The Hitchhiker's Guide to the Diana Optimization Subsystem *

Version 0.8.2b

Srgiy Gogolenko

July 21, 2009

Contents

*This is only draft version of the guide without any warranty that nothing will be drastically changed in the future. Complete version of the guide probably will be provided after final release of the Diana. And finally, this text were written on the language that is similar to the English but not equal. I call it "my English dialect". I cannot promise that if you'll translate it on top English your name will appear in the Diana's Hall of Fame but I'm sure that it'll serve a good job for further development of free scientific software

1 Introduction

This document is a guide to using and further improvement of Diana optimization subsystem which was developed as a part of Diana simulation environment in order to provide convenient interfaces for solving of typical optimization problems that appear during analysis of models for dynamic systems. Current implementation provides limited support of parameter estimation (PE) and optimal experimental design (OED). It allows also to minimize arbitrary continuous functions that are represented by "explicit" formulas. PE support is restricted to the maximum likelihood approach for measurements with Gaussian white noise. Diana optimization subsystem implements completely only Sigma Point approach for OED support (see [7]).

Diana optimization subsystem solves optimization problems using third party optimization libraries (see Section 3). The most of them implements stochastic black box global optimization routines (DIRECT V2.0.4, GENETIC, GMFL, BBOWDA) but a set of local optimization packages (implementation of derivative free Nelder-Mead approach as well as gradient based solvers IPOPT and L-BFGS-B) is also available.

The rest of this document is organized as follows. Section 2 discusses different aspects of using Diana optimization subsystem. This section gives some introductory notes for the users and includes instructions how to write optimization scripts. Section 4 contains brief overview of optimization subsystem for developers. The list of available optimizers with description of their facilities can be found in Section 3 Areas of further work are given in Section 4.2. Finally, the document contains appendix with examples of optimization scripts.

2 Using Optimization in Diana

Since this document assumes that the reader is familiar with the Python programming language and basic Diana features, it is highly recommended to look through the following tutorials:

- Diana Tutorial
- Python Tutorial

before further reading.

Examples in this section are based on the model of the continuous flow stirred tank reactor [7], which can be also found in the subfolder `test/models/SubstrateUptake` of the root folder with Diana sources. Full codes of these examples are presented in appendix A.

From user's point of view Diana optimization subsystem deals with two main groups of objects, namely optimizers (solvers) and optimization problems. Objects of optimization problems incapsulate knowledge about specific tasks that should be solved. They include (explicitly or partially implicitly) objectives, constraints, specifications of unknowns etc. Objects of optimizers realize optimization methods. The following text shows how to create these objects and to specify their parameters.

2.1 General Remarks

2.1.1 Loading Models and Solvers

The process of loading solvers and models is described in Diana Tutorial. The only specific thing concerned with optimization is that `CAPE_NLP` should be used as the first parameter in the method `CreateSolver` of solver factory and the second parameter in this method is ignored (typically one may use `None`). Therefore line for solver loading usually looks so

```
1 solver = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerName)
```

2.1.2 Initialization of Solver and Running Optimization

The optimization problem should be assigned with optimizer before its initialization. Once this is done solver can be initialized:

```
solver.SetNLPTask(task)
solver.Initialize()
```

After initialization it is possible to run optimization by calling of the method `Solve`. One may use `try-except` construction in order to treat exceptions that may occur during optimization:

```
try:
    optimizerOED.Solve()
except:
    print "Estimate of minimum before exception: ", \
        optimizerOED.GetSolution()
    raise
print "Solution: ", optimizerOED.GetSolution()
```

2.1.3 Specification of "explicit" Functions

Diana offers to define objective function and constraints of optimization problem explicitly in the Python code. In this case the function should take reference on the optimization problem as its input argument. It should return its value in the point that can be obtained by the method `GetNLPParameters` of the optimization problem. The method `GetNLPParameters` returns reference on the set of parameters. The method `GetParametersCount` allows to obtain amount of parameters in the set. The method `GetParameterValue` offers to extract a given parameter value.

The following code shows how to describe Rosenbrock's function

$$f(\vec{x}) = \sum_{i=1}^{n-1} \left[(1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right] \tag{1}$$

for Diana optimization subsystem

```
def objfunc_Rosenbrock(task):
    pars = task.GetNLPParameters()  # get reference on parameters
    par = pars.GetParameterValue(0) # extract value of the 1st parameter
    objfunc = 0.
    for i in xrange(pars.GetParametersCount()-1):
        par_next = pars.GetParameterValue(i+1) # obtain (i+1)-th parameter
        objfunc += 100*((par_next - par**2)**2) + ((par - 1)**2)
        par = par_next
    return objfunc
```

2.1.4 Specification of Unknowns

The most of optimization problems obtain information about unknowns in the form of a list that consists of `DianaNLPRealParameterSpec` instances. `DianaNLPRealParameterSpec` objects contain the following fields:

- name of unknown
- description
- default value
  It is used as initial value by some solvers.

Table 1: Common parameters of reporting interfaces for optimizers

Parameter	Default value	Description
`Author`	None	Authors' name (will be shown in title)
`OutputFolder`	`.`	Folder where report will be saved
`ConsoleOutput`	`False`	If its value is `True` solver outputs intermediate results on the console
`Samples`	`False`	If its value is `True`, reporting interface generates file with information about sample points and plot that shows their distribution

- lower and upper bound
- precision
  It is used by some solvers in order to generate mesh for search (e.g. GENETIC) or to check stopping criteria.

In the example below, the list from two unknowns is specified

```
sps = [
    diana.DianaNLPRealParameterSpec("k_s",  # name
                                    "",     # description
                                    1.0,    # default value
                                    1.0,    # lower bound
                                    3.0,    # upper bound
                                    0.0001),# precision
    diana.DianaNLPRealParameterSpec("mu_max", "",  4.0, 4.0, 6.0, 0.0001)
    ]
```

If precisions for unknowns are not taken into account by the chosen optimizer, one may use `DianaRealParameterSpec` instead of `DianaNLPRealParameterSpec`.

2.1.5 Reporting and Obtaining Optimization Results

Diana supports reporting for both optimizers and optimization problems. Reporting interfaces for optimizers gather and represent general information about optimization process and its results. Reporting interfaces for problems allows to collect problem dependent information about calculation of objective function and constraints. Both kinds of reporting interfaces save their data in `tex`-files.

Each optimizer has its own reporting class. Nevertheless all of them have the same structure and common parameters. Some common parameters of reporting interfaces for optimizers are listed in the table 1 below. Generated reports always include the following sections:

- title (information about report and author)
- table of content
- section of optimization results
  - found minimum and table of optimal parameters
  - solver's parameters that were used
- plots that show decrease of objective during optimization

The code below explains how to create and use reporting interfaces for solvers:

```
# create report
report = dmain.CreateReportingInterface(optimizerName)

# set parameters for reporting
pars = report.GetParameters()
pars["Author"].SetValue("Anonymous Diana User")
pars["OutputFolder"].SetValue("./report")
pars["ConsoleOutput"].SetValue(True)
pars["Samples"].SetValue(True)

# initialize report
report.Initialize()

# set prepared task and report to solver
solver.SetReportingInterface(report)
```

The reporting interface produces report's sources (`tex`-file etc) during optimization process after calling of solver's method `Solve`. Once optimization has been finished, the user can generate `pdf`-file from sources of report as shown below

```
cd ./report
latex    ./lbfgsbreport.tex
latex    ./lbfgsbreport.tex
dvipdf   ./lbfgsbreport.dvi
acroread ./lbfgsbreport.pdf &
```

Note that `latex` should be run twice in order to generate table of content correctly. In this case the user also cannot run `pdflatex` directly to produce `pdf`-file since reports usually contain `eps`-graphics.

