Доклади на Българската академия на науките
Comptes rendus de l'Académie bulgare des Sciences

Tome 64, No 1, 2011

MATHEMATIQUES
Machines à calculer

EMBEDDING C-XSC NONLINEAR SOLVERS IN *MATHEMATICA*

Evgenija D. Popova, Walter Krämer

(*Submitted by Academician S. Dodunekov on October 23, 2010*)

Abstract

This work presents the integration of C-XSC nonlinear problem-solving modules based on automatic differentiation into *Mathematica*® via *MathLink* protocol.

ACM: G.4, D.2.12, D.2.13

Key words: interval software, interoperability, C-XSC, *Mathematica*, *MathLink*

1. Introduction. Providing interoperability between the general-purpose environments (like *Mathematica*, Maple, etc.), which possess several features not attributable to the compiled languages, and the interval software, developed in some compiled language for efficiency reasons, is highly desirable for modelling, design and simulation of real-life problems involving uncertainties. However, the diversity in the implementation supporting environments and in the interval data representations hampers the collaborative usage of the existing interval software. Communication based on file reading/writing operations to exchange ordinary text is not suitable for interval data due to the necessity of avoiding decimal-to-binary/binary-to-decimal input/output conversions of floating-point intervals. The usage of communication protocols for integrating environments with interval computations was first studied in [1]. The communication of numerical intervals and the embedding of external interval software, provided by the C++ library C-XSC [2,3], in the general purpose environment *Mathematica* [4] via the communication protocol *MathLink* [5] is discussed therein. The initial research [1] shows that the communication of floating-point intervals between *Mathematica*

This work was partially supported by the DFG grant GZ: KR1612/7-1, AOBJ: 570029 and the Bulgarian Science Fund grant: DO 02-359/2008.

and C-XSC is transparent but requires providing compatibility between the fundamental C data types and the specific C-XSC data types. However, the complete integration of all C-XSC features into *Mathematica* presents additional challenges.

C-XSC is a C++ class library supporting data objects and problem-solving modules for computing with uncertain (interval) data and providing validated numerical results [2,3]. It supplies some arithmetic tools that are not available in other interval libraries, such as accurate dot product expressions. C-XSC contains modules for isolating zeroes of nonlinear functions, guaranteed global optimization, and modules for automatic differentiation (AD) of interval functions. These, as well as many other problems are formulated in terms of nonlinear functions, respectively require the latter as input data. Therefore, the embedding in *Mathematica* of these C-XSC modules is an important step toward creating a general interface that allows using all application functions delivered with C-XSC from within *Mathematica*.

The present work summarizes a so-called `ADExpressions` software which communicates (via *MathLink* protocol) and provides compatibility between the representation of nonlinear functions specified as *Mathematica* expressions and objects of suitable classes supported by C-XSC AD modules [6]. Although based on *MathLink* protocol for communicating the expressions, this basic software does not contain functions installable in *Mathematica*. Its purpose is to facilitate the development of such functions, e.g., the implementation of external *MathLink* programmes that integrate C-XSC routines for, or routines based on C-XSC AD arithmetic. The goal of this work is to present in more detail the embedding of these C-XSC modules into *Mathematica* via *MathLink* and thus, to provide a framework for further developments.

2. C-XSC Modules. The computation of guaranteed enclosures of function values (ranges) is essential for all interval methods. Many algorithms with result verification need computing enclosures also of the values of function derivatives. C-XSC[1] contains three AD modules (Table 1).

The module `ddf_ari` implements AD arithmetic for twice continuously differentiable functions $f : \mathbb{R} \to \mathbb{R}$. It supports the class `DerivType` including operations and elementary functions for the differentiation arithmetic up to second order. An object of type `DerivType` is implemented as a triple of intervals representing the function and the derivatives values, respectively. To get enclosures for the true values, the differentiation arithmetic is based on interval arithmetic.

Any application programme using `ddf_ari` must contain a C++ function, with argument and result type `DerivType`, which specifies the functional expression. For example, a function $f(x) = x(4 + x)/(3 - x)$ should be specified in a C++ function as follows.

```
DerivType f(const DerivType& x) { return x*(4.0 + x)/(3.0 - x); }
```

[1] Formerly the CToolbox library [7].

