# An Introduction to ParadisEO framework

Sébastien Verel

LISIC
Université du Littoral Côte d'Opale
Equipe OSMOSE

verel@univ-littoral.fr
http://www.lisic.univ-littoral.fr/~verel

2023, version 0.1

### Definition from wikipedia

*ParadisEO is a white-box object-oriented framework dedicated to the flexible design of metaheuristics."*

- Written in C++
- Open source, and free:
  `https://gitlab.inria.fr/paradiseo/paradiseo`
  `https://github.com/nojhan/paradiseo`
- Highly efficient ;-)

# Brief history

https://nojhan.github.io/paradiseo/#History

- EO module started in 1998/1999,
  Geneura Team, Univ. of Granada, Juan Julián Merelo.
- Reinforced by Maarten Keijzer (Vrije Universiteit Amsterdam), Marc Schoenauer (Inria Saclay, France)

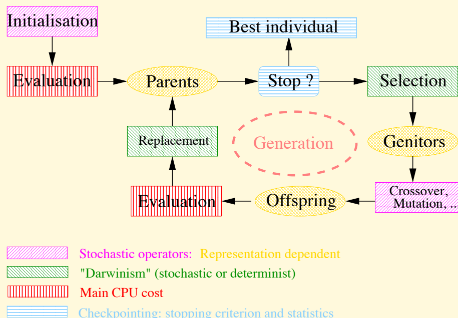
# Brief history

https://nojhan.github.io/paradiseo/#History

- Around 2003, parallelization modules, Dolphin Tean, inria Lille, E-G. Talbi, T. Legrand, S. Cahon, N. Melab,
- Around 2006, Multi-objective module,
  A Liefooghe, J. Humeau
- Around 2010, Local search and fitness landscape module,
  S. verel, A Liefooghe, J. Humeau
- EO, and EDO module,
  Johan Dréo (Thales)
  B. Bouvier, A. Quemy, MPI parallelization
- In 2012, the two projects (EO and Paradiseo) were merged.
- In 2020, automated algorithm selection tools.
  Johan Dréo

# Publications

## An Evolutionary Algorithm




From EO tutorial

*Evolving Objects – EA 2001*

•First •Last •Go Back •Go To •Find

EA'01 conference, Marc Schoenauer et al., 2001

# HellEO!

see code hellEO.cpp

```
#include <eo>     // Core
#include <ga.h>   // Genetic Algorithm part

typedef double Fitness;       // Fitness type
typedef eoBit<Fitness> Indi; // Type of objects to evolve


int main(int argc, char** argv) {
        // solution of 20 bits length
        Indi solution(20);

        // Print it, but without a valid fitness value.
        std::cout << solution << std::endl;
}
```

To compile and execute from directory `tuto`:

mkdir build

cd build

cmake ..

make

./src/hellEO

# HellEO!

see code hellEO.cpp

```
#include <eo>     // Core
#include <ga.h>   // Genetic Algorithm part

typedef double Fitness;      // Fitness type
typedef eoBit<Fitness> Indi; // Type of objects to evolve


int main(int argc, char** argv) {
        // solution of 20 bits length
        Indi solution(20);

        // Print it, but without a valid fitness value.
        std::cout << solution << std::endl;
}
```

To create the documentation from build directory of paradisEO:
`make doc`

Check the documentation of `eoBit`:

`eoBit` is an `eoVector` which is `EO`, and a `vector<AtomType = bool>`

The type of the solution `Indi` is the template of almost all methods.

# The core class EO

EO class defines the object to evolve: Evolution of Object (EO)

```
template<class F = double>
class EO: public eoObject, public eoPersistent
{
public:
  typedef F Fitness;

(...)

private:
  Fitness repFitness;   // value of fitness for this chromosome
  bool invalidFitness;  // true if the value of fitness is invalid
}
```

All solutions (individuals in the field of evolution algorithm) is derived from the EO class.

```
template <class FitT, class ScalarT = bool>
class eoBit: public eoVector<FitT, ScalarT> { ... };

template <class FitT, class GeneType>
class eoVector : public EO<FitT>, public std::vector<GeneType> { ... };
```