Support of reporting interfaces for problems is restricted. Only reporting interfaces for OED problems are internally supported in current version of Diana. For further details on this topic see 2.2.2

The user can also access to the optimized parameters in Python directly (without reporting interfaces) using

```
print "Unknowns: \n", solver.GetSolution()
```

or

```
print "Unknowns: \n", task.GetSoughtParameters()
```

The user is able to obtain value of found minimum and some other useful information about optimization process and results from the list of solver's parameters as shown below

```
pars_opt = solver.GetParameters()
print "Found minimum:                ", \
      pars_opt["MinObjFunc"].GetValue()
print "Total amount of objective calls: ", \
      pars_opt["TotalEvalsCount"].GetValue()
print "Total amount of iterations:      ", \
      pars_opt["CurrIteration"].GetValue()
print "Total execution time:             %s sec" % \
      pars_opt["TotalTimeOfWork"].GetValue()
```

Some common parameters of solvers are presented in the table 2

2.2 Problem Dependent Questions

This section shows how to specify optimization problems in Diana. The interfaces for parameter estimation, optimal experimental design and general NLP problems are considered in details. Diana contains

Table 2: Common parameters of optimizers

Parameter	Default value	Description
MinObjFunc	**OUT** [1]	Found minimum of objective function
TotalEvalsCount	**OUT**	Total amount of objective function's evaluations
CurrIteration	**OUT**	Total amount of performed iterations
StartingTime	**OUT**	Time when optimization was started
TotalTimeOfWork	**OUT**	Total execution time of optimizer (in sec)
StopCondition	scFprecision	Active stopping criteria
MaxCountOfEvals	100000* [2]	Upper bound for number of objective function's evaluations
MaxCountOfIters	10000	Upper bound for total number of iterations
MaxTimeOfWork	0.	Upper bound for total execution time (in sec)
UseCache	False*	If its value is True, caching of evaluated values of objective is used
VerboseLevel	0	Level of output from solver

also interfaces for some other optimization problems (specific dynamic optimization problems etc), but they weren't in use for a long time and now probably the most of them work unsatisfactory.

2.2.1 Parameter Estimation

Diana allows to identify parameters of models with fixed structure. It uses maximum likelihood approach and makes assumption that observation noise is white Gaussian, correlations between measurements are absent.The objective function (logarithmic maximum likelihood) has the form

$$L(x_0, p) = \sum_{i=1}^{n} \sum_{j=1}^{obs} \frac{(x^{(j)}(t_i; x_0, p) - x_i^{(j)})^2}{2\sigma_{ij}^2}, \tag{2}$$

where n is a total number of observations, obs is a number of observed state variables, $x_i^{(j)}$ and σ_{ij} denote respectively value and standard deviation of jth state variable observed at the moment t_i.

Because of some technical reasons Diana has two different classes for PE problem. The class ParameterFittingTask serves for PE with derivative free optimizers. It doesn't allow to evaluate gradient of objective. If the gradient is needed by solver, the user should try class SensParameterFittingTask instead of ParameterFittingTask. SensParameterFittingTask calculates gradient of objective by means of sensitivities. Components of gradient with respect to estimated parameters are calculated by

$$\frac{\partial L(x_0, p)}{\partial p_k} = \sum_{i=1}^{n} \sum_{j=1}^{obs} \frac{\partial x^{(j)}(t_i; x_0, p)}{\partial p_k} \frac{x^{(j)}(t_i; x_0, p) - x_i^{(j)}}{\sigma_{ij}^2} \tag{3}$$

Components of gradient with respect to initial values of state variables are defined as

$$\frac{\partial L(x_0, p)}{\partial x0_k} = \sum_{i=1}^{n} \sum_{j=1}^{obs} \frac{\partial x^{(j)}(t_i; x_0, p)}{\partial x0_k} \frac{x^{(j)}(t_i; x_0, p) - x_i^{(j)}}{\sigma_{ij}^2} \tag{4}$$

Since calculation of sensitivities is computationally expensive and can lead to worse convergence properties for integrator, the user should avoid SensParameterFittingTask if optimizer doesn't really need gradient of objective as far as it possible.

Specification of parameter estimation problem in Diana includes several steps, namely

1. provide measurement data

2. specify parameters of model and initial values of state variables that should be estimated
3. create task and assign it with the solver

In order to provide measurement data the user should create instance of `DianaMeasuredData` class. Constructor `DianaMeasuredData` takes two arguments. The first one is a reference to the model, and the second one is a list of observed state variables.

```
md = diana.DianaMeasuredData(model, ["c_x", "c_s"])
```

The easiest way to define measurement data is to load it from the file by method `load`

```
md.load('measuremets.dat')
```

The file should contain lines of the following format

$$\underbrace{\texttt{0.00001}}_{time\ t_i}\ \underbrace{}_{tab}\ \underbrace{\texttt{0.00000001}}_{measurement\ x_i^{(1)}}\ \underbrace{}_{tab}\ \underbrace{\texttt{0.00000001}}_{std.\ diviation\ \sigma_{i1}}\ \underbrace{}_{tab}\ \ldots\ \underbrace{}_{tab}\ \underbrace{\texttt{0.00000001}}_{measurement\ x_i^{(obs)}}\ \underbrace{}_{tab}\ \underbrace{\texttt{0.00000001}}_{std.\ diviation\ \sigma_{i,obs}}$$

The user is able to add information about measurement directly in Python scripts. Suppose that we'd like to add measurements at the beginning. It can be done as follows

```
mpt = diana.CapeMeasuredPoint(md)
# specify time
mpt.SetTime(0.)
# specify measurement for "c_x"
# standard deviation is .01 and measured value is 0.
mpt.SetValue(0, diana.CapeMeasuredValue(0.,.01))
# specify measurement for "c_s"
# standard deviation is .01 and measured value is 1.
mpt.SetValue(1, diana.CapeMeasuredValue(1.,.01))
# append information to measurement data object
md.AddMeasuredPoint(mpt)
```

If standard deviation is equal for all measured state variables, the simplified syntax can be used for the same purpose.

```
md.AddMeasuredPoint(1.5,     # time
                    0.01,    # standard deviation
                    [0.,1.]) # set of measurements
```

The user can specify estimated parameters and initial values in the lists of unknowns as it is mentioned in the section 2.1.4. Specifications for parameters and initial values should be represented in the separate lists and then these lists should be used in the constructor of the problem

```
sps = [
    diana.DianaNLPRealParameterSpec("k_s", "", 1.0, 1.0, 3.0, 0.0001),
    diana.DianaNLPRealParameterSpec("mu_max", "",  4.0, 4.0, 6.0, 0.0001)
    ]

taskPE = diana.ParameterFittingTask(dmain, 'ida',
                                    model, md, sps)
```

Constructor of `ParameterFittingTask` usually takes the following arguments

- reference to the `DianaMain` instance
- name of the DAE/ODE solver that will be used to integrate model equations
- reference to the model

- reference to the measurement data
- lists of specifications for estimated parameters and initial values (latter can be omitted)

In the case of `SensParameterFittingTask` the user has alternative way to specify estimated parameters and initial values. This class has several additional methods for it

- `AddEstimatedParameterBySpec`
  It takes two arguments, namely specification of estimated variable and its type (`DIANA_PARAMETER` or `DIANA_INITSTATE`).
- `AddEstimatedParameterByName`
  It takes name of estimated variable and its type (`DIANA_PARAMETER` or `DIANA_INITSTATE`) as arguments. This method uses name of parameter (state variable) to extract specification from model.
- `AddEstimatedParameterByIndex`
  The same method to the previous one, but it uses index of estimated variable in the list of model parameters (state variables) instead of name.

