QuantumGEP v1 Manual

Manual Version: May 12, 2022

Oak Ridge, 2018

Gonzalo Alvarez
Nanomaterials Theory Institute
Oak Ridge National
Laboratory

Oak Ridge, TN 37831
May 12, 2022

Contents

1 Preliminaries **5**
- 1.1 Disclaimer and Licensing . . . 5
  - 1.1.1 Please cite this work . . . 5
  - 1.1.2 References . . . 5
- 1.2 Verifying, Building and Running . . . 6
  - 1.2.1 Hash of the latest commit . . . 6
  - 1.2.2 Building and Running evendim . . . 6

2 Arithmetic Example **9**
- 2.1 Finding a Function from Training . . . 10
- 2.2 Multiple Variables . . . 10
- 2.3 Computational Engine Overview . . . 10

3 QuantumGEP **11**
- 3.1 Description of the Problem . . . 11
- 3.2 QuantumGEP for Ground State . . . 11
- 3.3 Input File Details . . . 11

4 Evendim **13**
- 4.1 Procedural Description . . . 13
- 4.2 Mutations, Recombinations and Swaps . . . 13
- 4.3 Interfaces to Primitives . . . 14
- 4.4 Interfaces to Fitness . . . 14
- 4.5 The TestSuite . . . 15

Chapter 1

Preliminaries

1.1 Disclaimer and Licensing

1.1.1 Please cite this work

EVENDIM is a free and open source computational engine for gene expression programming. The full software license for evendim version 0. can be found in file LICENSE. You are welcomed to use it and publish data obtained with evendim. If you do, please cite this work. Explain How To Cite This Work. FIXME. TBW.

1.1.2 References

```
@book{ferreira2006gene,
title={Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence},
author={Ferreira, C.},
isbn={9783540328490},
series={Studies in Computational Intelligence},
url={https://books.google.com/books?id=NkG7BQAAQBAJ},
year={2006},
publisher={Springer Berlin Heidelberg}
}
```

1.2 Verifying, Building and Running

1.2.1 Hash of the latest commit

Hash of the latest commit is also posted at FIXME

1.2.2 Building and Running evendim

EVENDIM is a computational engine for gene expression programming. There are many examples to use EVENDIM. To specify an example we need to give the primitives (or operators) and the inputs or leaves, as well as the training function. A simple example will be discussed first where the primitives are the arithmetic primitives in $\{+, -, *, /\}$, and the training function will be $f(x) = x^3 - x$.

Required Software

1. GNU C++ or LLVM clang++. Other C++ compilers may also work but EVENDIM was only tested on these two.
2. The LAPACK and BLAS libraries
3. The Gnu Scientific Library or GSL library
4. PsimagLite (see below)
5. make or gmake is optinal and only needed to use the Makefile
6. perl (may optionally be needed to run some auxiliary script)

Quick Start

1. Use your distribution repository tool to install gcc with support for C++ (or LLVM clang++), the LAPACK and BLAS libraries, the GSL library, make, perl, and git if you don't have them.
2. Issue

   ```
   cd someDirectory/

   git clone https://github.com/g1257/PsimagLite.git

   git clone https://github.com/g1257/evendim.git
   ```

3. Compile PsimagLite with `cd PsimagLite/lib; ./configure.pl; make -j 4`

4. Now issue

   ```
   cd evendim/src

   cp Config.make.sample Config.make

   make
   ```

5. You can run the arithmetic primitives with `./gep2 -i 1 -h 5 -p 100 -t 10` which will run an arithmetic test with one input, head size of a maximum of 5, population 100 individuals, and for t=10 generations. The function is in src/Functions/Example1.h and is $f(x) = x * (x - 1) * (x + 1)$.

Chapter 2

Arithmetic Example

This driver program named gep2 runs different "example" cases consisting of using GEP to find a function knowning only some inputs and outputs.

The primitives are under Primitives/PlusMinusMultiplyDivide.h, and the functions under Example1.h, Example2.h and Example3.h

The following command line arguments to gep2 are mandatory.

- -i inputs. The number of inputs to the function.
- -h headSize. The maximum number of the head or effective gene size.
- -p population. The number of GEP individuals to consider in each generation.
- -t generations. The number of generations to run GEP.