# Evaluation function

see code rndInit_fitness.cpp

```
#include <eo>
#include <mo>
#include <eval/oneMaxEval.h> // oneMax problem from problems/eval/oneMaxEval.h

int main(int argc, char** argv) {
    // pre-define fitness function to maximize: oneMax problem, i.e. number of 1
    oneMaxEval<Indi> eval;

    Indi solution(20);

    // evaluation of the solution
    eval(solution);

    std::cout << solution << std::endl;
}
```

# Typical evaluation function class

```
#include <eoEvalFunc.h>

template< class EOT >
class oneMaxEval : public eoEvalFunc<EOT>
{
public:

    void operator() (EOT& _sol) {
        unsigned int sum = 0;

        for (unsigned int i = 0; i < _sol.size(); i++)
            sum += _sol[i];

        _sol.fitness(sum);
    }
};
```

- Derived from `eoEvalFunc`
- Notice the template EOT which is the type of solutions
- The operator bracket () with parameter EOT& has to be defined
- Fitness value is set with the method `fitness`

# Random initialization

```
#include <eo>
#include <mo>
#include <eval/oneMaxEval.h> // oneMax problem from problems/eval/oneMaxEval.h

int main(int argc, char** argv) {
    unsigned int n = 20;
    unsigned int seed = 42;

    oneMaxEval<Indi> eval;

    // one single random generator for all paradisEO
    rng.reseed(seed);

    // initialization of one single gene (bit)
    eoUniformGenerator<Indi::AtomType> genInit;

    // Initialization of the solution (vector of bits)
    eoInitFixedLength<Indi> init(n, genInit);

    Indi solution(n);

    // Apply initialization method
    init(solution);

    eval(solution);

    std::cout << solution << std::endl;
}
```

# Initialization class, and concept of the framework

```
template <class EOT>
class eoInitFixedLength : public eoInit<EOT>
{
public:
   typedef typename EOT::AtomType AtomType;

   eoInitFixedLength(unsigned _combien, eoRndGenerator<AtomType>& _generator)
            : combien(_combien), generator(_generator) {}

   virtual void operator()(EOT& chrom) {
      chrom.resize(combien);
      std::generate(chrom.begin(), chrom.end(), generator);
      chrom.invalidate();
   }

 private :
    unsigned combien;
    eoSTLF<AtomType> generator;
};
```

- Derived from `eoInit`
- The operator bracket () with parameter EOT& has to be defined
- Functor style: parameters are initialized in the constructor, interface "operator()" is the same
- The method `invalidate()` is used to invalidate the fitness value

# Input parameters and parser

```
int main(int argc, char** argv) {
    eoParser parser(argc, argv);

    // random seed
    uint32_t seed = parser.createParam(time(0), "seed",
        "Random␣number␣seed", 'S', "Input").value();

    // bit string size
    size_t n = parser.createParam<size_t>(20, "vecSize",
       "Problem␣dimension", 'n', "Problem").value();

    make_verbose(parser);
    make_help(parser);
(...)
}
```

Execute from terminal:

```
./src/rndInit_fitness --help
```

```
./src/rndInit_fitness -S=42 -n=100
./src/rndInit_fitness --seed=42 --vecSize=100
```

# Exercice 1 on this lesson

## Quadratic fitness function

Define the following evaluation function:
$\forall x \in \{0,1\}^n$,

$$
\begin{aligned}
f(x) &= \sum_{i=0}^{n-2} g(x_i, x_{i+1}) + g(x_{n-1}, x_0) \\
&= g(x_0, x_1) + g(x_1, x_2) + \ldots + g(x_{n-2}, x_{n-1}) + g(x_{n-1}, x_0)
\end{aligned}
$$

with:

$$
\begin{aligned}
g(0,0) &= 1 \\
g(0,1) &= 0 \\
g(1,0) &= 0 \\
g(1,1) &= 2
\end{aligned}
$$

*Method*: copy/paste the class `oneMaxEval`, change the name of the class, and modify the method `void operator(EOT&)`.

# Exercice 2 on this lesson

## Sphere function

(i) Define the following evaluation function:

$$\forall x \in [-5, 5]^n,\ f(x) = \sum_{i=0}^{n-1} x_i^2$$

(ii) Define initialization method of solution $x \in [-5, 5]^n$.