Obviously, in this case specification of estimated variables should follow creating the PE problem and lists of estimated variables are not included in the constructor's arguments.

```
taskPE = diana.SensParameterFittingTask(dmain, 'ida', model, md)

taskPE.AddEstimatedParameterBySpec(
    diana.DianaNLPRealParameterSpec("mu_max", "", 4.0, 4.0, 6.0, 0.0001),
    diana.DIANA_PARAMETER)
taskPE.AddEstimatedParameterBySpec(
    diana.DianaNLPRealParameterSpec("k_s", "", 3.0, 1.0, 3.0, 0.0001),
    diana.DIANA_PARAMETER)
```

Once PE problem is created and all estimated variables are specified the user can assign the problem with the optimizer and initialize it. After that the problem can be solved

```
solverPE.SetNLPTask(taskPE)
solverPE.Initialize()
solverPE.Solve()
```

2.2.2 Experimental Design

Diana provides common interface for experimental design (OED) problems. It is realized in the class `DianaOEDTask`. Note, `DianaOEDTask` gives only common interface, the final implementation should be realized in the `so`-library that uses this interface. Current version of Diana includes such `so`-library (`oedsigmapoint.so`) for Sigma Point approach (see [7]). If the user would like to implement his own OED method, he can use `DianaOEDTask` as a base class for inheritance.

Specification of OED problem (for problems inherited from `DianaOEDTask`) includes such steps

1. specify PE problem and optimizer for PE problem (see section 2.2.1)
2. specify list of design variables (controlled parameters that should be optimized)
3. load OED task
4. set parameters of the task
5. initialize OED task, assign it with the solver and start solving

Table 3: Optimality criteria for OED problems

Name in Diana	Description
DIANA_OED_A	A-optimal design
DIANA_OED_D	D-optimal design
DIANA_OED_M	M-optimal design
DIANA_OED_E	E-optimal design
DIANA_OED_EStar	E^*-optimal design

The questions concerned with the first and the second steps are considered in the sections 2.2.1, 2.1.1, 2.1.4.

Since final implementations of OED problems are held in `so`-libraries, the creating OED tasks is similar to the loading models and solvers. The user should call method `CreateOEDTask` of the factory for optimization problems:

```
nlptaskfactory = dmain.GetNLPTaskFactory()
taskOED = nlptaskfactory.CreateOEDTask(taskPE, optimizerPE,
                                       colDesignVars, 'oedsigmapoint')
```

This method takes four arguments:

- reference to the PE problem
- reference to the solver for PE problem
- list of design variables
- name of `so`-library which implements OED method

After creating OED task it can be tuned to the needs of the user. Below you see an example how to do it

```
pars_oed = taskOED.GetParameters()
pars_oed["OptimalityCriterion"].SetValue(diana.DIANA_OED_EStar)
pars_oed["ExecException"].SetValue(True)
pars_oed["Alpha"].SetValue(0.5)
```

Class `CreateOEDTask` gives only one extra-parameter called `OptimalityCriterion`. It defines cost function over the covariance matrix C_θ. The list of possible values for `OptimalityCriterion` is represented in the table 3. D-criterion is used by default. Class for Sigma Point approach from `oedsigmapoint.so` has a set of specific parameters listed in the table 4.

Finally, after all these preparations the user can assign the OED problem with the solver, initialize it and run optimization.

```
optimizerOED.SetNLPTask(taskOED)
optimizerOED.Initialize()
optimizerOED.Solve()
```

2.2.3 Optimization of Arbitrary Scalar Objectives

Diana offers the possibility to solve the general NLP problem

$$\begin{cases} \min_{\vec{p}^L \le \vec{p} \le \vec{p}^U} f(\vec{x}, \vec{p}) \\ \text{subject to} \quad \vec{h}(\vec{x}, \vec{p}) = 0 \\ \qquad\qquad\quad \vec{g}(\vec{x}, \vec{p}) \le 0 \end{cases} \tag{5}$$

Table 4: Specific parameters for Sigma Point approach

Parameter	Default value	Description
`ExecException`	`False`	Throw exception if objective cannot be calculated
`Alpha`	`1.`	Scaling parameter α
`Beta`	`2.`	Scaling parameter β
`Kappa`	`1.`	Scaling parameter κ
`CowMatr0Weight`	**OUT**	Weight of central sigma point for covariance matrix calculation w_0^c
`Mean0Weight`	**OUT**	Weight of central sigma point for mean calculation w_0^m
`Weights`	**OUT**	Weight of non-central sigma points for mean and covariance matrix calculation w_i

Class `BasicNLPTask` realizes this functionality. One may use this class also to perform dynamic optimization (e.g., see appendix A.4).

In order to solve NLP problem the user should

1. formulate objective function and constraint functions (see section 2.1.3)
2. specify list of unknowns (see section 2.1.4)
3. create the instance of `BasicNLPTask`, assign objective and constraints with it.
4. initialize the task, assign it with the solver and run optimization

The class has only one argument, namely list of unknowns.

```
task=diana.BasicNLPTask(sp)
```

The objective and constraints should be assigned with the task by means of methods `SetObjFunction` and `AddConstraintFunc`. The second argument of `AddConstraintFunc` specifies type of constraint. If it is `ctGx` the first argument of `AddConstraintFunc` is inequality constraint, otherwise equality.

```
task.SetObjFunction(objfunc_Rosenbrock)
task.AddConstraintFunc(constrfunc1, diana.ctGx) # inequality constraint
task.AddConstraintFunc(constrfunc2, diana.ctHx) # equality constraint
```

Task must be initialized and assigned with the solver before running optimization.

```
task.Initialize()
solver.SetNLPTask(task)

solver.Initialize()
solver.Solve()
```

2.3 Possible Problems

Lots of problems may occur during optimization. Some of the widespread are mentioned below.

1. optimizer fails
   Try to tune parameters of optimizer or use another one. If it does not help, tighten boundaries for unknowns.

Table 5: Available optimizers

Feature	so-library								
	genetic	direct	bbowda	bayes1	unt	lbayes	nmsimplex	lbfgsb	ipopt
Global search	✓	✓	✓	✓	✓	–	–	–	–
Stochastic objectives	✓	✓	✓	✓	✓	✓	–	–	–
Need of gradient	–	–	–	–	–	–	–	✓	✓
Support of constraints handling									
bound constraints	✓	✓	✓	✓	✓	✓	–	✓	✓
direct constraints	✓	✓	–	–	–	–	–	–	–
inequality constraints	✓	✓	✓	–	–	–	–	–	✓
equality constraints	–	–	✓	–	–	–	–	–	✓

2. integrator fails
   Try to tune parameters of integrator. If it does not help, try another DE solver. On my knowledge `'ida'` gives the best results for DAEs, and `'odessa'` is good for ODEs. If you use `SensParameterFittingTask`, changing it on `ParameterFittingTask` may help. You can also try to use optimizer that treats direct constraints (see table 5). Finally you can tighten boundaries for unknowns in order to exclude infeasible points where integrator fails.

