

A C++ Function for Evolutionary Optimisation with Applications in Shape Matching

Patrick Min
Utrecht University

DRAFT, v0.47

June 14, 2005

Chapter 1

Introduction

This document describes the usage and implementation details of a C++ class for evolutionary optimisation. It was developed as a CGAL extension package, but it can be used in any other context (i.e. it does not depend on the CGAL library).

Evolutionary algorithms are one of several available techniques for non-linear optimisation (other examples are Powell’s direction set method and simulated annealing [3]). These techniques are applied when objective functions need to be optimised that are non-linear and not easily differentiable. For an overview of the relatively new area of evolutionary programming, see the book by Back and Bäck [1].

The basic idea of these algorithms lies in the theory of evolution, in particular the “survival of the fittest” rule. A population is constructed with each individual representing a single solution. The evolution of this population is then simulated, with “fitter” (i.e. having better objective function values) individuals being more likely to survive and produce offspring. Parameters such as the survival rate of a population allow control over the rate of convergence of the algorithm. The hope is that a slow enough convergence and sufficient variety in the population help to avoid local minima.

1.1 Problem Description

Our C++ class deals with one kind of optimisation problem, of which a fairly abstract description now follows. Given two objects, called a source and a target, which may be of different type, and a transformation which of an object of the source type, the goal is to find the transformation parameters that take the source closest to the target, given a certain distance measure. See Figure 1.1 for a schematic overview of this type of optimisation problem.

So in order to specify a problem of this type, we have to provide:

- a source object type
- a transformed source object type

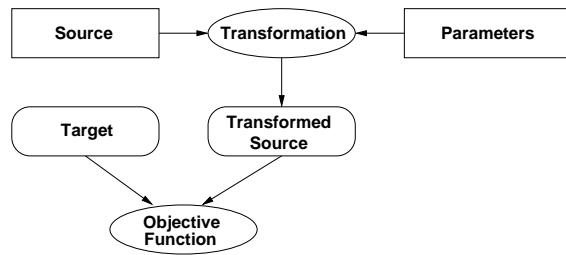


Figure 1.1: Schematic overview of the type of problem our code can handle

- a target object type
- a (parameterised) transformation which transforms an object of the source type
- an objective function which takes an object of the target type and one of the transformed source type and returns a distance (or *error*) value

The “best” parameters of the transformation (given the objective function) now have to be found. In terms of our evolutionary algorithm, an individual in the population contains specific values for each transformation parameter. Its “fitness” is computed by transforming the source object using these parameter settings, and evaluating the distance between the transformed source and the target. These steps are roughly indicated with the dashed arrow in Figure 1.2.

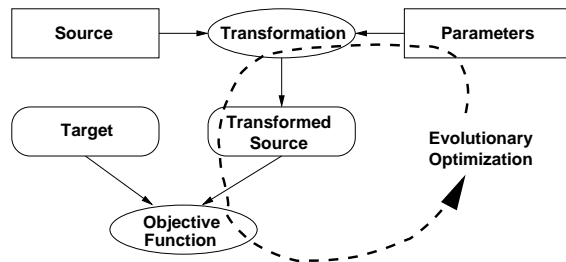


Figure 1.2: A single iteration of the optimization process

Chapter 2

Examples

The parameters of the method are the type of the (1) source object, (2) transformed source object, (3) target object, and the (4) transformation and (5) objective function. To implement this parameterisation, we use a construct called a *traits class*. A traits class is a templated class that allows one to parameterise aspects of a *method*. It is an extra layer of indirection for specifying a templated class.

2.1 Example: Matching 2D Pointsets under Translation and Scale

Suppose that we took a frontal picture of the face of a large number of people. For each picture, we then recorded the position of the eyes, nose, and corners of the mouth, giving five 2D points. Such a set of five 2D points will be our source datatype.