*Method*:
to define the type of solutions use:

```
#include <es.h>

eoReal<EOT> ...
```

for initialization use:

```
eoUniformGenerator<double> genInit(-5.0, 5.0);
```

The paradisEO framework is composed of different modules:

- EO "Evolving Object" : core and evolutionary algorithms
- **MO "moving object"** : dedicated to local search
- MOEO "multi-objective EO" : Multiobjective optimization
- EDO : estimation distribution algorithms
- SMP, and EO/MPI : parallelization of algorithms
- problems : collection of optimization problems

# Hill-Climber with first improvement pivot rule

see code hillClimber.cpp

```
#include <eo>  // Evolving Object: core of EO, evolutionary algorithm
#include <mo>  // Moving  Object: Local search, combinatorial optimization mainly
#include <eval/nkLandscapesEval.h>
#include <problems/eval/moNKlandscapesIncrEval.h>

typedef double Fitness;
typedef eoBit<Fitness> Indi;
typedef moBitNeighbor<Fitness> Neighbor  // type of neighbor

int main(int argc, char** argv) { (...)
    nkLandscapesEval<Indi> eval(n, k);

    // Incremental evaluation of the neighbor
    moNKlandscapesIncrEval<Neighbor> neighborEval(eval);

    // Exploration of the neighborhood in random order of the neighbor's index:
    moRndWithoutReplNeighborhood<Neighbor> neighborhood(n);

    // hill-climber local search with first-improvement pivot rule
    moFirstImprHC<Neighbor> solver(neighborhood, eval, neighborEval);

   Indi solution(n);
   init(solution);
   eval(solution);

    // run the algorithm
    solver(solution);

   std::cout << solution << std::endl;
}
```

# Moving a solution with a Neighbor

```
template<class EOType, class Fitness = typename EOType::Fitness>
class moNeighbor: public EO<Fitness> {
public:
        typedef EOType EOT;

        virtual void move(EOT & _solution) = 0;
};
```

- `moNeighbor` has the same properties of `EOT` (fitness value, etc.)
- The main template is `EOT`, but fitness type could be different
- `move` moves the solution on the neighbor (MO: Moving Object module)

# Example: moBitNeighbor

```
template<class EOT, class Fitness = typename EOT::Fitness>
class moIndexNeighbor: virtual public moNeighbor<EOT, Fitness> {
public:
        inline unsigned int index() const {  return key;   }

        void index(unsigned int _key) {    key = _key;    }
protected:
        // key allowing to describe the neighbor
        unsigned int key;
};

template< class Fitness >
class moBitNeighbor : public moIndexNeighbor<eoBit<Fitness> >, public moBackableNeighbor
{
public:
        virtual void move(EOT & _solution) {
                _solution[key] = !_solution[key];
                _solution.invalidate();
        }

        virtual void moveBack(EOT & _solution) {
                move(_solution);
        }
};
```

- `moIndexNeighbor` adds the property "index" (id of neighbor)
- `move` flips the corresponding bit
- `moveBack` allows to move to the orignal solution (`moBackableNeighbor`)

# Exploration of a Neighborhood

```
template< class Neighbor >
class moNeighborhood : public eoObject
{
public:
    typedef typename Neighbor::EOT EOT;

    moNeighborhood() {}

    virtual bool hasNeighbor(EOT & _solution) = 0 ;

    virtual void init(EOT & _solution, Neighbor & _current) = 0 ;

    virtual void next(EOT & _solution, Neighbor & _current) = 0 ;

    virtual bool cont(EOT & _solution) = 0 ;
};
```

- `moNeighborhood` is similar to an iterator over the all neighbors of a solution
- `init` : first neighbor, `next` : next neighbor, `cont` : test if there is a next neighbor
- `hasNeighbor` to test if there is at least one neighbor (safer algorithm)

# Example : moOrderNeighborhood

```
template<class Neighbor>
class moOrderNeighborhood: public moIndexNeighborhood<Neighbor> {
public:
         using moIndexNeighborhood<Neighbor>::getNeighborhoodSize;

        virtual bool hasNeighbor(EOT& _solution) {
          return getNeighborhoodSize() > 0;
        }

        virtual void init(EOT & _solution, Neighbor & _neighbor) {
                currentIndex = 0;
                _neighbor.index(_solution, currentIndex);
        }

        virtual void next(EOT & _solution, Neighbor & _neighbor) {
                currentIndex++;
                _neighbor.index(_solution, currentIndex);
        }

        virtual bool cont(EOT & _solution) {
          return (currentIndex < getNeighborhoodSize() - 1);
        }
protected:
        unsigned int currentIndex;
};
```