3. starting point cannot be improved by optimizer
   There are lots of reasons that can cause this problem. If you use some kind of local optimizer (especially gradient-based solver) the likeliest one is that approximation to the gradient of the objective function is nearly zero in the chosen starting point. Find a way how to achieve better accuracy in gradient estimation. If it is impossible or very hard, try to perform sensitivity analysis in order to find better starting point.

4. problems with transferring data into `gnuplot`
   It happens very rare. In this case you should avoid reporting interfaces and organize collecting and saving results yourself.

3 Available Software Packages

Table 5 gives insight into capabilities of available in Diana optimizers. The reminder of section contains overview of solvers in more datails.

3.1 Global Nonlinear Continuous Optimization

3.1.1 Global Random Search

Diana contains quite flexible implementation of Genetic algorithms (GA) called GENETIC. This implementation includes several useful add-in properties, namely:

- treatment of direct constraints (by means of simple death penalty method)
- a priory self-tuning of randomization parameters depending on problem

Table 6: Parameters of DIRECT V2.0.4 solver

Parameter	Default value	Description
`Infinity`	`1.e+100`	Infinity
`Eps`	`1.e-4`	Exceeding value (if `eps` > 0, we use the same epsilon for all iterations; if `eps` < 0, we use the update formula from Jones)
`Algmethod`	`true`	Type of DIRECT method realization (if it is `false`, original algorithm by D. R. Jones is used; if it is `true`, algorithm with localization by J. M. Gablonsky is used)
`Ierror`	**OUT**	Error flag (if `Ierror` is lower than 0, an error occurred)
		Stopping criteria
`Maxf`	`10000`	The maximum number of objective function evaluations
`MaxT`	`600`	The maximum number of iterations
`FGlobal`	`-1.e+100`	Supposed lower bound for the global optimum. If this value is a good approximation to global minimum, algorithm converges faster
`FGlPer`	`0.`	The percent error to terminate the optimization
`VolPer`	`0.`	The volume of the hyperrectangle to terminate the optimization
`SigmaPer`	`0.`	The measure of the hyperrectangle S with $f(c(S)) = f_{min}$ to terminate the optimization

GAs represent a family of global adaptive generational random search algorithms. These algorithms are insensitive to small noise in objective function, but in average they shows logarithmic convergence rate even for simple problems (such as quadratic function minimization) [8, 9]. Therefore the users have to avoid GAs and global random search at all, if calculation of objective and/or constraints takes a lot of time (depending on problem values from several milliseconds to several seconds are critical), which is typical for PE, OED and some other dynamic optimization problems. In this case GAs cannot produce reliable solution in an acceptable time.

3.1.2 Statistical Methods

DIRECT V2.0.4 is the most useful statistical optimization package that is interfaced with Diana. It implements dividing rectangular global optimization method [1, 2]. The implementation supports:

- treatment of direct constraints
- treatment of inequality constraints
- two different algorithms: the first one emphasizes on the global search (D. R. Jones algorithm), and the second one has stronger local optimization properties (J. Gablonsky algorithm)

Package GMFL contains poor Fortran implementations of several statistical optimizers. Diana has interfaces for three of them (see [4]):

- routine BAYES1
  Bayesian global optimization method
- routine UNT
  The global method of extrapolation type by A. Zilinskas
- routine LBAYES
  The local Bayesian method by J. Mockus

Table 7: Parameters of solvers from GMFL package

Parameter	Default value	Description
IprPeriod	1	Period of values printing
M	100	The maximum total number of function evaluations
LT	10/30	The number of initial random points which are uniformly distributed by LP sequences of Sobol' (1969) ($0 < LT \leq m$)
		Specific for unt **routine**
ML	1	The maximal number of local minima, $0 < ML <= 20$
		Specific for lbayes **routine**
NIPA	0	The number of integer variables
ANIU	0.01	The rate of decreasing of the differentiation step
BETA	-1	The rate of decreasing of the iteration step

Table 8: Parameters of NMSIMPLEX solver

Parameter	Default value	Description
Step	–	Initial step size (equal for all unknowns)
Steps	–	Array of initial step sizes
MaxFn	100	The maximum number of function evaluations allowed
StopCr	1.e-4	Stopping criterion
NLoop	20	The stopping rule is applied after every NLoop function evaluations
IQuad	true	Flag for fitting of a quadratic surface
Simp	1.e-6	Criterion for expanding the simplex to overcome rounding errors before fitting the quadratic surface
Var	**OUT**	Contains the diagonal elements of the inverse of the information matrix
IFault	**OUT**	return of MINIM routine

The use of these routines is reasonable only if objective function is computationally extra-expensive and one cannot perform more than 40-60 evaluations of objective.

3.2 Local Nonlinear Continuous Optimization

3.2.1 Derivative Free Methods

Package NMSIMPLEX realizes Nelder-Mead downhill simplex method [5]. The implementation is performed in Fortran 95 by D. E. Shaw. NMSIMPLEX has a set of parameters listed in the table 8.

3.2.2 Quasi-Newton Methods

IPOPT (short for "**I**nterior **P**oint **Opt**imizer") is a powerful free for charge package for large scale local continuous nonlinear optimization. It is a part of COIN-OR project. The most of code has been written by Andreas Wächter and Carl Laird (Carnegie Mellon University). IPOPT library is distributed under Common Public License (CPL). IPOPT supports:

- interfacing with C/Fortran routines and C++ classes
- treatment of sparse Jacobians and Hessians
- Hessian approximation using a BFGS update (dense as well as limited memory)

Table 9: Parameters of L-BFGS-B solver

Parameter	Default value	Description
`MaxCorrections`	`10`	The maximum number of variable metric corrections used to define the limited memory matrix
`Factr`	`1.e+7`	Tolerance of function in the stopping criteria κ_f
`PGradTol`	`1.e-5`	Tolerance of the projected gradient in the stopping criteria
`TotalExploredIntervals`	**OUT**	Total number of intervals explored in the search of Cauchy points
`TotalSkippedUpdates`	**OUT**	Total number of skipped BFGS updates before the current iteration
`PriorUpdates`	**OUT**	Total number of BFGS updates prior the current iteration
`PGradNormInf`	**OUT**	Infinity norm of the projected gradient
`EvPrecision`	**OUT**	Evaluated precision ($\kappa_f \cdot \varepsilon_{mch}$)
`EpsMch`	**OUT**	Machine precision ε_{mch} generated by the code
`TimeCauchyPoints`	**OUT**	Accumulated time spent on searching for Cauchy points
`TimeSubspaceMinimization`	**OUT**	Accumulated time spent on subspace minimization
`TimeLineSearch`	**OUT**	Accumulated time spent on line search

- solving problems with equilibrium constraints (this feature has been implemented by Arvind Raghunathan; it allows to solve some MINLP problems)

Majority of above-mentioned features are available in Diana and can be tuned by means of solvers parameters. Description of the most useful IPOPT parameters is presented in [3]. Note, IPOPT has to be compiled together with at least BLAS, LAPACK, ASL and MUMPS third party libraries for successful integration with Diana.