Later, someone brings us a new picture of a face, taken from the front, and wants to know if there are pictures with a similar arrangement of eyes, nose, and mouth in our database. So in this picture we also record the position of these features. It is our target object, so the target datatype is also a 2D pointset containing five points. Let's assume we can be sure that all pictures were taken exactly from the front, and "straight up" (i.e., the eyes are on top), but we don't know where in the picture the face is positioned, and how large the face is on the picture. So the only potential "variability" is in the position and size of the face, in other words, in 2D translation and uniform scale (assuming we know the aspect ratio of the camera pixels). As a result, we have three transformation parameters, namely (t_x, t_y) and s .

For the source 2D pointset we will now try to find the set of transformation parameters (t_x, t_y, s) that take it closest to the target 2D pointset. Of course we have to decide what "closest" means, i.e. we have to define a distance between two of our pointsets: we will use the average Euclidean distance over all five corresponding point pairs. This will be the objective function.

Now we are ready to provide a list as in Section 1.1:

- *source object*: a 2D pointset of 5 points
- *transformed source object*: a 2D pointset of 5 points
- *target object*: a 2D pointset of 5 points
- *transformation*: a 2D uniform scale followed by a 2D translation
- *objective function*: the average Euclidean distance over all 5 corresponding point pairs

2.1.1 Types

We implement the 2D pointset as a separate class `PointSet2D`, and have to specify that this will be the source and target type:

```
typedef PointSet2D SourceType;
typedef PointSet2D TransformedSourceType;
typedef PointSet2D TargetType;
```

The `PointSet2D` constructor should contain initialisation code that creates a pointset of the required size (5 in our example).

2.1.2 Operations

Next, the transformation and objective function have to be specified. Both are implemented as a *function object*, by defining a class that has an `operator()` member function. In the implementation we use the identifier `Map` for a transformation. The objective function is called an `Error` function. First, we specify that the map and error function are of a certain type:

```
typedef class Map_2d_Scale_Trans MapType;
typedef class Avg_Euclidean ErrorType;
```

We also declare two corresponding member variables of our traits class, and define methods that return a copy:

```
Map_2d_Scale_Trans& map_object() const {
    return (Map_2d_Scale_Trans&) map; }
Avg_Euclidean& error_object() const {
    return (Avg_Euclidean&) error; }
private:
    Map_2d_Scale_Trans map;
    Avg_Euclidean error;
```

The functions themselves are defined in a separate class. For example, the declaration of the `Map_2d_Scale_Trans` class looks like this:

2.1. EXAMPLE: MATCHING 2D POINTSETS UNDER TRANSLATION AND SCALE⁷

```
class Map_2d_Scale_Trans : public Map
{
public:
    void operator()(const PointSet2D& source,
                    PointSet2D& result,
                    const vector<double>& parameters);
};
```

In this example, the `parameters` are the three degrees of freedom of the transformation, determining the x and y translation, and scale factor.

To help the optimisation process, we should specify reasonable ranges for each parameter. This should be done in the constructor of the mapping function `Map_2d_Scale_Trans`, as follows:

- declare an `initial_ranges` variable in its class header file:
`static double initial_ranges[6];`
- specify the actual initial values in the `.cc` file:
`double Map_2d_Scale_Trans::initial_ranges[6] = {-25, 25, -25, 25, -1, 1};`
- in its constructor, call:
`Map::init_parameter_ranges(initial_ranges, 6);`

The mapping function should also define three informational functions (which are virtual functions in the base class `Map`), `get_name()`, `get_nr_pars()`, and `get_par_name(int index)`:

```
string get_name() { return "Map_2d_Scale_Trans"; }
int get_nr_pars() { return 3; }
string get_par_name(int index) { return par_names[index]; }
```

`par_names` is a static member variable:

```
private:
    static string par_names[3];
```

It is defined in the `.cc` file:

```
string Map_2d_Scale_Trans::par_names[3] =
    {"translation x", "translation y", "scale"};
```

2.1.3 Running the optimisation

After having created the traits classes, all we have to do next is create instances of the traits classes and source and target variable.

