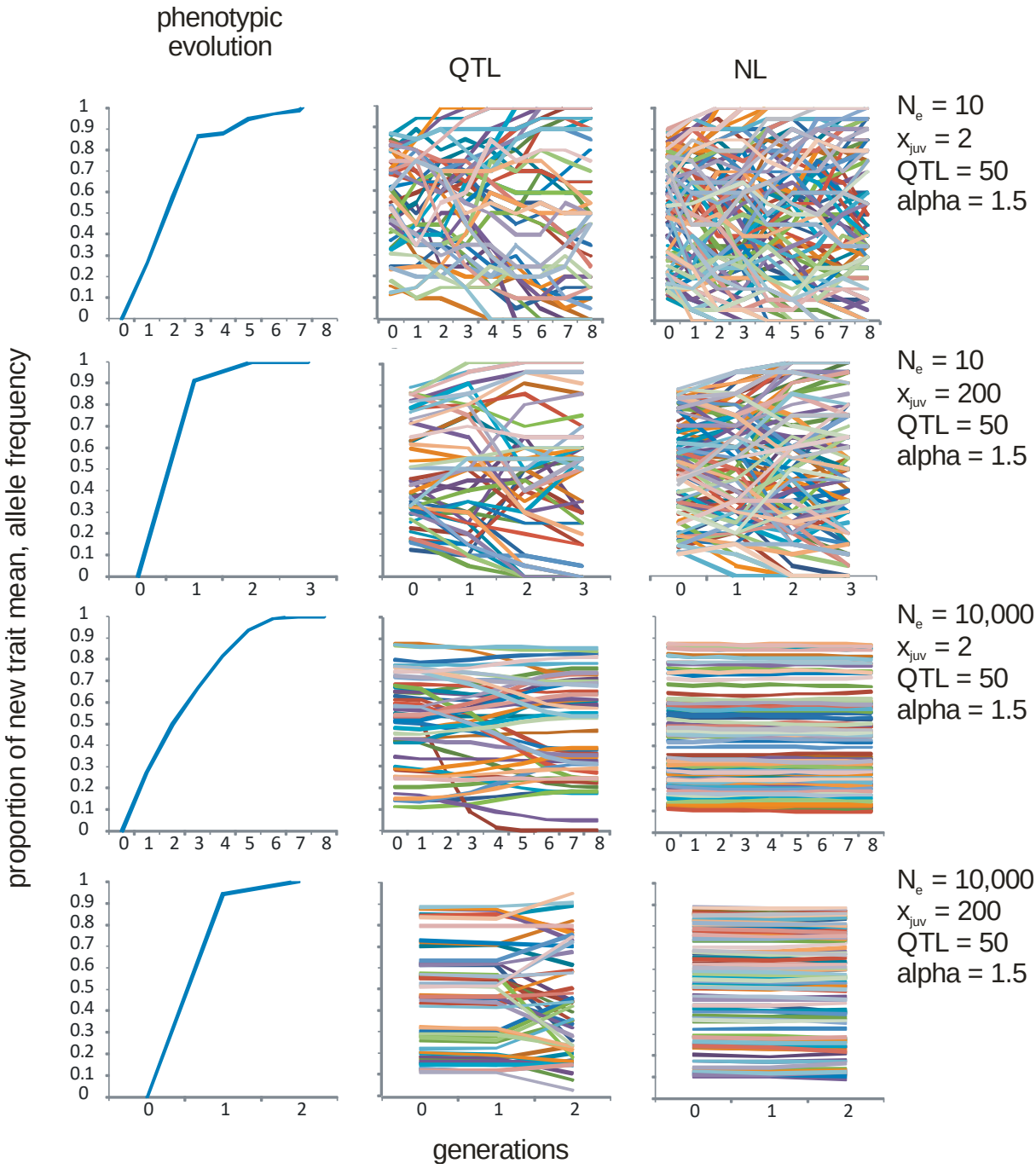
```
#####OUTPUT
              outf.write(str(rep) + "\t" + str(generations) + "\t" + str(newfit) +
"\t" + str(mean[nloc*2 + 1]) + "\t" + str(sd[nloc*2 + 1]) + "\t")
              for loc in range(0,nloc):
                     outf.write(str(allelefreq[loc][k]) + "\t")
              outf.write("\t")
              for loc in range(0,neutl):
                     outf.write(str(neutrallfreq[loc][k]) + "\t")
              outf.write("\n")
              #Check if adaptation takes too long
              if generations >= 100:
                     print("failed to adapt rapidly")
                     adapfail = adapfail + 1
                     break
              #Check if there is still adaptation possible
              if previous_mean == current_mean:
                     print("failed to adapt")
                     adapfail = adapfail + 1
                     break
              k = k + 1
#####INFERENCE OF ALLELE FREQUENCY CHANGES AND FIXED LOCI AMONG FIRST AND LAST
GENERATION
       outf.write("Allelefrequency changes\t\t\t\n\t\t\t\t\t")
       alleleselec = []
       alleledrift = []
       fixquant = 0
       for i in range(0,nloc):
              absdiff = (math.sqrt(np.square(initialquant[i][0] -
allelefreq[i][k])))
              if allelefreq[i][k] == 1.0:
                     fixquant = fixquant + 1
              elif allelefreq[i][k] == 0.0:
                     fixquant = fixquant + 1
              alleleselec.append(absdiff)
              outf.write(str(absdiff) + "\t")
       QTfix.append(fixquant)
       outf.write("\t")

       fixneut = 0
       for i in range(0,neutl):
              absdiff = (math.sqrt(np.square(initialneutr[i][0] -
neutrallfreq[i][k])))
              if neutrallfreq[i][k] == 1.0:
                     fixneut = fixneut + 1
              elif neutrallfreq[i][k] == 0.0:
                     fixneut = fixneut + 1
              alleledrift.append(absdiff)
              outf.write(str(absdiff) + "\t")
       neutfix.append(fixneut)
       outf.write("\n")
       outf.write("fixed QT loci: \t" + str(fixquant) + "\tfixed neutral loci\t" +
str(fixneut) + "\n")

       outf.write("Phenotype values of alleles \t variance contribution for each
locus (i.e. difference between alleles\n")
       phenodiff = []
       deltasumpheno = 0