L-BFGS-B package implements limited-memory variant of Broyden-Fletcher-Goldfarb-Shanno (BFGS) method subject to simple bounds on the variables. It intended to solve large nonlinear bound-constraint optimization problems if objective's gradient is known but information on the Hessian matrix is difficult to obtain. It can also be used for unconstrained problems. The algorithm is implemented in Fortran 77 and was developed by C. Zhu and J. Nocedal (see [10]). This routine works quite well and if equality and inequality constraints are absent it is a good (and rather simple) alternative for IPOPT.

4 Introductory Notes for Developers

4.1 Optimization Framework

A number of people with different tastes were involved in the process of Diana optimization subsystem development. Therefore the codes of this subsystem are non-uniform (but just a little, don't worry). In this brief introduction for developers I'll try to catch and highlight main ideas, and to prove that these codes are not so chaotic as it seems for some beginners.

The kernel of Diana optimization subsystem corresponds to the simplified UML-diagram on the Figure 1.

The core of optimization subsystem is formed from two groups of classes. The first one corresponds to solvers (optimizers), and the second one describes optimization problems. In order to create optimization problems or solvers one should use factories. The factory `NumericNLPTaskFactory` allows to create optimization problems. The factory `NumericNLPTaskFactory` can be used to allocate optimizers.

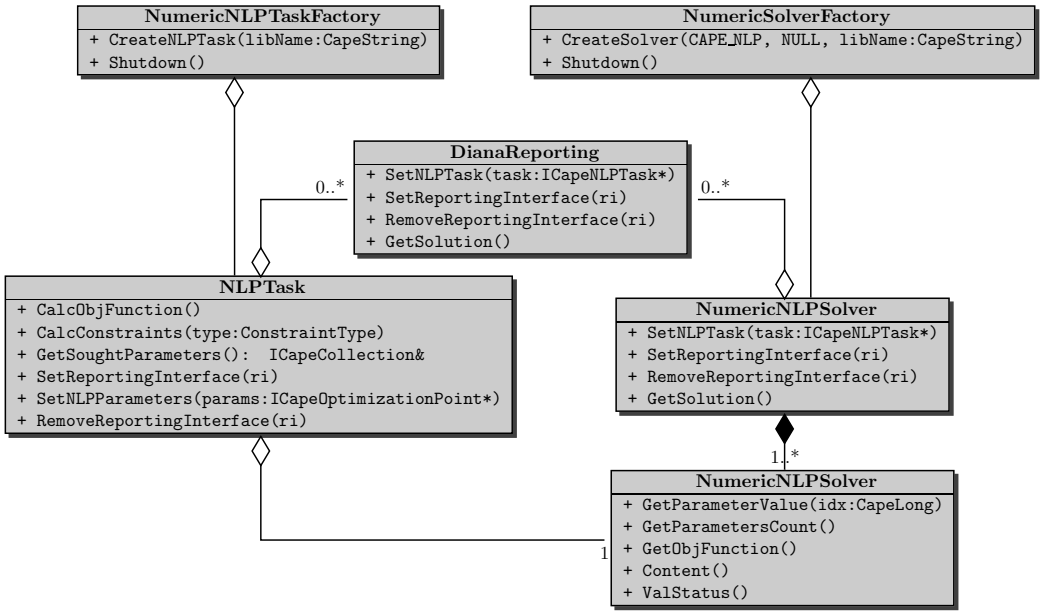

Figure 1: Kernel of Diana optimization subsystem

Classes that end on `NLPTask-` hold information about optimization problems. They provide interfaces for calculation of objective and constraints in the given sample points. Optimizers (usually end on `NumericNLPSolver-`) perform solving of optimization problems.

Each optimization problems `NLPTask` or optimizer `NumericNLPSolver` can contain reporting interfaces. Reporting interfaces allow to gather information about solution and optimization process.

The interaction between problems and optimizers is performed by means of `OptimizationPoint` class. Instance of `OptimizationPoint` contains information about sample point such as its coordinates and some computationally expensive data concerned with optimization problem in this point (usually value of objective function and values of some expensive constraints). It allows easily to organize caches of sample points and to omit recalculation of computationally expensive data in samples with the same coordinates. During optimization process optimizer initializes coordinates of its sample points, assigns this points with problem and calls methods of `NLPTask`.

4.2 Further Work

Some important directions of further work are listed below.

1. support of least square optimization
   Unfortunately till yet Diana doesn't support any interfaces for least square optimization problems. But this support is important to provide a proper numerical solution of least squares curve fitting problems with efficient software that implements Gauss-Newton and Levenberg-Marquardt algorithms.

2. implementation of some important optimization problems
   Current facilities for PE and OED are quite restrictive in Diana. Again Diana has not convenient interfaces for support another optimization problems that are important in analysis of dynamic systems.

3. benchmarks for constraints treatment

4. automatic differentiation in order to obtain gradients of objectives and constraints or just availability of explicit specification of gradients in Python code
   Unfortunately, now the users of Diana's Python interface have not possibility even to specify gradients directly in Python code. The gradients for gradient-based solvers are estimated by finite differences.

5. parallelization of computationally expansive problems

6. support of complete and/or rigorous optimization by means of interval arithmetics [6]

Sure, this list is incomplete. Here are mentioned only things that can (and should) be done in the near future. "Global" directions of further work such as multiobjective, combinatorial, mixed-integer optimization are omitted.

APPENDIX

A Scripts for Diana optimization subsystem

All scripts in this section deal with a simple unstructured growth model for a continuous stirred tank bio-reactor [7].