- Iterate over the indexed neighbor from 0 to the size - 1
- See also `moRndWithReplNeighborhood`, and `moRndWithoutReplNeighborhood`

# Evaluation of a neighbor

```
template< class Neighbor >
class moOneMaxIncrEval : public moEval<Neighbor>
{
public:
    virtual void operator()(EOT & _solution, Neighbor & _neighbor) {
        if (_solution[_neighbor.index()] == 0)
            _neighbor.fitness(_solution.fitness() + 1);
        else
            _neighbor.fitness(_solution.fitness() - 1);
    }
};
```

- `operator()` sets the fitness value of the neighbor according to solution, and neighbor
- Partial evaluation (incremental) when it is possible, it reduces the time complexity.

# When incremental evaluation is not possible: moFullEvalByModif, or moFullEvalByCopy

```
template<class BackableNeighbor>
class moFullEvalByModif : public moEval<BackableNeighbor>
{
public:
    moFullEvalByModif(eoEvalFunc<EOT>& _eval) : eval(_eval) {}

    void operator()(EOT & _sol, BackableNeighbor & _neighbor)
    {
        // save current fitness value
        Fitness tmpFit;
        tmpFit = _sol.fitness();

        // move the current solution wrt _neighbor
        _neighbor.move(_sol);

        // eval the modified solution
        _sol.invalidate();
        eval(_sol);

        // set the fitness value to the neighbor
        _neighbor.fitness(_sol.fitness());

        // move the current solution back
        _neighbor.moveBack(_sol);
        _sol.fitness(tmpFit);
    }
};
```

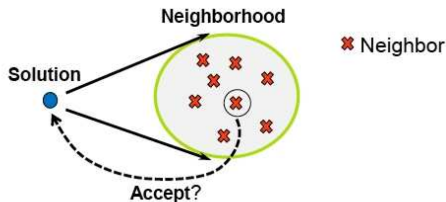
# Metaheuristic with an unique solution : Local Search (LS)

- $\mathcal{X}$ set of solutions (search space)
- $f : \mathcal{X} \to \mathbb{R}$ objective function
- $\mathcal{V}(x)$ set of neighbor's solutions of $x$

Initialize $x \in \mathcal{X}$
Evaluate $x \in \mathcal{X}$

**repeat**
  Select $x'$ from $\mathcal{V}(x)$
  **if** Accept$(x, x')$ **then**
    $x \leftarrow x'$
  **end if**

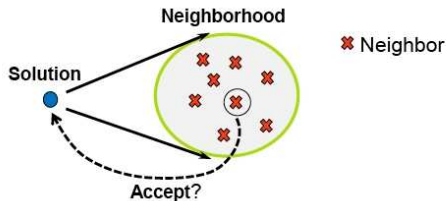
**until** not continue$(x)$



# Metaheuristic with an unique solution : Local Search (LS)

- $\mathcal{X}$ set of solutions (search space)
- $f : \mathcal{X} \to \mathbb{R}$ objective function
- $\mathcal{V}(x)$ set of neighbor's solutions of $x$

```
Initialize x ∈ X
Evaluate x ∈ X
Initialize param. θ of LS
repeat
   Select_θ x' from V(x)
   if Accept_θ(x, x') then
      x ← x'
   end if
   Update param. θ
until not continue(x)
```




Parameter vector $\theta$ of the LS could be temperature, memory, etc.

# Main class: moLocalSearch

```
template<class Neighbor>
class moLocalSearch: public eoMonOp<typename Neighbor::EOT> {
public:
        virtual bool operator()(EOT & _solution) {
                if (_solution.invalid())  fullEval(_solution);

                // initialization of the parameter of the search
                searchExplorer.initParam(_solution);
                // initialization of the external continuator
                cont->init(_solution);

                bool b;
                do {
                        // explore the neighborhood of the solution
                        searchExplorer(_solution);
                        // if a solution in the neighborhood can be accepted
                        if (searchExplorer.accept(_solution)) {
                                searchExplorer.move(_solution);
                                searchExplorer.moveApplied(true);
                        } else
                                searchExplorer.moveApplied(false);

                        // update the parameter of the search
                        searchExplorer.updateParam(_solution);

                        b = (*cont)(_solution);
                } while (b && searchExplorer.isContinue(_solution));

                searchExplorer.terminate(_solution);
                cont->lastCall(_solution);
                return true;
        }
```

