

# Galib Tutorial

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy  
s.benedettini@unibo.it

Artificial Intelligence 2011

Stefano  
Benedettini

## Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

## What is Galib?

- Efficient C++ library for constraint programming

## Galib Advantages

- Reasonable number of features
- Easy to get started
- Customizable with little effort 95% of the times

## Galib Disadvantages

- It's old-style C++!
- Documentation is incomplete
- A pain to customize the other 5% of the times
- Lacks advanced features
  - Multi-objective optimization
  - Memetic algorithm support (can be “faked” though)
  - Genetic programming support

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- [Galib](#) home page
- [Galib API documentation](#)
- [Galib class hierarchy](#)
- [Commented examples](#)

- To many to enumerate!
  - Hundreds of programs, libraries and tools
    - A quick search of “genetic algorithm” on [sourceforge](#) yields about 300 results
  - Academic or commercial projects
    - Small-scale to big industrial efforts
  - Disparate features
  - Implementations in every mainstream language

## Honorable Mentions

**Evolving Objects:** solid C++ framework for evolutionary computation.

**ParadisEO:** a C++ framework, integrating EO, for hybrid parallel metaheuristics.

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

## Foreword

- Tested configuration:
  - Ubuntu 10.04
  - GCC 4.4.3
- Latest version (2.4.7) of Galib is required
  - Download it [here](#)

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

**Genome (also Chromosome or Individual):** representation of a solution of a problem

**Gene:** a component of a chromosome

**Allele:** a value of a gene

**Population:** a collection of individuals

**Objective Function:** problem specific function that evaluate a solution

**Fitness Function:** a scoring function that determines the “goodness” of an individual

**Recombination Operators:** operations that alter a chromosome or produce new offspring

**Crossover:** mating operator that spawns new chromosomes

**Mutation:** modification operator to a single chromosome

# GAs from a Software Engineer Perspective

## Characteristics

- Simple structure
- Very well understood
- Easy to implement
- Genericity
  - Problem independent algorithm
  - Generic as in generic programming

## GAs Versatility

- Many aspects of a genetic algorithm are:
  - Customizable
  - Independent of genome representation
- We can say GAs are made of reusable plug-in components
  - Each components must respect a suitable interface
  - Strategy Pattern

# GAs from a Software Engineer Perspective

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:   evaluation(Offspring)
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Genetic algorithms partly depend on genome representation
- A genome is an instance of a particular type

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- A problem specific way to initialize a population

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Termination condition is orthogonal to other concepts
- Many options available

# GAs from a Software Engineer Perspective

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Individual are selected for reproduction
- Usually depends only on individual fitness

# GAs from a Software Engineer Perspective

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Crossover and mutation are obviously genome dependent
- Notice that genome representation *can be* problem independent
  - Think of continuous optimization problems

# GAs from a Software Engineer Perspective

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:   evaluation(Offspring)
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Evaluation is dependent on both the genome type and the problem

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Population update usually depends only on fitness
- But again, many choices are available

# GAs from a Software Engineer Perspective

## Genetic Algorithm Structure

```
1:  $P \leftarrow \text{generateInitialPopulation}()$ 
2: while termination conditions not met do
3:    $S \leftarrow \text{selection}(P)$ 
4:   Offspring  $\leftarrow \text{recombination}(S)$ 
5:    $\text{evaluation}(\text{Offspring})$ 
6:    $P \leftarrow \text{update}(P, \text{Offspring})$ 
7: end while
8: return  $P$ 
```

## Comments

- Population update usually depends only on fitness
- But again, many choices are available

GAs simple and modular architecture enables several opportunities for optimizations and variations.

- Multiple populations
  - Simulates niching
  - Exchange of individuals among populations
- Parallel computation
  - Individual evaluation can be performed in parallel
  - Multiple populations on multiple cores/machines
- Hybridization (memetic)
  - Adds a further local search step
  - Before or after population update

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

## Problem

- Suppose I have an algorithm
- How can I customize its behaviour without:
  - rewriting it from scratch
  - copying/pasting any code

## (Object Oriented) Solution

Step One: characterize those parts that need to be made generic

- Identify **modification points**
- Define a suitable interface for them

Step Two: design a class that provides the desired behaviour

Sorting algorithm: depends on a *comparison function*

Filtering: depends on filter definition, but usually a function (uniform, Gaussian)

Integration: depends on *integrand function*

Database queries: `select` clause applies a function to tuples, `group by` groups according a criterion