A.1 Optimization of "explicit" Function

The following script aims to find minimum of generalized Rosenbrock's function (1) for $\vec{x} \in [-100, 100]^n$.

```
# ----------------------------------------------------------------------
#  Diana process modelling, simulation and analysis software
#  Copyright (c) 2006, Sergiy Gogolenko
#  e-mail: gogolenk@mpi-magdeburg.mpg.de
#  All rights reserved.
# ----------------------------------------------------------------------
#  $Id: rosenbrock.py Fri Aug 11 13:38:55 2006 gogolenk $
# ----------------------------------------------------------------------
#  Written under: i586-suse-linux
# ----------------------------------------------------------------------
#  Last update: Fri Aug 11 13:38:55 2006
# ----------------------------------------------------------------------
#  Description:
#   Simple example of using DIANA in order to find minimum of
#  Rosenbrocks' function (10 unknowns) with deterministic
#  optimizer (in this case LBFGS-solver).

__author__ = "Sergiy Gogolenko (gogolenk@mpi-magdeburg.mpg.de)"
__version__ = "@PACKAGE_VERSION@"
__date__ = "$Date: Fri Aug 11 13:38:55 2006 $"
__copyright__ = "Copyright (c) 2006 Sergiy Gogolenko"
__license__ = ""

import diana, sys
from math import *

optimizerName = "lbfgsb" # "direct"

# initialize Diana main class
dmain = diana.GetDianaMain(sys.argv)

# create LBFGSB-optimizer
sfactory = dmain.GetSolverFactory()
solver = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerName)

# specify list of sought parameters
sp = []
for i in range(10):
    sp.append(diana.DianaNLPRealParameterSpec
              ("x"+str(i+1), "x"+str(i+1), 2, -512, 512, 0.001))

# create task
task=diana.BasicNLPTask(sp)

# define objective
def objfunc_Rosenbrock(task):
```

```
    pars = task.GetNLPParameters()
    par = pars.GetParameterValue(0)
    objfunc = 0.
    for i in xrange(pars.GetParametersCount()-1):
        par_next = pars.GetParameterValue(i+1)
        objfunc += 100*((par_next - par**2)**2) + ((par-1)**2)
        par = par_next
    return objfunc

# assign objective with task
task.SetObjFunction(objfunc_Rosenbrock)
task.Initialize()
print "Rosenbrock task initialized"

# create report
report = dmain.CreateReportingInterface(optimizerName)

# set parameters for reporting
pars_rep = report.GetParameters()
pars_rep["Author"].SetValue("MG")
pars_rep["OutputFolder"].SetValue("./report")
pars_rep["ConsoleOutput"].SetValue(True)

report.Initialize()
print "Report initialized"

# assign prepared task and report to solver
solver.SetReportingInterface(report)
solver.SetNLPTask(task)
solver.Initialize()
print "LBFGSB optimizer initialized"

# start optimization
print "Rosenbrock function (%i parameters) optimization..." \
      % (task.GetSoughtParameters().Count())
try:
    solver.Solve()
except:
    print "Solving error"
pars_opt = solver.GetParameters()
print '#'*80
print "Found minimum:                    ", \
      pars_opt["MinObjFunc"].GetValue()
print "Total amount of objective calls: ", \
      pars_opt["TotalEvalsCount"].GetValue()
print "Total amount of iterations:      ", \
      pars_opt["CurrIteration"].GetValue()
print "Total execution time:             %s sec" % \
      pars_opt["TotalTimeOfWork"].GetValue()
print '#'*80
print "Solution: ", solver.GetSolution()
print '#'*80
```

A.2 Parameter Estimation

Parameter estimation with derivative free solver

```
# ------------------------------------------------------------------------
```

```
#  Diana process modelling, simulation and analysis software
#  Copyright (c) 2006, Sergiy Gogolenko
#  e-mail: gogolenk@mpi-magdeburg.mpg.de
#  All rights reserved.
# ----------------------------------------------------------------------
#  $Id: HafkeReactor.py Fri Aug 11 13:22:08 2006 gogolenk $
# ----------------------------------------------------------------------
#  Written under: i586-suse-linux
# ----------------------------------------------------------------------
#  Last update: Fri Aug 11 13:22:08 2006
# ----------------------------------------------------------------------
#  Description:
#   Simple example of using DIANA for solving parameter fitting problems
#  with deterministic optimizer (in this case LBFGS-solver).

import diana, sys, os

modelPath = "Substrateuptake.so"
observedParams = ["c_x", "c_s"]
fileMesData = "./data/observations.dat"
integratorName = "ida" # "odessa"
optimizerName = "direct" # "nmsimplex"

# initialize Diana main class
dmain=diana.GetDianaMain(sys.argv)

# create NMSimplex-optimizer
sfactory = dmain.GetSolverFactory()
solverPE = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerName)

# create model
mmanager=dmain.GetModelManager()
model = mmanager.CreateModel(diana.CAPE_CONTINUOUS, modelPath)
model.Initialize()

# load measured data (state values and time points) from specified file
md = diana.DianaMeasuredData(model, observedParams)
md.load(fileMesData)

# Describe estimated parameters
sps = [
    diana.DianaNLPRealParameterSpec("k_s",  # name
                                    "",     # description
                                    1.0,    # default value
                                    1.0,    # lower bound
                                    3.0,    # upper bound
                                    0.0001),# precision
    diana.DianaNLPRealParameterSpec("mu_max", "",  4.0, 4.0, 6.0, 0.0001)
    ]

# create paramerter fitting task
taskPE = diana.ParameterFittingTask(dmain, integratorName,
                                    model, md, sps)
taskPE.Initialize()

print "Parameter fitting task for %s is initialized" \
      % model.GetComponentName()

# create report
```

```
report = dmain.CreateReportingInterface(optimizerName)
# set parameters for reporting
pars_solverPErep = report.GetParameters()
pars_solverPErep["Author"].SetValue("SG")
pars_solverPErep["OutputFolder"].SetValue("./parfit")
pars_solverPErep["ConsoleOutput"].SetValue(True)
pars_solverPErep["Samples"].SetValue(True)
report.Initialize()
print "Report initialized"

# set prepared task and report to solver
solverPE.SetReportingInterface(report);
solverPE.SetNLPTask(taskPE);
solverPE.Initialize();
print "Optimizer %s initialized" % solverPE.GetComponentName()

# start optimization
print "Parameter fitting for %s..." % model.GetComponentName()
try:
    solverPE.Solve()
except:
    print "Best solution: ", solverPE.GetSolution()
    raise
print "Solution: ", solverPE.GetSolution()
print "Parameter values:"
print taskPE.GetSoughtParameters()
```

Parameter estimation with gradient-based solver

```
# ----------------------------------------------------------------------
#  Diana process modelling, simulation and analysis software
#  Copyright (c) 2006, Sergiy Gogolenko
#  e-mail: gogolenk@mpi-magdeburg.mpg.de
#  All rights reserved.
# ----------------------------------------------------------------------
#  $Id: HafkeReactor.py Fri Aug 11 13:22:08 2006 gogolenk $
# ----------------------------------------------------------------------
#  Written under: i586-suse-linux
# ----------------------------------------------------------------------
#  Last update: Fri Aug 11 13:22:08 2006
# ----------------------------------------------------------------------
#  Description:
#   Simple example of using DIANA for solving parmeter fitting problems
#  with deterministic optimizer (in this case LBFGS-solver).

import diana, sys, os

modelPath = "Substrateuptake.so"
observedParams = ["c_x", "c_s"]
fileMesData = "./data/observations.dat"
integratorName = "ida" # "odessa"
optimizerName = "lbfgsb" # "ipopt"

# initialize Diana main class
dmain=diana.GetDianaMain(sys.argv);

# create NMSimplex-optimizer
sfactory = dmain.GetSolverFactory();
solver = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerName);
```

```
31
32 # set solver's parameters L-BFGS-B
33 pars_PEsolver = solver.GetParameters()
34 pars_PEsolver["StopCondition"].SetValue(1+8)
35 pars_PEsolver["MaxTimeOfWork"].SetValue(200)
36 pars_PEsolver["MaxCountOfEvals"].SetValue(1000)
37 pars_PEsolver["MaxCorrections"].SetValue(100)
38 pars_PEsolver["VerboseLevel"].SetValue(0)
39 pars_PEsolver["Factr"].SetValue(1e+7)
40 pars_PEsolver["PGradTol"].SetValue(1e-5)
41
42 # create model
43 mmanager=dmain.GetModelManager()
44 model = mmanager.CreateModel(diana.CAPE_CONTINUOUS, modelPath)
45 model.Initialize()
46
47 # load measured data (state values and time points) from specified file
48 md = diana.DianaMeasuredData(model, observedParams)
49 md.load(fileMesData)
50
51 # create paramerter fitting task
52 taskPE = diana.SensParameterFittingTask(dmain, integratorName, model, md)
53
54 # describe estimated parameters
55 taskPE.AddEstimatedParameterBySpec(
56     diana.DianaNLPRealParameterSpec("mu_max", "", 4.0, 4.0, 6.0, 0.0001),
57     diana.DIANA_PARAMETER)
58 taskPE.AddEstimatedParameterBySpec(
59     diana.DianaNLPRealParameterSpec("k_s", "", 3.0, 1.0, 3.0, 0.0001),
60     diana.DIANA_PARAMETER)
61
62 taskPE.Initialize()
63 print "Parameter fitting task for %s is initialized" \
64       % model.GetComponentName()
65
66 # create report
67 report = dmain.CreateReportingInterface(optimizerName)
68 # Set parameters for report class
69 pars_rep = report.GetParameters()
70 pars_rep["Author"].SetValue("SG")
71 pars_rep["OutputFolder"].SetValue("./parfit")
72 pars_rep["ConsoleOutput"].SetValue(True)
73 pars_rep["Samples"].SetValue(True)
74
75 report.Initialize()
76 print "Report initialized"
77
78 # set prepeared task and report to solver
79 solver.SetReportingInterface(report);
80 solver.SetNLPTask(taskPE);
81 solver.Initialize();
82 print "Optimizer %s initialized" % solver.GetComponentName()
83
84 # start optimization
85 print "Parameter fitting for %s..." % model.GetComponentName()
86 try:
87     solver.Solve()
88 except:
89     print "Best solution: ", solver.GetSolution()
```

```
    raise

print "Solution: ", solver.GetSolution()
print "Parameter values:"
print taskPE.GetSoughtParameters()
```