- a specific LS is defined by `moNeighborhoodExplorer`
- Large number of pre-defined LS:
  Hill-Climbers, Simulated Annealing, Tabu Search, ILS, VNS, etc.
- `LocalSearch` is also a mutation operator (see later Hydrid EA)

```
template< class Neighbor >
class moNeighborhoodExplorer : public eoUF<typename Neighbor::EOT&, void> {
public:
    virtual void initParam (EOT& _solution) = 0 ;
    virtual void operator() (EOT& _solution) = 0 ;
    virtual void updateParam (EOT& _solution) = 0 ;
    virtual bool isContinue(EOT& _solution) = 0 ;
    virtual bool accept(EOT& _solution) = 0 ;
    virtual void terminate(EOT& _solution) = 0 ;

protected:
  // the current neighbor of the exploration : common features of algorithm
    Neighbor currentNeighbor;
  // the selected neighbor after the exploration of the neighborhood
    Neighbor selectedNeighbor;
};
```

# Hill-Climber with first improvement pivot rule

```
typedef double Fitness;
typedef eoBit<Fitness> Indi;
typedef moBitNeighbor<Fitness> Neighbor  // type of neighbor

int main(int argc, char** argv) { (...)
    nkLandscapesEval<Indi> eval(n, k);

    // Incremental evaluation of the neighbor
    moNKlandscapesIncrEval<Neighbor> neighborEval(eval);

    // Exploration of the neighborhood in random order of the neighbor's index:
    moRndWithoutReplNeighborhood<Neighbor> neighborhood(n);

    // HERE it can be any Local Search
    moFirstImprHC<Neighbor> solver(neighborhood, eval, neighborEval);

   Indi solution(n);
   init(solution);
   eval(solution);

    // run the algorithm
    solver(solution);

   std::cout << solution << std::endl;
}
```

Easy ?

# Stopping criterium: moContinuator, and eoContinue

```
// time stopping criterium
moTimeContinuator<Neighbor> timeContinuator(duration, false);

// maximum number of evaluations
moEvalsContinuator<Neighbor> evalContiuator(eval, neighborEval, maxEval, false);

// multiple stopping criteria
moCombinedContinuator<Neighbor> continuator(timeContinuator);
continuator.add(evalContiuator);
```

- A large collection of available stopping criteria
- Can be combined with `moCombinedContinuator`:
  Stop when at least one criterium is fulfilled
- You can defined your criterium by deriving `moContinuator`
  `bool operator() (EOT &)`

# Example of an Iterated Local Search

see code ils.cpp

```
moLocalSearch<Neighbor> * hc;

// parameter to select the pivot rule
switch (selectionStrategy) {
        // best improvement
  case 0:
  hc = new moSimpleHC<Neighbor>(orderNeighborhood, eval, neighborEval, continuator);
  break;
        // first improvement
  case 1:
  hc = new moFirstImprHC<Neighbor>(rndNeighborhood, eval, neighborEval, continuator);
  break;
}

// Perturbation: standard bit-flip mutation for bitstring
eoBitMutation<Indi>  mutation(mutationRate_perBit);

moILS<Neighbor, Neighbor> solver(*hc, eval, mutation, continuator);
```

# Exercice 3

**Questions**

(i) From the code `ils.cpp`, change the fitness function to use the quadratic fitness function from exercice 1.

(ii) Add a new continuator to stop the algorithm reach the maximum fitness ($2n$ is the maximum fitness)

(iii) Test different parameters settings to design the more efficient algorithm.

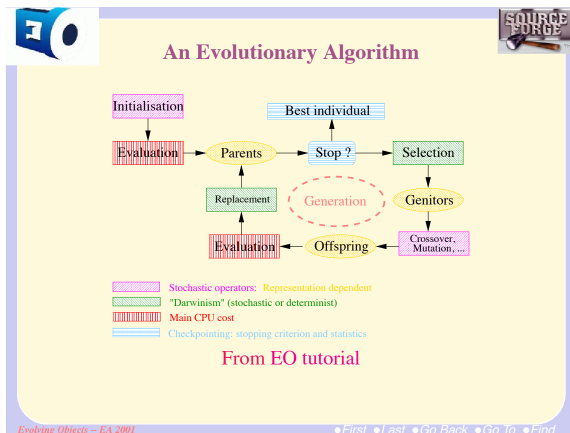
Method:

(i) For evaluation of a neighbor, there is two possibilities. The easiest one is to use `moFullEvalByModif`. The fastest one is to define a new `moIncrEval` to compute the partial evaluation (see `mo/src/problems/eval/moOneMaxIncrEval.h` as a prototype)

(ii) To add this new continuator, please see `moFitContinuator`.