In our example, the required variables can be declared as follows:

```

#include "evo_traits.h"
#include "evo_par_default_traits.h"
#include "EvoFunc.h"

template class EvoFunc<evo_traits, evo_par_default_traits>;

evo_traits::SourceType source;
evo_traits::TargetType target;
evo_traits gt;

EvoFunc<evo_traits, evo_par_default_traits> my_evo_opt(source, target, gt);

```

Now we are ready to run the optimisation:

```
my_evo_opt.run(250); // run a maximum of 250 iterations
```

2.1.4 Results

The optimisation terminates when one of the following is true:

- The best error value so far falls below a certain limit
- The specified maximum number of iterations has been run

It is then possible to get the parameter settings resulting in the smallest error value:

```

Indiv *ind_p = my_evo_opt.get_best_indiv();
int nr_pars = ind_p->get_nr_pars();
for(int i=0; i < nr_pars; i++) {
    cout << i << ": " << (*ind_p)[i] << endl;
}

```

Next, to get the best error value:

```
double best_error = my_evo_opt.evaluate_individual(*ind_p);
```

2.2 Matching 2D Pointsets under 2D Affine Transformations

This next example is very similar to the one in the previous section, just the transformation and objective function are slightly more complicated.

Now we want to find the *2D affine* transformation which takes one 2D pointset into another, whereby distance between the pointsets is measured using the *average Hausdorff* distance. Or, given in list form as in the Section 1.1:

- *source object type*: a 2D pointset

2.2. MATCHING 2D POINTSETS UNDER 2D AFFINE TRANSFORMATIONS 9

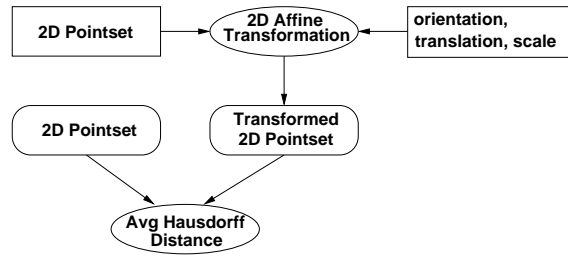


Figure 2.1: The schematic of Figure 1.1 for this example

- *transformed source object type*: a 2D pointset
- *target object type*: a 2D pointset
- *transformation*: a 2D affine transformation
- *objective function*: the average Hausdorff distance

Figure 2.1 shows the same schematic as in Figure 1.1, but now with the specifics of this example filled in.

2.2.1 Types

These definitions are identical to those in the previous example:

```

typedef PointSet2D SourceType;
typedef PointSet2D TransformedSourceType;
typedef PointSet2D TargetType;

```

2.2.2 Operations

These definitions and declarations are also the same as in the previous example, with different names for the map and for the error (objective) function:

```

typedef class Map_2d_Aff MapType;
typedef class Avg_Hausdorff ErrorType;

```

Instantiating them in the traits class:

```

Map_2d_Aff map_object() const { return map; }
Avg_Hausdorff error_object() const { return error; }
private:
    Map_2d_Aff map;
    Avg_Hausdorff error;

```

In this example, the parameters are the five degrees of freedom of a 2D affine transformation, determining the rotation angle, x and y translation, and x and y scale.

2.2.3 Running the optimisation

This part is identical to that of the previous example.

2.3 Source Code Examples

There are source code examples in the `evofunc/src/examples` subdirectory (actually there is only one just now).

2.3.1 `example1` matching 2D pointsets under translation and rotation

The mapping function (transformation) is defined in the `Map_2d_Iso` class, which you can find in `evofunc/src/map_functions`.

The error function (objective function, distance measure) is the average Hausdorff distance, defined in `Avg_Hausdorff.h` (in `evofunc/src/error_functions`).

Traits class definitions are stored in `evofunc/src/traits`. The traits class bringing the 2D isometric transformation and the average Hausdorff distance together is defined in `evo_traits_2d_iso_avg_hausdorff.h`.

After all this has been set up, the `main()` function of the example program is straightforward. It initializes a source and a target pointset, and makes the target be a transformed version of the source. The optimization then tries to recover the three transformation parameters t_x , t_y , and angle ϕ .

