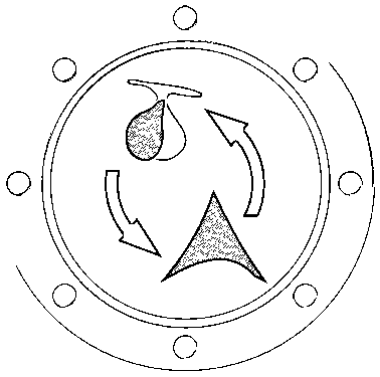
J. W. Dietrich, N. Siegmar

Evolutionary Algorithms

Details of Implementation



Contents

Introduction

SimulAdren's model is based on the MiMe-NoCoDI platform of nonlinear biological feedback loops [Dietrich and Boehm 2015; Dietrich et al. 2024]. It is implemented as a model with two consecutive MiMe saturation elements (Fig. 1).

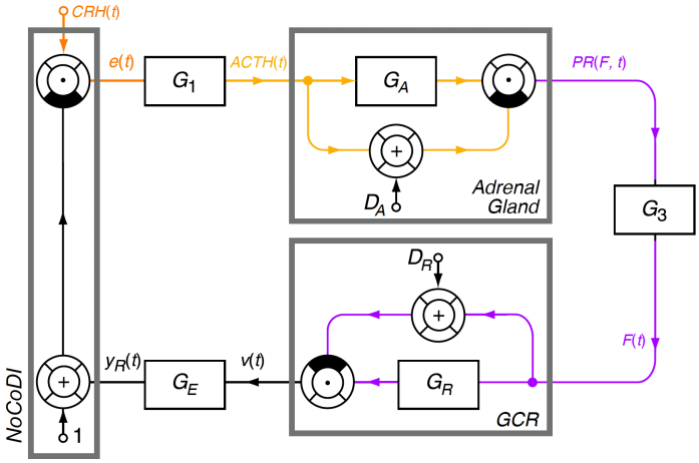


Figure 1: Processing structure of the model used by SimulAdren

The majority of the parameters used by SimulAdren is substantiated in scientific evidence [Dietrich and Boehm 2006], see Appendix A of SimulAdren's handbook for details. However, two parameters (G_E and G_R) are unobservable and have to be estimated. In early versions of the model [Dietrich et al. 2006], they were scaled as 1, but there remains room for variation. Therefore, a more neutral and methodical approach to the estimation of parameters has been implemented in the form of a genetic algorithm. Genetic algorithms are inspired by natural mutation and selection mechanisms (Fig. 2).

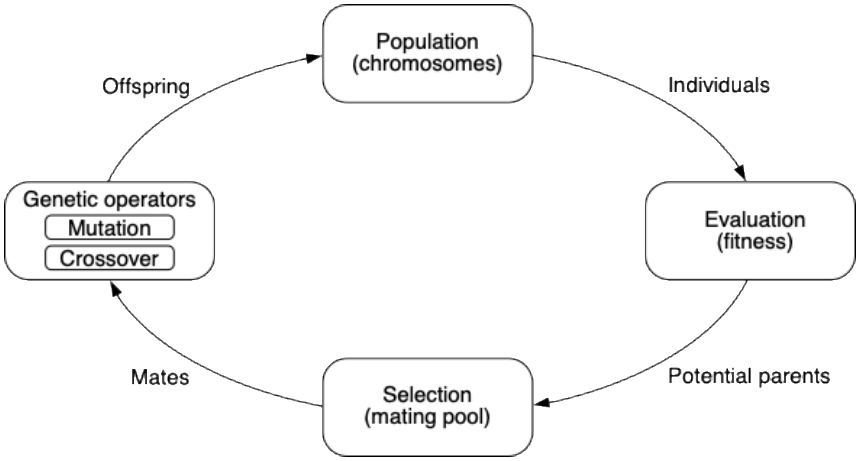


Figure 2: The cycle of a genetic algorithm
[changed from Ribeiro Filho and Treleaven 1994]

They provide useful optimisation techniques for large search spaces [Alhijawi and Awajan 2024; Ghosh and Mandal 2022; Katoch et al. 2021; Ribeiro Filho and Treleaven 1994].

The algorithms used by SimulAdren are subsequently presented and explained in detail.