A.3 Optimal Experimental Design

Experimental design with sigma point approach

```
# ----------------------------------------------------------------------
#  Diana process modelling, simulation and analysis software
#  Copyright (c) 2009, Sergiy Gogolenko
#  e-mail: gogolenk@mpi-magdeburg.mpg.de
#  All rights reserved.
# ----------------------------------------------------------------------
#  Description:
#   Simple example of using DIANA for solving parameter fitting problems
#  with deterministic optimizer (in this case LBFGS-solver).

import diana, sys, os
#from math import *

modelPath = "Substrateuptake.so"
observedParams = ["c_x", "c_s"]
fileMesData = "./data/observations.dat"
integratorName = "odessa" #"ida"
optimizerPEName = "nmsimplex"
methodOEDName = "oedsigmapoint"
optimizerOEDName = "direct" #"genetic" #"nmsimplex"

# initialize Diana main class
dmain=diana.GetDianaMain(sys.argv)

###############################################################################
#                           PI Definition Section                             #
###############################################################################

# create optimizer for PI
sfactory = dmain.GetSolverFactory()
optimizerPE = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerPEName)

# set optimizer's parameters
pars_PEsolver = optimizerPE.GetParameters()
pars_PEsolver["StopCondition"].SetValue(1+8)
pars_PEsolver["MaxTimeOfWork"].SetValue(10)
pars_PEsolver["MaxCountOfEvals"].SetValue(1)
pars_PEsolver["VerboseLevel"].SetValue(0)

pars_PEsolver["Step"].SetValue(0.01)
pars_PEsolver["MaxFn"].SetValue(40)
pars_PEsolver["StopCr"].SetValue(1e-2)
pars_PEsolver["NLoop"].SetValue(20)
pars_PEsolver["IQuad"].SetValue(True)
pars_PEsolver["Simp"].SetValue(1e-6)

# create model
mmanager = dmain.GetModelManager()
```

```
model = mmanager.CreateModel(diana.CAPE_CONTINUOUS, modelPath)
model.Initialize()
#print model.GetActiveESO().GetParameters()

# load measured data (state values and time points) from specified file
md = diana.DianaMeasuredData(model, observedParams)
md.load(fileMesData)

# describe estimated parameters
sps = [
    diana.DianaNLPRealParameterSpec("k_s", "", 1.0, 1.0, 3.0, 0.0001),
    diana.DianaNLPRealParameterSpec("mu_max", "",  4.0, 4.0, 6.0, 0.0001)
    ]

# create paramerter fitting task
taskPE = diana.ParameterFittingTask(dmain, integratorName, model, md, sps)
taskPE.Initialize()
print "Parameter fitting task "\
      "for %s is initialized" % model.GetComponentName()

# set prepeared PI task to PI optimizer
optimizerPE.SetNLPTask(taskPE)
optimizerPE.Initialize()
print "PI optimizer %s initialized" % optimizerPE.GetComponentName()

###############################################################################
#                           OED Definition Section                            #
###############################################################################
# describe design variables
colDesignVars = [
    diana.DianaNLPRealParameterSpec("q_0", "",  0.09, 0.07, 0.08, 0.0001)
    ]

# create OED problem
nlptaskfactory = dmain.GetNLPTaskFactory()
taskOED = nlptaskfactory.CreateOEDTask(taskPE, optimizerPE,
                                       colDesignVars, methodOEDName)
pars_OEDtask = taskOED.GetParameters()
pars_OEDtask["OptimalityCriterion"].SetValue(diana.DIANA_OED_EStar)
pars_OEDtask["ExecException"].SetValue(True)
pars_OEDtask["Alpha"].SetValue(0.5)
taskOED.Initialize()
print "OED task initialized..."

###############################################################################
#                      Create OED task reporting                              #
###############################################################################

# create reportOEDTask
reportOEDTask = dmain.CreateReportingInterface(methodOEDName)
# set parameters for reportOEDTask class
pars_taskOEDrep = reportOEDTask.GetParameters()
pars_taskOEDrep["OutputFolder"].SetValue("./oedtask")
pars_taskOEDrep["ConsoleOutput"].SetValue(True)
pars_taskOEDrep["SamplingPoints"].SetValue(True)

reportOEDTask.Initialize()

taskOED.SetReportingInterface(reportOEDTask)
```

```
print "OED task report initialized"

###############################################################################
#                            OED Solving Section                              #
###############################################################################

# create optimizer for OED problem
optimizerOED = sfactory.CreateSolver(diana.CAPE_NLP,
                                     None, optimizerOEDName)

# settings for OED optimizer
pars_OEDsolver = optimizerOED.GetParameters()
pars_OEDsolver["Eps"].SetValue(1.0000000000e-4)
pars_OEDsolver["Maxf"].SetValue(5000)
pars_OEDsolver["MaxT"].SetValue(1000)
pars_OEDsolver["Algmethod"].SetValue(False)
pars_OEDsolver["FGlPer"].SetValue(1.0000000000e-02)
pars_OEDsolver["VolPer"].SetValue(-1.0000000000e+00)
pars_OEDsolver["SigmaPer"].SetValue(-1.0000000000e+00)
pars_OEDsolver["FGlobal"].SetValue(-10.15320)
pars_OEDsolver["VerboseLevel"].SetValue(2)

# create report
report = dmain.CreateReportingInterface(optimizerOEDName)
# set parameters for report class
pars_solverOEDrep = report.GetParameters()
pars_solverOEDrep["Author"].SetValue("SG")
pars_solverOEDrep["OutputFolder"].SetValue("./oed")
pars_solverOEDrep["ConsoleOutput"].SetValue(True)
pars_solverOEDrep["Samples"].SetValue(True)

report.Initialize()
print "Report initialized"

# set prepeared task and report to OED optimizer
optimizerOED.SetReportingInterface(report)
optimizerOED.SetNLPTask(taskOED)
optimizerOED.Initialize()
print "OED optimizer %s initialized" % optimizerOED.GetComponentName()

# start optimization
print "OED by %s for %s..." % (taskOED.GetComponentName(),
                               model.GetComponentName())
try:
    optimizerOED.Solve()
except:
    print "Best solution: ", optimizerOED.GetSolution()
    raise
print "Solution: ", optimizerOED.GetSolution()
print "Parameter values:"
print task.GetSoughtParameters()
```