       for loc in range(0,nloc):
              phenostart = (phenoval[loc][0]* initialquant[loc][0]) +
(phenoval[loc][1] * (1 - initialquant[loc][0]))
              phenoend = (phenoval[loc][0]* allelefreq[loc][k]) + (phenoval[loc][1]
* (1 - allelefreq[loc][k]))
              delta = phenoend - phenostart
              phenodiff.append(delta)
              deltasumpheno = deltasumpheno + delta
       signloci = 0
       propexp = 0
       for loc in range(0,nloc):
              outf.write("locus " + str(loc) + "\t" + str(phenoval[loc][0]) + "\t" +
str(phenoval[loc][1]) + "\t" + str(math.sqrt(np.square(phenoval[loc][0] -
```

```
phenoval[loc][1]))) +"\t" +  str(initialquant[loc][0]) + "\t" +
str(allelefreq[loc][0])+ "\t" + str(phenodiff[loc]/deltasumpheno) + "\t")
              if alleleselec[loc] >= np.percentile(alleledrift,99):
                     outf.write("* \n")
                     signloci = signloci + 1
                     propexp = propexp + phenodiff[loc]/deltasumpheno
              else:
                     outf.write("- \n")
       detectloc.append(signloci)
       phenoexpl.append(propexp)
       outf.write("\n")
       selectmean.append(np.mean(alleleselec))
       selectmax.append(np.amax(alleleselec))
       selectmin.append(np.amin(alleleselec))
       driftmean.append(np.mean(alleledrift))
       driftmax.append(np.amax(alleledrift))
       driftmin.append(np.amin(alleledrift))
       drift99.append(np.percentile(alleledrift,99))
       corr.append(pearsonr(phenodiff,alleleselec))
       adaptime.append(generations)
#####END OF REPLICATES, WRITE SUMMARY RESULTS TO OUTFILE
outf.write("mean shift selected loci: \t" + str(np.mean(selectmean)) + "\t s.d.\t"
+ str(np.std(selectmean))  + "\n")
outf.write("max shift selected loci: \t" + str(np.mean(selectmax)) + "\t s.d.\t" +
str(np.std(selectmax))  + "\n")
outf.write("min shift selected loci: \t" + str(np.mean(selectmin)) + "\t s.d.\t" +
str(np.std(selectmin))  + "\n")
outf.write("mean shift drifted loci: \t" + str(np.mean(driftmean)) + "\t s.d.\t" +
str(np.std(driftmean))  + "\n")
outf.write("max shift drifted loci: \t" + str(np.mean(driftmax)) + "\t s.d.\t" +
str(np.std(driftmax))  + "\n")
outf.write("min shift drifted loci: \t" + str(np.mean(driftmin)) + "\t s.d.\t" +
str(np.std(driftmin)) + "\n")