2.3.2 `example2` computing fitness values for parameter ranges

For this purpose, the `EvoFunc` class has a function `tt run_2d_grid`:

```
void run_2d_grid(int index[2],
                 vector<vector<double> >& fitness_matrix);
```

The indices of the parameters to use are passed in `index`, the resulting 2D array of fitness values is returned in `fitness_matrix`. The function also writes these values to a file `grid_<i1><i2>.dat` (with `<i1>` and `<i2>` the indices passed in `index`), in a format suitable for `gnuplot`.

Chapter 3

Transformation and Error Functions

This chapter describes the transformation (mapping) and error (objective) functions that are included in the `evofunc` package.

The map functions are in `evofunc/src/map_functions`, the error functions in `evofunc/src/error_functions`, and the traits classes combining them can be found in `evofunc/src/traits`.

3.1 Transformation Functions

3.1.1 2D Isometry

3.1.2 2D Similarity

3.1.3 2D Affine

3.1.4 2D Perspective Projection

3.1.5 2D General Projection

3.1.6 3D General Projection

3.2 Error Functions

3.2.1 Hausdorff distance

3.2.2 Average Hausdorff distance

Chapter 4

Reference

Concept *EvoFunc_Traits*

Definition

The `EvoFunc_Traits` traits class specifies object and function types the user has to define in order to be able to use the `EvoFunc` optimisation class.

Types

`EvoFunc_Traits::SourceType` The type of the object to be transformed

`EvoFunc_Traits::TransformedSourceType` The type of the object after transformation

`EvoFunc_Traits::TargetType` The type of the object we need to get “close” to

Creation

Only a default constructor is required.

Operations

Member functions for the transformation and for computing the distance value have to be provided.

`EvoFunc_Traits::MapType` A function object that takes three parameters: (1) an object of type `SourceType`, (2) an object of type `TransformedSourceType`, and (3) a list of parameters of type `vector<double>`. It should transform the source object according to the parameter settings, and store the result in the second parameter

`EvoFunc_Traits::ErrorType` A function object that takes two parameters, one of type `TransformedSourceType` and one of type `TargetType`, and returns a distance value of type `double`

Has Models

Currently, two models have been implemented for this traits class:

`evo_traits_2d_aff_avg_hausdorff`
`evo_traits_3d_proj_avg_hausdorff`

Concept *EvoPar_Traits*

Definition

All optimisation parameters are initialised to “reasonable” defaults, defined in the *EvoPar* class. A traits class has to be defined containing a setting for each optimisation parameter. No types or operations are defined in this traits class, just constants.

Constants

Stopping criteria

```
int EvoPar_Traits::MAX_ITERATIONS
```

The maximum number of iterations the optimisation is allowed to run

```
double EvoPar_Traits::TERMINATION_CUTOFF
```

If the error value is below this limit, optimisation will stop

```
double EvoPar_Traits::ERROR_EPSILON
```

If the current and previous error value differ by less than this value, optimisation will stop

Evolutionary optimisation parameters

The default values for these parameters were taken from [2].

```
int EvoPar_Traits::POPULATION_SIZE
```

The initial population size

```
double EvoPar_Traits::PARENTS_PERCENTAGE
```

The percentage of the population that will become a parent

```
int EvoPar_Traits::NR_CHILDREN
```

The number of children per parent pair

Miscellaneous parameters

```
int EvoPar_Traits::VERBOSITY
```

Setting this to 0 disables all output to `stdout` during the optimisation (by default set to 1)

Other variables of the method are, for example, the way individuals are mutated and recombined, how parents are selected, and how survivors are selected. The methods we use were also taken from [2]. In future versions of *EvoFunc*, it will be possible to make a selection from multiple choices for each of these methods.

Creation

A default constructor is sufficient.

Has Models

The `evo_par_default_traits` class is an example model of the `EvoPar_Traits` concept. It specifies a reasonable default value for each parameter.