List filtering: depends on a predicate

- Each algorithm delegates to a function part of its behaviour
- Functions are the key!

```
struct F2 { // Abstraction of  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$   
    virtual double operator()(double, double) = 0;  
};  
double foldl(vector<double>& v, double accumulator,  
            BinaryOperation* f) {  
    for(int i = 0; i < v.size(); ++i)  
        accumulator = (*f)(accumulator, v[i]);  
    return accumulator;  
}  
  
struct Sum : F2 { double operator()(double acc,  
    double elem) { return acc + elem; } };  
struct Prod : F2 { double operator()(double acc,  
    double elem) { return acc * elem; } };  
struct Max : F2 { double operator()(double acc,  
    double elem) { return max(acc, elem); } };  
  
foldl(someVector, 0, new Sum()); // sums the elements  
    in a vector  
  
foldl(someVector, 1, new Prod()); // multiplies the  
    elements in a vector  
  
foldl(someVector, numeric_limits<double>::min(), new  
    Max()); // return the maximum element
```

- F2 is the type of all “things” that get a pair of doubles and return a double
  - Notice that F2 is an abstract type
  - Objects of type F2 are always passed by pointer or reference
- In C++ a functions can be one those “things”

```
double my_sum(double x, double y) {  
    return x + y;  
}
```

- In C++ functions *have* a type
- You can pass (pointer to) functions around as you do with objects

## Function Pointer Types

- Look like a function signature with a star before its name
- Function argument type list goes *after* its name

```
int (*varName)(int); // varName is a pointer to a  
                      function  $\mathbb{Z} \rightarrow \mathbb{Z}$   
double (*f)(double, double); f :  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ 
```

- Use a typedef

```
typedef int (*PtrFunIntInt)(int);  
typedef double (*F)(double, double);  
PtrFunIntInt varName; // same as before  
double foldl(vector<double>& v, double x0, F f);
```

## Using Function Pointers

- Use operator `&` to get a pointer of a function

```
foldl(someVector, 0.0, &my_sum);
```

- Before applying a function pointer, remember to dereference it

```
int (*pf)(int, int) = &my_sum;  
int result = (*pf)(10.0, 20.0);
```

## Pointers to Static Member Functions

- Static methods have the same type as free functions
- Mind that *non-static* method have a whole different type!

```
struct S {  
    static int staticMethod(int);  
    int instanceMethod(int);  
}  
int (*pf)(int) = &S::staticMethod;  
// pf = &S::instanceMethod; // doesn't work
```

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

- Two main [modules](#):
  - 1 Genetic algorithm solvers
    - Several algorithm variants implemented (look them up [here](#))
    - Collection of [statistics](#)
    - [Termination conditions](#)
    - Support for [parameter](#) parsing
  - 2 Data structures for genomes
    - 1D/2D/3D and other linear genomes (from [here on](#))
    - [Tree](#) genomes
    - Several genome-specific recombination operators
- Function types for modification points defined [here](#)

## Purpose of a Genome Object

- It represents a solution
- It serves as template for other genomes

## Genome Interface

- Documented [here](#)
- Pointer functions used to specify:
  - Evaluator:** calculates objective function
  - Initializer:** called whenever a new genome is generated
  - Mutator and Crossover:** preferred recombination operators
  - Comparator:** returns the distance value between two genomes

- Control algorithm execution:
  - Which individuals can reproduce
    - Selection schemes are documented [here](#)
  - Which individuals are replaced
    - Only for incremental GAs
  - Frequency of mutation and crossover operators application
  - When to terminate algorithm execution
  - Whether to minimize or maximize the fitness function
  - How to scale the objective function
    - Bundled scaling functions are documented [here](#)
- Computes a number of statistics:
  - Indicators of fitness value distribution
  - Population *convergence*
  - Population *diversity* (is a suitable genome comparator was specified)

- ① Simple Genetic Algorithm ([GASimpleGA](#)):
  - Fixed-size population
  - Offspring replace the whole population
  - Optional elitism (the best individual is carried over across generations)
- ② Steady-state Genetic Algorithm ([GASteadyStateGA](#)):
  - Fixed-size population
  - Partial “overlap” between offspring and current population
    - Overlap percentage  $r$  is tunable
    - Only the best  $popSize \cdot r$  offspring carry over
- ③ Incremental Genetic Algorithm ([GAIncrementalGA](#)):
  - Fixed-size population
  - Only 2 children are generated per iteration
  - Replacement scheme is customizable