Table 1

C-XSC modules for AD: supported data types and predefined routines

C-XSC module	`ddf_ari`	`grad_ari`	`hess_ari`
scalar func.	$\mathbb{R} \to \mathbb{R}$	$\mathbb{R}^n \to \mathbb{R}$	$\mathbb{R}^n \to \mathbb{R}$
object type	`DerivType`	`GradType`	`HessType`
arg. type	`DerivType`	`GTvector`	`HTvector`

predefined functions	`fValue(DerivType)`
	`dfValue(DerivType)`
	`ddfValue(DerivType)`
	`fEval()`
	`dfEval()`
	`ddfEval()`
		`fValue(GradType)`
		`gradValue(GradType)`
		
		`fEvalG()`
		`fgEvalG()`
			`fValue(HessType)`
			`gradValue(HessType)`
			`hessValue(HessType)`
			`fEvalH()`
			`fgEvalH()`
			`fghEvalH()`

vector func.		$\mathbb{R}^n \to \mathbb{R}^n$	$\mathbb{R}^n \to \mathbb{R}^n$
object type		`GTvector`	`HTvector`
arg. type		`GTvector`	`HTvector`

predefined functions		`fValue(GTvector)`
		`JacValue(GTvector)`
		`fEvalJ(), fJEvalJ()`
			`fValue(HTvector)`
			`JacValue(HTvector)`
			`fEvalJ(), fJEvalJ()`

Let `x, fx` be declared as `DerivType` variables and `y(123.0)` be an interval variable. The objects `x = DerivVar(y)` and `fx = f(x)` of class DerivType contain a variable with value y and the representation of $f(x)$ in the point y, respectively. Then, `fValue(fx), dfValue(fx), ddfValue(fx)` get the function and the derivative values in the point interval $y = 123.0$. The C-XSC functions `fEval(), dfEval(), ddfEval()` simplify the mechanism of function and derivative evaluation. Thus, `ddfEval(f, y, fy, dfy, ddfy)` evaluates the three enclosures.

The module `hess_ari` supports classes `HessType` and `HTvector`, as well as operators and elementary functions for interval differentiation arithmetic of gradients and Hessians of scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$. Computing Jacobians of vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}^n$ is also supported. The module `grad_ari` supports classes `GradType` and `GTvector` for computing only gradients of scalar-valued functions and Jacobians of vector-valued functions without the storage overhead for the Hessian components. Details can be found in [7].

The C-XSC problem-solving modules given in Table 2 are based on the AD modules. The usage of these problem-solving modules requires that the user defines C++ functions with corresponding variable and result types which specify the expression of the function in use [7]. These functions are compiled together with the corresponding main programme and the functional expression cannot be changed during the programme execution. One goal of the designed *Mathematica* interface to these modules is to bring dynamics and interactivity in their execution. The latter means dynamic execution for different functions without any precompilation.

3. Communication of functional expressions. At the core of *Mathematica* is the foundational idea that everything – data, programmes, formulas,

Table 2

C-XSC problem-solving modules using automatic differentiation arithmetic

Problem-solving module	`nlfzero`	`gop1`	`nlinsys`	`gop`
Description	zeroes of 1D functions	1D global optimization	zeroes of nD functions	nD global optimization
Used AD module	`ddf_ari`	`ddf_ari`	`grad_ari`	`hess_ari`
Data type	`DerivType`	`DerivType`	`GTVector`	`HessType`

graphics, documents – can be represented as symbolic expressions. A prototypical example of a *Mathematica* expression is `f[x,y,...]`. All expressions – whatever they may represent – ultimately have a uniform structure. The function `FullForm` gives the full functional form of an expression, without shortened syntax. E.g. `FullForm[1+x^2+x/y]` gives `Plus[1, Power[x,2], Times[x,Power[y,-1]]]`. Whenever *Mathematica* knows that a function is associative, it tries to remove parentheses (or nested invocations of the function) to get the function into a standard "flattened" form [4]. A function like addition is also commutative, which means that expressions with terms in different orders are equal. One reason for putting expressions into standard forms is that if two expressions are really in standard form, it is obvious whether or not they are equal. However, it is not always desirable that expressions be reduced to the same standard form. This is especially valid in interval arithmetic, where the machine addition and multiplication are not associative, and in range computation, where different forms of an expression can give different quality of the range enclosure. The *Mathematica* function `Hold[expr]` provides "wrappers" inside which the expressions remain unevaluated.

In order to provide communication and compatibility between the *Mathematica* representation of a nonlinear function and the corresponding C-XSC representation used by the AD modules a *MathLink* based C++ software, called `ADExpressions`, is designed and implemented [6]. It consists of two basic template classes:

- `MLCXSCFunctionScalar<class T>` which communicates, represents and evaluates one- or multi-variate scalar functions $f(x) : \mathbb{R} \longrightarrow \mathbb{R}$ or $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$;
- `MLCXSCFunctionVector<class T>` which communicates, represents and evaluates vector-valued functions $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}^n$.