Appendix

Compiling the library

This package was successfully compiled under a Linux kernel 2.4.20, using `gcc` version 3.2.2, under Mac OS X kernel Darwin 6.8, using `gcc`, and under Windows 2000, using Visual Studio C++ 7.1.

First download and unpack the file `evofunc-0.47.tar.gz`. It will unpack to a subdirectory `evofunc-0.47`.

- **Linux and Mac OS X**

1. run `make` in the directory `evofunc-0.47/src/lib`
2. add this directory to your library file search path (using `-L`)
3. add the `evofunc-0.47/src` directory to your include file search path (using `-I`)
4. add `-levofunc` to your link line

- **Windows**

1. in Visual Studio, create a new project `evofunc` as a Win32 project, static library, no precompiled headers
2. **Note:** change the project location to `evofunc-0.47/src/lib`
3. add all files in `evofunc-0.47/src/lib`
4. build the project

Compiling example programs

To compile the example program (which is in `evofunc-0.47/src/examples/example1/example1.cc`):

- **Linux and Mac OS X**

1. change to the directory `evofunc-0.47/src/examples/example1`
2. run `make`

- **Windows**

1. in Visual Studio, create a new project `example1` as a Win32 console project, and check the option “Empty Project”
2. **Note:** change the project location to `evofunc-0.47/src/examples/example1`
3. add the file `example1.cc`
4. add the files in `evofunc-0.47/src/data_types`, *except* `PointSet3D.*` and `MyRandom.*`
5. add the files in `evofunc-0.47/src/error_functions`
6. add the files `Map.*` and `Map_2d_Iso.*` that are in `evofunc-0.47/src/map_functions`
7. select one of the source files (!), and open the project properties dialog (Project → `evofunc` Properties)
8. select C/C++ → General → Additional Include Directories, and add the directory `evofunc-0.47/src`
9. select Linker → General → Additional Library Directories, and add the directory `evofunc-0.47/src/lib/Debug`
10. select Linker → Input → Additional Dependencies, and type `evofunc.lib`
11. now build the project

Using the Traits Class Types and Functions

Here we describe how the types and functions of the traits class are used in the `EvoFunc` class. It is not necessary to know this for using the `EvoFunc` class, but it is useful in case you want to know more about how to use a traits class.

First, we need to put some `typedef`’s in the `EvoFunc` class definition that refer to our method parameters:

```
typedef typename Traits::SourceType SourceType;
typedef typename Traits::TransformedSourceType TransformedSourceType;
typedef typename Traits::TargetType TargetType;

typedef typename Traits::MapType MapType;
typedef typename Traits::ErrorType ErrorType;
```

We also maintain references to the source, transformed source, target and traits class, and copies of the map and error function objects:

```
private:
```

```
SourceType& source;
TransformedSourceType& transformed_source;
TargetType& target;
```

```
const Traits& gen_traits;
```

```
MapType map;
```

```
ErrorType error;
```

The EvoFunc constructor initializes the references and copies the function objects:

```
template<class Traits>
EvoFunc<Traits>::EvoFunc(SourceType& source_ref,
                        TransformedSourceType& transformed_source_ref,
                        TargetType& target_ref,
                        const Traits& evo_traits_ref) :
    source(source_ref),
    transformed_source(transformed_source_ref),
    target(target_ref),
    evo_traits(evo_traits_ref),
    map(evo_traits.map_object()),
    error(evo_traits.error_object())
{
    ...
}
```

Now it is straightforward to call both functions:

```
TransformedSourceType result;
```

```
// get_pars() returns a reference to an individual's parameters
map(source, result, ind_p->get_pars());
```

```
double error_value = error(result, target);
```


Bibliography

- [1] T. Back and T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, January 1996.
- [2] R. Geraerts. Pose estimation met evolutionaire strategieën. Master's thesis, Utrecht University, December 2001. #INF/SCR-01-30, (in Dutch).
- [3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*, chapter 10, Minimization or Maximization of Functions, pages 463–469. Cambridge University Press, 2nd edition, 1992.