The complete source code of the evolutionary engine implemented in SimulAdren is provided in order to foster understanding and enable reproducibility.

Unit RandomFunctions

This unit provides basic functionality for evolutionary algorithms.

Types

```
TIntArray = array of integer;
```

Global constants

```
kError100 = 'Runtime error: Parameter zero';
kError101 = 'Runtime error: Negative parameters';
kError102 = 'Runtime error: Parameter out of range';
kError103 = 'Runtime error: min > max or low > high';
```

Contains

```
function contains(const theArray: array of integer; theNumber:
         integer): boolean;
var
  i: integer;
begin
  Result := False;
  i := 0;
  repeat
    if theArray[i] = theNumber then
      Result := True;
    Inc(i)
  until (Result = True) or (i >= length(theArray) - 1);
end;
```

This function returns **true** if the integer `theNumber` is contained in the array `theArray`.

Sample

```
function Sample(const theArray: TIntArray; const theLength:
         integer): TIntArray;
var
  i, r: integer;
begin
  assert(theLength <> 0, kError100);
  assert(theLength >= 0, kError101);
  assert(theLength <= length(theArray), kError102);
  setLength(Result, theLength);
  for i := 0 to theLength - 1 do
  begin
    repeat
      r := random(length(theArray)) + 1;
    until not contains(Result, r);
    Result[i] := theArray[r - 1];
  end;
end;
```

This function draws a sample of length `theLength` from the vector `theArray` without replacement. The sample is returned as a vector, i. e. an array of integer.

runif

```
function runif(const min, max: real): real;
begin
  assert(max > min, kError103);
  result := min + random * (max - min);
end;
```

The `Random` function in Pascal delivers either a random number between 0 and 1 or, if a parameter **L** is passed, between 0 and **L**. The function `runif` is more flexible. It delivers a random number from a uniform distribution between **min** and **max**.

IncIndex

```
function IncIndex(const size: integer): TIntArray;
var
  i: integer;
begin
  assert(size <> 0, kError100);
  assert(size >= 0, kError101);
  SetLength(Result, size);
  for i := 0 to size - 1 do
    Result[i] := i;
end;
```

This is a simple helper function that returns an ordered array of integer.

Unit evoEngoine

This unit implements the genetic algorithm used by SimulAdren.

Types

```
TIndividual = record
    GR, GE: extended;
    fitness: real;
end;

TPopulation = array of TIndividual;
TParents = array[0..1] of TIndividual;
TChildren = array[0..1] of TIndividual;
TFittest = array of TIndividual;
TAllPopulations = array of TPopulation;
TAllele = array[0..1] of real;
```

A population of type `TPopulation` is an array (ordered set) of candidate solutions. Each individual (chromosome, candidate solution) consists of the two "genes" GE and GR (representing the corresponding gains of the feedback loop) with their alleles and the property of a fitness encoded as a real number (Fig. 3).

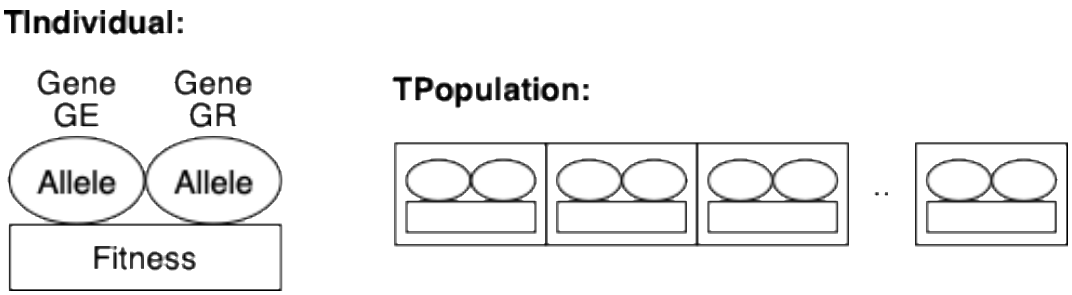


Figure 3: Structure of the types `TIndividual` and `TPopulation`

Global constants

```
LowerBound = 0;
UpperBound = 100;
PopulationSize = 50;
Generations = 50;
MutationRate = 0.1;
TournamentSize = 3;
```