The following command line arguments to gep2 are optional.

- -e example. The example number to run: 1, 2 or 3. Defaults to 1.
- -g genes. The number of genes to be used. Defaults to 1.
- -s seed. The seed for the random number generator. Defaults to 1234.
- -c constants. The number of GEP constants to use. Default to 0.
- -H maximum head size for ADF. ADF stands for automatic defined funtions. Defaults to 0.
- -a adfs. The number of ADFs to use. Defaults to 0.
- -n samples. The number of training samples to cache. Defaults to 100.
- -v indicates that GEP should be verbose. Defaults to false.
- -S indicates that GEP should stop when a perfect individual is found. Defaults to false.

The options -v and -S take no arguments.

2.1 Finding a Function from Training

You can run the arithmetic primitives with `./gep2 -i 1 -h 5 -p 100 -t 10` which will run an arithmetic test with one input, head size of a maximum of 5, population 100 individuals, and for t=10 generations. The function is in src/Functions/Example1.h and is $f(x) = x * (x - 1) * (x + 1)$.

2.2 Multiple Variables

Example3Fitness illustrates the case of a training function with many variables and consists of a function f(x0, x1, ..., x5) of six variables. The variables are in the space of valid alphanumeric characters. If x0 is a digit then the function returns that digit plus one. If not, but if x1 is a digit then the function returns that digit plus one. And so on until all arguments to f are evaluated. If none of them are digits, then the function returns -1.

2.3 Computational Engine Overview

The main loop in gep2 is

```
// total = number of generations
for (SizeType i = 0; i < total; ++i)
  engine.evolve(i);
```

The `Engine` class is templated on a `Fitness` template that represents the training class, and determines how fit a GEP individual is. It is also templated on the Evolution type. The `Engine` constructor takes an input parameters object, and an evolution object. Evolution is templated on Primitives, which represents the GEP primitives or "operators" to be considered. Evolution's constructor takes a primitives object, a seed, and an verbose boolean. In this file, gep2.cpp, Primitives is set to the class PlusMinusMultiplyDivide so that the primitives are plus, minus, multiply and divide.

Engine::evolve() function starts by considering all parent chromosomes. It then computes the fitness of these parent chromosomes. It then applies one-point recombination, two-point recombination, mutation, inversion, and swap algorithms to all parent chromosomes to generate the descendants for this generation. It then canonicalizes them and selects the best p chromosomes and discards the ones with lowest fitness, where p is the population number set from the input file or the command line.

Chapter 3

QuantumGEP

3.1 Description of the Problem

This driver program named quantumGep uses GEP to find a quantum circuit. There are two usages: (i) the quantum circuit to be found implements a function known only by some of its input and outputs, and (ii) the quantum circuit to be found yields the ground state of a known Hamiltonian when applied to an initial quantum state. The primitives are under Primitives/QuantumCircuit.h, and consist of one-bit and two-bit gates.

quantumGep takes one mandatory argument: -f filename, with the name of the input file. It takes the following optional arguments.

-S threads. The number of threads for shared memory parallelization.

-p precision. The precision for printing numbers.

-v indicates that quantumGep be verbose.

3.2 QuantumGEP for Ground State

When Runtype="GroundState" in the input file, quantumGEP finds circuits that, when applied to the initial state, yield the ground state of a chosen Hamiltonian. Class GroundStateFitness implements the fitness for this case. The fitness of an individual equals minus the value of $\langle v|H|v\rangle$, where $|v\rangle$ is the vector produced by the individual (that is, the quantum circuit) when applied to the initial state, and H is the Hamiltonian.

3.3 Input File Details

The input file contains parameters of the form name=value; where a semicolon must be included at the end. Moreover, the first line of the input file must start with

```
##Ainur1.0
```

Input parameters can be divided in engine parameters, which are those that deal with the GEP algorithm itself regardless of the fitness function used. Then there are fitness parameters, which for quantumGEP will include those regarding the Hamiltonian for RunType="GroundState" and also the minimization parameters. We list and describe them in turn in what follows.

The engine parameters can be specified with `Generations=100;` in the input file, and similarly for the others, which are as follows.