The template parameter `T` in the class `MLCXSCFunctionScalar` can be one of the three C-XSC data types `DerivType`, `GradType`, or `HessType`, corresponding to the respective C-XSC AD modules, while the template parameter in the second class can be `GTVector` or `HTVector`. In the initialization of the above classes it is supposed that the functional expression is in the *MathLink* queue. The two basic template classes have the following four public methods:

- `Init()` reads the *Mathematica* expression via appropriate *MathLink* tools,

parses the expression providing compatibility between the two environments, and initializes an object, called representation list, with the functional expression which should be evaluated;

• `operator()` reads from *Mathematica* an interval value for each of the expression variables, substitutes the latter (in the internal expression representation) with objects of respective classes `DerivType`, ..., `HTVector`, where the objects are instantiated with the input values, and evaluates the expression according to its internal representation and by the respective C-XSC AD arithmetic. The result of the evaluation is an object of the appropriate C-XSC class;

• `IsInitialized()` gives `true` if the representation list is initialized, and `false` otherwise;

• `Invalidate()` makes a representation list not valid.

The overall work of this software is separated in two stages in order to provide re-usability of the evaluation step. A nonlinear function represented as a *Mathematica* expression may contain:

- numerical constants: integer, real, interval, the *Mathematica* constants `E`, `Pi`; the interval constants should be represented by the *Mathematica* object `Interval[{a, b}]`, where the interval end-points `a, b` can be integer or real numbers; exact singletons cannot be used at the interval end-points but can be used in the other parts of a functional expression; for simplicity, *Mathematica* numbers with heads `Rational` and `Complex`[2], or another syntax of the interval object are not admissible for the functional expression;
- variables having interval values; the names of the variables should follow the *Mathematica* convention for names, or be *Mathematica* indexed variables with the syntax `name[i]`.
- arithmetic operations, a set of elementary functions, and the *Mathematica* function `Hold` enlisted with more details in [6].

A class `MLCXSCFunctionException` is defined to facilitate catching and communicating error messages. The latter are defined in detail at that place of `ADExpressions` code at which they arise and are associated with a particular error code having the following enumerated values.

```
ecIllegalArgumentError // Init,operator called with illegal arguments
ecNotInitializedError  // calling operator() before executing Init()
ecExpressionError      // error in parsing a Mma expression by Init()
ecInternalError        // internal error that should not normally occur
```

Different error messages, regarding the same stage of the algorithm, are communicated by one error code. For example, all kinds of errors that arise during the expression parsing are associated with and communicated by the `ecExpressionError` code. Some of these messages are: `Unknown variable`, `Unknown function`, `Unknown Data Type`, etc.

[2] C-XSC AD modules do not support complex arithmetic.

For the sake of readability, `ADExpressions` is split into three files: `types.h`, `expression.h` and `expression.cpp` (cf. [6]). In order to use the software, `expression.h` must be included in the source file of an own *MathLink* programme, as it is discussed below.

4. C-XSC Modules in *Mathematica*. For the embedding in *Mathematica* of any *Mathematica* or C-XSC software based on the C-XSC automatic differentiation, `ADExpressions` must be included in a *MathLink* programme designed to provide the interface and implemented according to *MathLink* technology. We have designed and implemented as packages separate *MathLink* programmes for each of the C-XSC modules in Table 1 and one package embedding the problem solvers from Table 2. These are involved in the archives `ADpackages.zip`, `nlADproblems.zip` downloadable from `http://www.math.bas.bg/~epopova/software/`. Since these packages (which functions are enlisted in Table 3) are similar, we discuss the implementation aspects on `gradAri`.

Table 3

Mathematica packages interfacing the C-XSC AD modules

ddfAri	gradAri	hessAri
`SetFunction[var,func]`	`SetFScalarGrad[{vars}, func]`	`SetFScalarHess[{vars}, func]`
`ReadyFunctionQ[ ]`	`ReadyScalarGradQ[ ]`	`ReadyScalarHessQ[ ]`
`fValue[int]`	`fValueScalarG[{ints}]`	`fValueScalarH[{ints}]`
`dfValue[int]`	`gradValueG[{ints}]`	`gradValueH[{ints}]`
`ddfValue[int]`		`hessValue[{ints}]`
`dfEval[var,fun,int]`	`fgEvalG[{vars}, func, {ints}]`	`fgEvalH[{vars}, func, {ints}]`
`ddfEval[var,fun,int]`		`fghEvalH[{vars}, func, {ints}]`
	`SetFVectorGrad[{vars},{funcs}]`	`SetFVectorHess[{vars},{funcs}]`
	`ReadyVectorGradQ[ ]`	`ReadyVectorHessQ[ ]`
	`fValueVectorG[{ints}]`	`fValueVectorH[{ints}]`
	`JacValueG[{ints}]`	`JacValueH[{ints}]`
	`fJEvalJGrad[{vars}, {funcs}, {ints}]`	`fJEvalJHess[{vars}, {funcs}, {ints}]`