Fitness

```
function Fitness(const CRH: extended; const params: TParams;
         const theGuess: TIndividual; const EvoTargets:
         TEvoTargets): real;
var
  i: integer; // index for steady-state solution to use
  distanceA, distanceF, distance: real; // distances from target
  steadyState: TPredictionArray; // steady-state solutions
  testParams: TParams; // parameter set for feedback loop
begin
  testParams := params; // passed parameters for feedback loop
  if isNan(testParams.GE) then
    testParams.GE := theGuess.GE;
  if isNan(testParams.GR) then
    testParams.GR := theGuess.GR;
  // penalise physiologically nonsense parameters:
  if (theGuess.GE <= 0) or (theGuess.GR <= 0) then
    distance := Math.Infinity
  else
  begin
    steadyState := PredictSteadyState(CRH, testParams);
    if steadyState[0].ACTH > steadyState[1].ACTH then
      i := 0
    else
      i := 1;
    distanceA := steadyState[i].ACTH - EvoTargets.ACTH;
    distanceF := steadyState[i].F - EvoTargets.F;
    // Euclidian distance of ACTH and F from the target:
    distance := sqrt(sqr(distanceA) + sqr(distanceF));
  end;
  Result := -distance;
end;
```

This function delivers the fitness of the individual `theGuess` as a real number, which represents the negative Euclidean distance from the target values provided in `EvoTargets`. A higher result denotes a higher fitness. In order to be able to select the parameters to approach `GE` and `GR` are only evaluated if they are passed as **NaN**.

InitialPopulation

```
function InitialPopulation(const size: integer; const params:
         TParams; const lowBound, highBound: real): TPopulation;
var
  i: integer;
  individual: TIndividual;
begin
  assert(size <> 0, kError100);
  assert(size >= 0, kError101);
  assert(highBound > lowBound, kError103);
  SetLength(Result, size);
  for i := 0 to size - 1 do
  begin
    if isNan(params.GE) then   // NaN if to be estimated by GA
      individual.GE := runif(lowBound, highBound);
    if isNan(params.GR) then
      individual.GR := runif(lowBound, highBound);
    Result[i] := individual;
  end;
end;
```

The function `InitialPopulation` delivers an initial population with parameters that are passed as `params`. Parameters that should be subject to random variation are to be passed as **NaN**. In this case, the respective parameters are provided as random numbers from a uniform distribution between `lowBound` and `highBound`).

Fittest

```
function Fittest(const Population: TPopulation): TIndividual;
var
  i, index: integer;
  bestScore: real;
begin
  index := 0;
  bestScore := -Math.Infinity;
  for i := 0 to length(Population) - 1 do
  begin
    if Population[i].fitness > bestScore then
    begin
      bestScore := Population[i].fitness;
      index := i;
    end;
  end;
  Result := Population[index];
end;
```

This function returns the fittest member of a given population.

Selection

```
function Selection(const population: TPopulation;
         const TournamentSize: integer): TPopulation;
var
  indices, tournament: TIntArray;
  competitors: TPopulation;
  winner: TIndividual;
  i, j: integer;
begin
  assert(TournamentSize <> 0, kError100);
  assert(TournamentSize >= 0, kError101);
  SetLength(tournament, TournamentSize);
  SetLength(competitors, TournamentSize);
  SetLength(Result, length(Population));
  indices := IncIndex(length(population));
  for i := 0 to length(population) - 1 do
  begin
    tournament := Sample(Indices, TournamentSize);
    for j := 0 to TournamentSize - 1 do
      competitors[j] := population[tournament[j]];
    winner := Fittest(competitors);
    Result[i] := winner;
  end;
end;
```

The function `Selection` implements an evolutionary selection algorithm that is based on a tournament method. It returns a population consisting of the fittest members of repeated tournament samples. This population defines the parents for the crossover operation.