# Core of EO part on Evolutionary Algorithm (EA)

Go back to the original tutorial of EO:




Main principle: lego-like design

- Define each component of the algorithm to define your efficient EA

# Population, and initialization with evaluation

see code hybridEA.cpp

```
// create a pop of size mu using init for initialization
eoPop<Indi> pop(mu, init);

// evaluation of the initial population
for(unsigned i = 0; i < pop.size(); i++)
   eval(pop[i]);
```

- eoPop is mainly a vector of EOT
  with some useful features such as print, best, sort, shuffle, etc.

# Selection component

```
// Tournament selection: to select one individual
eoDetTournamentSelect<Indi> selectOne(tSize);

// is now encapsulated in a eoSelectPerc to select a population (by default lambda=mu)
eoSelectPerc<Indi> select(selectOne);
```

- Other selection strategies are possible:
  inheritance from `eoSelect`
  See documentation for more

# Variation operators component

```
// standard bit-flip mutation for bitstring
eoBitMutation<Indi>  bitOp(mutationRate_perBit);
eoProbabilisticMonOp<Indi> bitMutation(bitOp, mutationRate);

// combine with Hill-Climber: bit flip mutation, then hill-climber
eoCombinedMonOp<Indi> mutation(bitMutation);
mutation.add(*hc);

// 1-point crossover for bitstring
eo1PtBitXover<Indi> xover;

// The operators are encapsulated into an eoTransform object
eoSGATransform<Indi> variation(xover, xoverRate, mutation, 1.0);
```

Mutation operator: EOT $\rightarrow$ EOT

- Derived from `eoMonOp`, and overload `operator()(EOT&)`
- Can be combined iteratively with `eoCombinedMonOp`, or in "parallel" with `eoPropCombinedMonOp`
- `moLocalSeach` is an `eoMonOp`: easy to design an hydrid EA

CrossOver operator: EOT $\times$ EOT $\rightarrow$ EOT $\times$ EOT

- Derived from `eoQuadOp`, and overload `operator()(EOT&, EOT&)`
- Can be in "parallel" with `eoPropCombinedQuadOp`

# Replacement component

```
// (mu + lambda)-Evolution Strategy replacement
eoPlusReplacement<Indi> replace;
```

- Derived from `eoRemplacement`:
  `void operator()(eoPop<EOT>& _parents, eoPop<EOT>& _offspring)`
- `eoPlusReplacement` merges the $\mu$ parents, and the $\lambda$ offspring, and keep the $\mu$ best

# The complete hydrid Evolutionary Algorithm

```
eoEasyEA<Indi> solver(continuator, eval, select, variation, replace);

\\ Run it on the population
solver(pop);
```

Easy ?

- Hydrid EA, also called memetic EA:
  combined a local search with a classical EA

# Output, and statistics during execution

It is always useful to compute, and to report some information
But, it is also very painful to make it in a generic way

```
// checkpoint: substitute continuator to report some statistic, ouput, etc.
eoCheckPoint<Indi> checkpoint(continuator);

eoEasyEA<Indi> algo(checkpoint, eval, select, variation, replace);
```

- EO replaces the continuator (stopping crit.) by a checkpoint to report statistics, and other information (such as state of the search for restart)

# Statistics in paradisEO

```
// Create a counter parameter
eoValueParam<unsigned> generationCounter(0, "iteration");
// Increment this counter at each generation to report the number generation
eoIncrementor<unsigned> increment(generationCounter.value());
// add to the checkpoint to be processed, and monitor to be reported
checkpoint.add(increment);

// time counter
eoTimeCounter timeStat;
checkpoint.add(timeStat);

// best fitness in population
eoBestFitnessStat<Indi> bestStat("best");
checkpoint.add(bestStat);

// second moment stats: average and stdev
eoSecondMomentStats<Indi> avgStdStat("avg std");
checkpoint.add(avgStdStat);
```