A.4 Dynamic Optimization

The following script shows haw to use arbitrary objectives in order to solve dynamic optimization problems

```
# ------------------------------------------------------------------------
```

```
#  Diana process modelling, simulation and analysis software
#  Copyright (c) 2008, Sergiy Gogolenko
#  e-mail: gogolenk@mpi-magdeburg.mpg.de
#  All rights reserved.
# ----------------------------------------------------------------------------
#  $Id: control.py Mon Mar 30 13:22:08 2008 gogolenk $
# ----------------------------------------------------------------------------
#  Written under: i586-suse-linux
# ----------------------------------------------------------------------------
#  Last update: Mon Mar 30 13:22:08 2008
# ----------------------------------------------------------------------------
#  Description:
#   Example of using DIANA for solving optimal control problems (just a way
#   how one may define "nonstandard" optimization probem in DIANA).

import diana, sys, os
#from math import *

modelPath = "Substrateuptake.so"
observedParams = ["c_x", "c_s"]
integratorName = "odessa" #"ida"
optimizerName = "direct" #"nmsimplex"

controled_param = "q_0"

cntTimePoints = 21; T0, Tend = 0., 5.
timePoints = [(T0+i*(Tend - T0)/(cntTimePoints - 1)) \
              for i in xrange(cntTimePoints)]



# initialize Diana main class
dmain=diana.GetDianaMain(sys.argv)

################################################################################
#                      Specify optimization problem                            #
################################################################################

# create model
mmanager=dmain.GetModelManager()
model = mmanager.CreateModel(diana.CAPE_CONTINUOUS, modelPath)
model.Initialize()

# receive ESO from model
eso = model.GetActiveESO()
esopar = eso.GetParameters()
esovars = eso.GetStateVariables()
# print esovars # esopar

# define a list of unknowns
spars = []
for i in xrange(len(timePoints) - 1):
    name = '%s[%02i]' % (controled_param, i)
    descr = 'Parameter "%s" at the moment t=[%.2f, %.2f)' \
            % (controled_param, timePoints[i], timePoints[i+1])
    lower_bound, upper_bound = 0., 1.
    default_val = lower_bound + \
                  (upper_bound - lower_bound)*i/(len(timePoints) - 2)
    precision = 0.0001
```

```
    spars.append(diana.DianaNLPRealParameterSpec(
        name, descr, default_val, lower_bound, upper_bound, precision))

# create integrator (ODE/DAE solver)
sfactory = dmain.GetSolverFactory()
integrator = sfactory.CreateSolver(diana.CAPE_DAE, model, integratorName)
integrator.Initialize()
intpar = integrator.GetParameters()

# define objective
saved_esovars = eso.GetAllVariables()
def objFunc(task):
    # prepares to integration
    pars = task.GetNLPParameters()
    eso.SetAllVariables(saved_esovars)
    intpar["Start"].SetValue(True)
    intpar["T0"].SetValue(timePoints[0])
    intpar["VerboseLevel"].SetValue(0)
    ret_val = 0.
    # step-wise integration
    for i in xrange(1, len(timePoints)):
        par = pars.GetParameterValue(i-1)
        esopar[controled_param].SetValue(par)
        intpar["Tend"].SetValue(timePoints[i])
        integrator.Solve()
        ret_val += (esovars["c_s"].GetValue() \
                    - esovars["c_x"].GetValue())**2
    return ret_val

# create optimization problem
opttask = diana.BasicNLPTask(spars)
opttask.SetObjFunction(objFunc)
opttask.Initialize()
print 'Optimal control task for "%s" is initialized' \
      % model.GetComponentName()

################################################################################
#                            Specify optimizer                                 #
################################################################################

# create optimizer
optimizer = sfactory.CreateSolver(diana.CAPE_NLP, None, optimizerName)

# set optimizer's parameters (just if you need)
solpar = optimizer.GetParameters()
solpar["Eps"].SetValue(1.0000000000e-4)
solpar["Maxf"].SetValue(5000)
solpar["MaxT"].SetValue(1000)
solpar["Algmethod"].SetValue(False)
solpar["FGlPer"].SetValue(1.0000000000e-02)
solpar["VolPer"].SetValue(-1.0000000000e+00)
solpar["SigmaPer"].SetValue(-1.0000000000e+00)
solpar["FGlobal"].SetValue(-10.15320)
solpar["VerboseLevel"].SetValue(2)

# create report and set its parameters (just if you need)
report = dmain.CreateReportingInterface(optimizerName)
reppar = report.GetParameters()
reppar["Author"].SetValue("SG")
```

```
reppar["OutputFolder"].SetValue("./control")
reppar["ConsoleOutput"].SetValue(True)
reppar["Samples"].SetValue(True)
report.Initialize()
print "Report is initialized"

# assign task and report with optimizer
optimizer.SetReportingInterface(report)
optimizer.SetNLPTask(opttask)
optimizer.Initialize()
print 'Optimizer "%s" is initialized' % optimizer.GetComponentName()

##############################################################################
#                         Solve the problem                                  #
##############################################################################
print "Oprimization progress..."
try:
    optimizer.Solve()
except:
    print "Best solution (found before exception): ",\
          optimizer.GetSolution()
    raise
print "Solution: ", optimizer.GetSolution()
print "Parameter values:"
print opttask.GetSoughtParameters()
```

References

[1] Daniel E. Finkel. DIRECT optimization algorithm user guide. Technical report, Center for Research in Scientific Computation, North Carolina State University, 2003.

[2] J. M. Gablonsky and C. T. Kelley. A locally-biased form of the direct algorithm. *J. of Global Optimization*, 21(1):27–37, 2001.

[3] Yoshiaki Kawajir, François Margot, and Andreas Wächter. Introduction to IPOPT: A tutorial for downloading, installing, and using IPOPT. Technical report, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA; Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2008.

[4] Jonas Mockus, William Eddy, and Gintaras Reklaitis. *Bayesian Heuristic Approach to Discrete and Global Optimization: Algorithms, Visualization, Software and Applications*. Ser. Nonconvex Optimization and Its Applications. Springer, 1996.

[5] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[6] Arnold Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13:271–369, 2004.

[7] R. Schenkendorf, A. Kremling, and M. Mangold. Optimal experimental design with the sigma point method. *IET systems biology*, 3(1):10–23, January 2009.

[8] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. Complex Adaptive Systems. The MIT Press, Cambridge, 1999-08-27.

[9] Anatoly Zhigljavsky and Antanas G. Zilinskas. *Stochastic Global Optimization*, volume 9 of *Ser. Springer Optimization and Its Applications*. Springer-Verlag, Berlin/Heidelberg, 2008.

[10] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, 1997.