Crossover

```
function Crossover(const parents: TParents; const params:
         TParams): TChildren;
var
  alleles: record
    GE, GR: tAllele;
    end;
  meioticIndex, crossing: TIntArray;
begin
  SetLength(meioticIndex, 2);
  SetLength(crossing, 2);
  meioticIndex[0] := 0;
  meioticIndex[1] := 1;
  if isNan(params.GE) then
  begin
    alleles.GE[0] := parents[0].GE;
    alleles.GE[1] := parents[1].GE;
    crossing[0] := Sample(meioticIndex, 1)[0];
    crossing[1] := 1 - crossing[0];
    Result[0].GE := alleles.GE[crossing[0]];
    Result[1].GE := alleles.GE[crossing[1]];
  end;
  if isNan(params.GR) then
  begin
    alleles.GR[0] := parents[0].GR;
    alleles.GR[1] := parents[1].GR;
    crossing[0] := Sample(meioticIndex, 1)[0];
    crossing[1] := 1 - crossing[0];
    Result[0].GR := alleles.GR[crossing[0]];
    Result[1].GR := alleles.GR[crossing[1]];
  end;
end;
```

The function `crossover` implements a crossover algorithm, separately for GE and GR. It is based on a uniform order-based crossover operator (Fig. 4). Parameters to be modified are marked by assigning **NaN** to them.

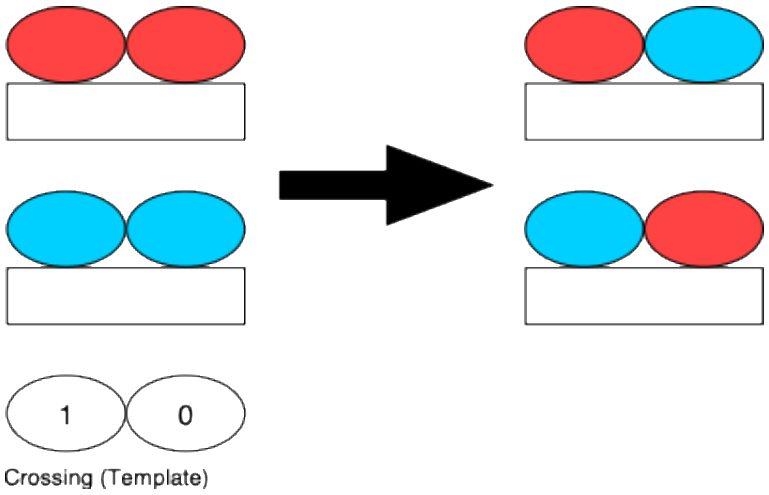


Figure 4: Uniform order-based crossover

Mutated

```
function Mutated(const Individual: TIndividual; const params:
         TParams; const MutationRate: real; const lowBound,
         highBound: real): TIndividual;
var
  intensity: real;
begin
  assert(MutationRate <> 0, kError100);
  assert(MutationRate >= 0, kError101);
  assert(highBound > lowBound, kError103);
  Result := Individual;
  if random < MutationRate then
  begin
    intensity := runif(-1, 1);
    if isNan(params.GE) then
    begin
      Result.GE := Individual.GE * intensity;
      Result.GE := max(min(Result.GE, highBound), lowBound);
    end;
    if isNan(params.GR) then
    begin
      Result.GR := Individual.GR * intensity;
      Result.GR := max(min(Result.GR, highBound), lowBound);
    end;
  end;
end;
```

This function implements a mutation algorithm that is applied to `Individual`. The probability of mutation is defined by `MutationRate`. Parameters that should be subject to mutation have to be passed as **NaN**.

GeneticAlgorithm

```
procedure GeneticAlgorithm(const size: integer; const CRH:
          extended; var params: TParams; const lowBound,
          highBound: real; const EvoTargets: TEvoTargets; const
          generations: integer; mutationRate: real; var
          AllPopulations: TAllPopulations; var theFittest:
          TFittest);
var
  curPopulation, nextPopulation: TPopulation;
  bestIndividual: TIndividual;
  parents: TParents;
  children: TChildren;
  i, j, k: integer;
begin
  assert(size <> 0, kError100);
  assert(size >= 0, kError101);
  assert(highBound > lowBound, kError103);
  assert(generations <> 0, kError100);
  assert(generations >= 0, kError101);
  assert(MutationRate <> 0, kError100);
  assert(MutationRate >= 0, kError101);
  SetLength(AllPopulations, generations);
  SetLength(theFittest, generations);
  SetLength(nextPopulation, size);
  curPopulation := InitialPopulation(size, params, lowBound,
                   highBound);
  for i := 0 to generations - 1 do
  begin
    for j := 0 to size - 1 do
      curPopulation[j].fitness :=
        Fitness(CRH, params, curPopulation[j], EvoTargets);
    bestIndividual := Fittest(curPopulation);
    theFittest[i] := bestIndividual;
    AllPopulations[i] := CurPopulation;
    curPopulation := Selection(curPopulation, TournamentSize);
    for k := 0 to length(curPopulation) - 1 do
    begin
      if not odd(k) then
      begin
        parents[0] := curPopulation[k];
        parents[1] := curPopulation[k + 1];
        children := Crossover(parents, params);
        nextPopulation[k] :=
          Mutated(children[0], params, MutationRate, lowBound,
          highBound);
        nextPopulation[k + 1] :=
          Mutated(children[1], params, MutationRate, lowBound,
```

```
          highBound);
      end;
    end;
    nextPopulation[0] := bestIndividual;
    curPopulation := nextPopulation;
  end;
  if isNan(params.GR) then
    params.GR := theFittest[generations - 1].GR;
  if isNan(params.GE) then
    params.GE := theFittest[generations - 1].GE;
end;
```