Generations	The number of GEP generations. Integer. Mandatory.
Population	The number of GEP individuals. Integer. Mandatory.
HeadSize	The size of the head (that is, the maximum effective gene size). Integer. Mandatory.
Samples	The samples to be cached. Optional. Defaults to 50 and is unused in quantumGEP.
Threads	The number of shared memory threads to use. Optional. Defaults to 1. Not all fitness classes support paralellization, that is, a number greater than one here.
Primitives	A comma-separated list of quantum gates to consider by GEP. String. Optional. Defaults to "C,H,P".
EngineOptions	A comma-separated list of options. String. Optional. Default to the empty string.

The EngineOptions are case-insensitive and can be none or more of the following.

stopEarly	Stops quantumGEP as soon as a perfect individual (that is, circuit) is found.
noncanonical	Disables the canonicalization step.
progressBar	Prints a progress bar for each generation.
printCompact	Prints individuals in compact form.

Chapter 4

Evendim

4.1 Procedural Description

The engine constructor creates the initial individuals randomly. Engine::evolve() function starts by considering all parent chromosomes. It then computes the fitness of these parent chromosomes. It then applies one-point recombination, two-point recombination, mutation, inversion, and swap algorithms to all parent chromosomes to generate the descendants for this generation. It then canonicalizes them and selects the best p chromosomes and discards the ones with lowest fitness, where p is the population number set from the input file or the command line.

4.2 Mutations, Recombinations and Swaps

The evolve function in the computational engine starts by creating new individuals, in its first call from the initial population, and in subsequent calls from the surviving individuals. It does so by using the following four algorithms in succession: (1) one-point recombination, (2) two-point recombination, (3) mutation, (4) inversion, and (4) swap; all these algorithms were implemented as detailed in [?], and we briefly review them in the following.

Recombination involves two parent chromosomes and results in two new individuals. One-point recombination consists of paring the parent chromosomes side by side, choosing a random point at which the parent chromosomes are split up, and exchanging the genetic content after the recombination point between the two chromosomes. Two-point recombination pairs the chromosomes side by side as before, chooses two random points, and exchanges the genetic material between these two points, creating two new individuals. A mutation changes one character of the string representation of the chromosome; in the head any character can change to any other, so any function can be changed to any other without regards to the number of arguments. In the tail, terminal or leafs are changed only into terminal or leafs so that the head and tail structure of the chromosome is preserved by the mutation. Inversion involves inverting the characters in the head of the chromosome, and does not affect the tail. EVENDIM inverts the complete head even though subsets of the head could be inverted also. Finally, a swap exchanges two characters in the string representation of a chromosome such that the head and tail structure is preserved.

After the new population has been created, which also includes the surviving individuals from the previous generation, an optional canonicalization procedure is applied. For quantum circuits, the canonicalization orders the gates by the bit it acts on; there is also here an opportunity for symbolic simplifications: for example, the Pauli matrix gate σ^z if applied on the same bit twice yields the identity. More complicated simplifications could be added here as well, based on commutation rules among operators or gates. Finally, QuantumGEP sort the individuals by fitness, its definition depending on the problem to be solved, and discards as many individuals with lowest fitness as needed to obtain the population supplied in the input file.

4.3 Interfaces to Primitives

This is the interface for Primitives, that is, these are the functions that a programmer needs to write to implement a new Primitives class to use with EVENDIM.

```
virtual const VectorNodeType& nodes() const = 0;
```

Returns a vector of nodes containing all different nodes.

```
virtual const VectorValueType& dcValues() const = 0;
```

Returns a vector of defined constant values

```
virtual const VectorStringType& dcArray() const = 0;
```

Returns a vector of defined constant names

4.4 Interfaces to Fitness

The BaseFitness class provides an interface that fitness classes must follow. It does also provide some basic non-virtual functionality.

```
virtual RealType getFitness(const ChromosomeType& chromosome,
                            long unsigned int seed,
                         SizeType threadNum) = 0;
```

Returns the fitness of the passed chromosome, where seed is a seed for a random number generator, and threadNum is the number of the thread in case fitness is computed in parallel.

```
virtual RealType maxFitness() const = 0;
```

Returns the maximum fitness for this problem.

4.5 The TestSuite