- A large number of statistics can be computed (see eoStat)
- Of course, you can defined your own statistic (see next slide)

# eoStat to define your statistics

```
template <class EOT, class T>
class eoStat : public eoValueParam<T>, public eoStatBase<EOT> { ... };
```

```
template <class EOT>
class eoStatBase : public eoUF<const eoPop<EOT>&, void> {
public:
  virtual void operator(const eoPop<EOT>&) = 0; \\ from eoUF
  virtual void lastCall(const eoPop<EOT>&) {}
};
```

and, eoValueParam a class to embed a value:

```
template <class ValueType>
class eoValueParam : public eoParam {
(...)
protected:
    ValueType repValue;
};
```

# Export information into file

```
// output into a file: file name, separator, keep file, header
eoFileMonitor monitor(fileOutName, "␣", false, true);

// report some statistics into the file
monitor.add(generationCounter);  // iteration value
monitor.add(timeStat);           // time report
monitor.add(eval);               // number of evaluations
monitor.add(bestStat);           // best fitness in the population
monitor.add(avgStdStat);         // avg and std fitness in the population

// add the monitor to the checkpoint to be processed
checkpoint.add(monitor);
```

- `eoFileMonitor` allows to print into a text file
- Statistics, or other values (from `eoValueParam`) can be reported
- The monitor must be added to the ckeckpoint
  *monitors are processed after statistics in eoCheckpoint*

Please execute the code `hybridEA.cpp` to see the csv file.

# State-of-art black-box continuous algorithms

CMA-Evolution Strategy (CMA-ES) and Differential Evolution (DE) are two evolutionary algorithms
and state-of-art algorithms for black-box numerical optimization.

CMA-ES is defined efficiently in paradisEO (module edo)

Please execute the code `cmaES.cpp`: `./src/cmaES`
The code is based on the web site of paradisEO, and on the code of the previous section.

Notice the fitness type for minimization problem:

```
typedef eoMinimizingFitness Fitness;
```

defines in file `eo/src/eoScalarFitness.h`. This class redefines the `operator<` to deal with minimization problems.

DE is not defined in ParadisEO, so let's go !...

# DE in short

Disclaimer: this is not a tutorial on DE

**DE algorithm: EA algorithm**

```
Initialize(pop)
Evaluate(pop)
repeat
   Mutation(pop, offsprings)
   Xover(pop, offsprings)
   Evaluate(offsprings)
   Replace(pop, offsprings)
until not continue(pop)
```

# DE operators

## Mutation: Rand/1

For each element $i$ of the population:

```
mutant[i] = pop[r1] + F * (pop[r2] - pop[r3])
```

with i, r1, r2, r3 four different indices with r1, r2, r3 random and $F \in [0, 2]$ a parameter (mutation factor)

## Crossover

For each element $i$ of the population:

```
jrand = random(0, d)
for(unsigned j = 0; j < d; j++)
  if (j = jrand or rnd() < CR)
        offspring[i][j] = mutant[i][j] ;
  else
        offspring[i][j] = parents[i][j] ;
```

with $CR \in [0, 1]$ a parameter (crossover rate)

## Replacement

```
  if (offsprings[i] is better than parents[i])
        parents[i] = offsprings[i] ;
```

# DE algorithm: Use standard EA of EO

See code DE.cpp

DE is an EA, so use eoEasyEA with specific breeder, and replacement operators (to define) :

```
DERand1Mutation<Indi> mutation(pop, mutationFactor);
DEStandardXover<Indi> xover(pop, xoverRate);
DEBreed<Indi> breed(mutation, xover);
DEReplacement<Indi> replace;

eoEasyEA<Indi> solver(checkpoint, eval, breed, replace);
```

Indeed, the main function of eoEasyEA is (see documentation):

```
virtual void operator()(eoPop<EOT>& _pop)    {
  popEval(empty_pop, _pop); // A first eval of pop.

  do {
          offspring.clear(); // new offspring
          breed(_pop, offspring);
          popEval(_pop, offspring); // eval of parents + offspring if necessary
          replace(_pop, offspring); // after replace, the new pop. is in _pop
    } while ( continuator( _pop ) );
}
```

# DE Replacement

see utils/deReplacement.h

## DEReplacement inherits from eoReplacement

```
template <class EOT>
class DEReplacement : public eoReplacement<EOT>
{
public:
  void operator()(eoPop<EOT>& _parents, eoPop<EOT>& _offspring)
  {
    for(unsigned i = 0; i < _parents.size(); i++) {
      if (_parents[i].fitness() <= _offspring[i].fitness()) { // if better or equal
        _parents[i] = _offspring[i] ;
      }
    }
  }
};
```

# DE mutation

Several mutation operators are possible in DE (Rand/1, Rand/k, Best/1, etc.), so we design a generic class.