This is the main algorithm of the evolutionary mechanism. The stop criterion is that the maximum number of generations has been reached. The parameters of the feedback loop are received with the variable `params` that is called by reference. The results are returned with the same variable. As in other functions, parameters to be subject to evolution are to be marked with **NaN**.

References

1. Alhijawi, B., Awajan, A. Genetic algorithms: theory, genetic operators, solutions, and applications. Evol. Intel. 17, 1245–1256 (2024). https://doi.org/10.1007/s12065-023-00822-6
2. Dietrich JW, Boehm BO. Equilibrium behaviour of feedback-coupled physiological saturation kinetics. In Trappl R (Editor): Cybernetics and Systems 2006. Austrian Society for Cybernetic Studies. doi: 10.13140/2.1.2400.2568
3. Dietrich, JW, Boehm, BO Die MiMe-NoCoDI-Plattform: Ein Ansatz für die Modellierung biologischer Regelkreise. German Med. Sci. Abstr. 284. doi: 10.3205/15gmd s058 (2015)
4. Dietrich JW. A Methodology for Vertical Translation Between Molecular and Organismal Level in Biological Feedback Loops. bioRxiv 2021.09.20.461028; doi: 10.1101/2021.09.20.461028
5. Dietrich, JW, Siegmar, N, Hojjati, JR, Gardt, O, & Boehm, BO (2024). CyberUnits Bricks: An Implementation Study of a Class Library for Simulating Nonlinear Biological Feedback Loops. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal, 13(1), e31762. https://doi.org/10.14201/adcaij.31762
6. Ghosh D, Mandal C. Clustering Based Parameter Estimation of Thyroid Hormone Pathway. IEEE/ACM Trans Comput Biol Bioinform. 2022 Jan-Feb;19(1):343-354. doi: 10.1109/TCBB.2020.2995589. PMID: 32750849.
7. Katoch S, Chauhan SS, Kumar V. A review on genetic algorithm: past, present, and future. Multimed Tools Appl. 2021;80(5):8091-8126. doi: 10.1007/s11042-020-10139-6. PMID: 33162782; PMCID: PMC7599983.
8. Ribeiro Filho JL, Treleaven, PC. Genetic-Algorithm Programming Environments. Computer 1994;26(6):28–43. doi: 10.1109/2.294850.

Contact for Correspondence

apl. Prof. Dr. med. Johannes W. Dietrich, MLS[1, 2, 3, 4]
[1]Sektion Diabetologie, Endokrinologie und Stoffwechsel, St. Josef-Hospital, Ruhr University of Bochum, Gudrunstr. 56, D-44791 Bochum, NRW, Germany
[2]Diabeteszentrum Bochum/Hattingen, Klinik Blankenstein, Im Vogelsang 5–11, D-45527 Hattingen, NRW, Germany
[3]Zentrum für Seltene Endokrine Erkrankungen (ZSE), Centrum für Seltene Erkrankungen Ruhr (CeSER), Alexandrinenstr. 5, D-44791 Bochum, NRW, Germany
[4]Zentrum für Diabetestechnologie (ZDT), Katholisches Klinikum Bochum, Im Vogelsang 5–11, D-45527 Hattingen, NRW, Germany

http://**simuladren**.sf.net

DOI 10.5281/zenodo.16324703