outf.write("99% detection treshold: \t" + str(np.mean(drift99)) + "\t s.d.\t" +
str(np.std(drift99)) + "\n")
outf.write("mean number of detectable loci: \t" + str(np.mean(detectloc)) + "\t
s.d.\t" + str(np.std(detectloc)) + "\n")
outf.write("mean proportion of phenotypic difference accounted for: \t" +
str(np.mean(phenoexpl)) + "\t s.d.\t" + str(np.std(phenoexpl)) + "\n")
outf.write("mean no. fixed QT loci: \t" + str(np.mean(QTfix)) + "\t s.d.\t" +
str(np.std(QTfix)) + "\n")
outf.write("mean no. fixed neutral loci: \t" + str(np.mean(neutfix)) + "\t s.d.\t"
+ str(np.std(neutfix)) + "\n")
outf.write("mean Pearson correlation coeff. between locus phenotypic effect and
allelefreq change: " + str(np.mean(corr, axis=0)) + "\t s.d.\t" + str(np.std(corr,
axis=0)) + "\n")
outf.write("mean no. generations to adaptation: \t" + str(np.mean(adaptime)) + "\t
s.d.\t" + str(np.std(adaptime)) + "\n")
outf.write("mean no. generations to 50% adaptation: \t" + str(np.mean(adaptime50))
+ "\t s.d.\t" + str(np.std(adaptime50)) + "\n")
outf.write("mean no. generations to 95% adaptation: \t" + str(np.mean(adaptime95))
+ "\t s.d.\t" + str(np.std(adaptime95)) + "\n")
outf.write("times adapation failed: \t" + str(adapfail) + "\n")
outf.write("raw data \n")
outf.write(str(selectmean)+ "\n")
outf.write(str(driftmean)+ "\n")
outf.write(str(drift99)+ "\n")
outf.write(str(detectloc)+ "\n")
outf.write(str(phenoexpl)+ "\n")
outf.write(str(QTfix)+ "\n")
outf.write(str(neutfix)+ "\n")
outf.write(str(corr)+ "\n")
outf.write(str(adaptime)+ "\n")
outf.close()
```

Supplemental Figure 1. Examples for phenotype evolution and allele-frequency shifts in QTL and NL for selected factor combinations. The left column shows the course of phenotype evolution, the middle column the QTL, in the right the NL allele frequency changes through time.



Supplemental Figure 2. Illustration of genetic bet hedging by phenospace filling. Given are the phenotype value distributions for the offspring of five randomly chosen breeding pairs with 2 (above), 20 (middle panel) and 200 (below) offspring per parent. While the phenotype distribution for a small number of offspring is more or less random, the distribution is smooth and covers a wide phenotypic range for a large number of offspring. The phenotypic trait values are completely additive values from the contribution of 30 quantitative trait loci.