Usually in EO, mutation operators inherit from class `eoMonOp<EOT>` with function to define:

```
virtual bool operator()(EOT & _solution)
```

But in DE, mutation operator also depends on parent population, and the index in the population of the solution to mutate.

Generic class of DE operator (see `utils/deOp.h`):

```
template <class EOT>
class DEOp : public eoMonOp<EOT> {
public:
    DEOp(eoPop<EOT> & _parents) : parents(_parents), id(0) {  }

    void pop(const eoPop<EOT> & _parents) {  parents = _parents;   }
    void index(unsigned _id) {   id = _id;   }
protected:
  eoPop<EOT> & parents;
  unsigned id;
};
```

# Rand/1 mutation

see code utils/deRand1Mutation.h

```
template <class EOT>
class DERand1Mutation : public DEOp<EOT> {
  using DEOp<EOT>::parents;
  using DEOp<EOT>::id;
public:
    DERand1Mutation(eoPop<EOT>& _parents, double _F) : DEOp<EOT>(_parents), F(_F) {}

    virtual bool operator()(EOT & _solution) {
      if (parents.size() > 4) {
        // random different indices
        unsigned i1, i2, i3;
        initRnd();
        i1 = nextRnd();
        i2 = nextRnd();
        i3 = nextRnd();

        // mutation
        _solution.resize(parents[id].size());

        for(unsigned j = 0; j < _solution.size(); j++)
          _solution[j] = parents[i1][j] + F * (parents[i2][j] - parents[i3][j]);

        _solution.invalidate();
        return true;
    }
protected:
  double F;

  // (...) defines initRnd, and nextRnd below, see code
};
```

# DE crossover

see code utils/deStandardCrossover.h

Indeed in DE, the crossover only produces one solution, so it could be an eoMonOp<EOT>. Moreover, the crossover needs parents, and index. So crossover can inherit from DEOp<EOT> :

```
template <class EOT>
class DEStandardXover : public DEOp<EOT> {
  using DEOp<EOT>::parents;
  using DEOp<EOT>::id;
public:
  DEStandardXover(eoPop<EOT> & _parents, double _CR) : DEOp<EOT>(_parents), CR(_CR) { }

  virtual bool operator()(EOT & _solution) {
    unsigned jrand = rng.random(_solution.size());

    for(unsigned j = 0; j < jrand; j++)
      if (rng.uniform() > CR)
        _solution[j] = parents[id][j];

    for(unsigned j = 0; j < jrand + 1; j++)
      if (rng.uniform() > CR)
        _solution[j] = parents[id][j];

    _solution.invalidate();
    return true;
  }
protected:
  double CR;
};
```

# DE breed

## DEBreed<EOT> inherits from eoBreed<EOT>

```
template <class EOT>
class DEBreed : public eoBreed<EOT>
{
public:
    DEBreed(DEOp<EOT> & _mutation, DEOp<EOT> & _xover) : mutation(_mutation), xover(_xov

    void operator()(const eoPop<EOT>& _parents, eoPop<EOT>& _offspring) {
      _offspring.resize(_parents.size());

      mutation.pop(_parents);  // to set the parent vector
      xover.pop(_parents);       // to set the parent vector

      for(unsigned i = 0; i < _offspring.size(); i++) {
        mutation.index(i); // to set the index of the solution to mutate
        mutation(_offspring[i]);

        xover.index(i);     // to set the index of the solution to mutate
        xover(_offspring[i]);
      }
    }
private:
  DEOp<EOT> & mutation;
  DEOp<EOT> & xover;
};
```

# Continuous optimization

The DE algorithm is completed, and show you how to design your own algorithm with available components of paradisEO

**Exercice 4**

Create the Best/1 mutation operator for which the indice r1 is the indice of the best solution of parent population.