- Recombination operators are genome-specific
  - Each genome type defines its own operators
- You rarely have to define custom operators
  - Unless you define your own genome type
  - Unless you come up with something particularly clever for your problem
- Default ones are good enough
  - A graphical depiction of their behaviour is shown [here](#)

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- 1 Introduction
- 2 Genetic Algorithm as a Software Framework
- 3 C++ Interlude: Function Pointers
- 4 Programming with Galib
  - Galib Overview
  - Use Cases

Introduction

Genetic  
Algorithm as a  
Software  
FrameworkC++ Interlude:  
Function  
PointersProgramming  
with GalibGalib Overview  
Use Cases

- Simple bitstring genome
- Evaluation function is almost algorithm independent
- Helpful parameter management
- Notice how algorithm implementation can be easily changed

- An `AlleleSet<T>` represents either an enumerable set or an interval of values of type `T`
- `AlleleSets` might make your code simpler and more intuitive
- Use `AlleleSets` when your genes are an enumeration type
  - Characters
  - Subset or intervals of integers
- `AlleleSets` are also useful for real genes
  - Intervals of real numbers
  - Enumerable subset of reals, like  $\{0.1, 0.2, \dots, 0.9, 1\}$
- For intervals, provide upper and lower bounds
- `GANDArrayAlleleGenome<T>` combines a  $N$  dimensional genome array whose genes take value in an `AlleleSet<T>`

# Custom Termination Condition: Timeout

- A function cannot carry state between invocations
- A global timer must be defined

## A Word on Algorithm Comparison

- Comparing algorithm is not a trivial task
- Usually, given a fixed amount of “computational resources”, the quality of solution is evaluated
- Computational resources can be:
  - Runtime
    - Experimental setting must be the same among competing algorithms
  - Number of objective function evaluations
    - Makes sense only if function evaluation is the most expensive operation

# Custom Termination Condition: Timeout

- A function cannot carry state between invocations
- A global timer must be defined

## A Word on Algorithm Comparison

- Comparing algorithm is not a trivial task
- Usually, given a fixed amount of “computational resources”, the quality of solution is evaluated
- Computational resources can be:
  - Runtime
    - Experimental setting must be the same among competing algorithms
  - Number of objective function evaluations
    - Makes sense only if function evaluation is the most expensive operation

# How to Compose Termination Conditions

## Function Pointer Limitations

- Functions are *not* first-class entities in C++
  - You can't write small unnamed functions (lambdas)
  - It is difficult to compose them

## Hack Your Way Through

- Specifically, in Galib you can “artificially” compose termination conditions
  - Write a custom terminator function
  - Inside its body invoke all the terminators you want to compose and save their result (a GABoolean)
  - Write and return Boolean expression involving all of the terminators' responses

# How to Compose Termination Conditions

## Function Pointer Limitations

- Functions are *not* first-class entities in C++
  - You can't write small unnamed functions (lambdas)
  - It is difficult to compose them

## Hack Your Way Through

- Specifically, in Galib you can “artificially” compose termination conditions
  - Write a custom terminator function
  - Inside its body invoke all the terminators you want to compose and save their result (a GABoolean)
  - Write and return Boolean expression involving all of the terminators' responses

# How to Implement a Local Search Step

## Galib is not Meant for Memetics

- A local search step is not natively available
  - There is no modification point that lets you slip some LS code in

## Hack Your Way Through

- This problem can be circumvented:
  - Create a template population
  - Write a population evaluation function that invokes your local search
  - Set its evaluator to the custom evaluation function
- Remember to invoke the genome evaluation function!

# How to Implement a Local Search Step

## Galib is not Meant for Memetics

- A local search step is not natively available
  - There is no modification point that lets you slip some LS code in

## Hack Your Way Through

- This problem can be circumvented:
  - Create a template population
  - Write a population evaluation function that invokes your local search
  - Set its evaluator to the custom evaluation function
- Remember to invoke the genome evaluation function!

Stefano  
Benedettini

Introduction

Genetic  
Algorithm as a  
Software  
Framework

C++ Interlude:  
Function  
Pointers

Programming  
with Galib

Galib Overview  
Use Cases

- Recombination operators are partly undocumented
- Description of recombination algorithm is included in the source code
- Mind that sometimes functions return a `GABoollean` and not a `bool`

# Galib Tutorial

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy  
s.benedettini@unibo.it

Artificial Intelligence 2011