The distribution contains two files for each package, e.g. `gradAri.tm/cpp`. The corresponding executables should be built from these files and from `ADExpressions` following the normal way of building template-based external functions, see [4].

4.1. Implementation issues. The following steps constitute the development process of every *MathLink*-compatible programme: (1) Create a template file (cf. `gradAri.tm`) whose main purpose is to establish a correspondence between the functions that will be called from *Mathematica* and the respective external C++ functions that will call the C-XSC functions. (2) A corresponding commu-

nication module (`gradAri.cpp`) should be written in C++ to combine the template file with `ADExpressions` and the external C-XSC software. (3) Process the *MathLink* template information and compile all the source code. (4) Install the binary in the current *Mathematica* session. We assume that the reader is familiar with the basics of *MathLink* technology [4,5]. Some details about the connectivity between *Mathematica* and external interval programmes can be found also in [1].

The *Mathematica* package code is included in the template file `gradAri.tm`. Thus, when installing the *MathLink* connection in step (4) above, all the functions involved in the package will be ready for use. The basic design of the template file (resp. the package) is as follows. All the package code is put in `:Evaluate:` sections. Some of the functions visible to the *Mathematica* user (such as `ReadyScalarGradQ, ReadyVectorGradQ` and `SetFScalarGrad, SetFVector Grad`) are named in templates and map directly to functions in the communication module. For example, the external functions `ReadyScalarGradQ` and `ReadyVectorGradQ` are so simple that they do not need a *Mathematica* wrapping function. Part of the argument checking for `SetFScalarGrad` and `SetFVector Grad` is done in the *Mathematica* function specification line `:Pattern:` of the template, e.g. `:Pattern: SetFScalarGrad[varnames_List/; Length [varnames]>0, func_]`. The other functions that are visible to the user (such as `fValueScalarG, gradValueG, fgEvalG` and their vector-valued analogues) are written in *Mathematica*. They perform some error checking of the arguments. Then they call public functions (such as `Ready ScalarGradQ`, and `SetFScalar Grad`) and/or private functions (such as `CalcScalarGrad`), which are the ones named in templates, and map to functions in the communication module, e.g.

```
:Evaluate: gradValueG[vals:{__List}/;And@@(Length[#]==2&/@ vals)]:=
   Block[{res}, res/; (res=gradValueScalar[vals]; res=!=$Failed)]/;
ReadyScalarGradQ[]||(Message[MLCXSCErrorGrad::notrdy,msgScalarGrad];
False);
```

The communication module `gradAri.cpp` is written in C++ to combine the template file with `ADExpressions` library and the external C-XSC software. Thus, `gradAri.cpp` begins with `#include "mathlink.h"` and `#include "ADExpressions/expression.h"`. The C++ source file must involve function objects of C-XSC classes supported by the corresponding C-XSC module for automatic differentiation, `GradType` and `GTvector` in `grad_ari`. These C-XSC function objects are defined via the two template classes from `ADExpressions` which provide interactive specification of the particular functional expression. For example,

```
mlcxsc::MLCXSCFunctionScalar<GradType> scalarfunc_grad;
GradType sf_grad(const GTvector& x) { return scalarfunc_grad(x);  }
```

For each embedded C-XSC function, a C++ function, specified in the template file, reads the *Mathematica* generated data via variables of fundamental C++ data types, initializes new variables whose data types are specific for the particular C-XSC module and after invoking the actual C-XSC computations transforms the computed result into variables of fundamental C++ data types

that are passed back to *Mathematica*. All actions necessary for reading, parsing and storing the functional expressions are done by the `Init()` function member of the corresponding class from the basic communication software. However, the functions `SetFScalarGrad()` and `SetFVectorGrad()` must read by themselves a list of variable names involved in the functional expression. A separate auxiliary function `getVariable()`, which reads (via suitable *MathLink* functions), parses and stores a variable name, is provided in `gradAri.cpp`. The evaluation of function and/or derivative values is done via the corresponding C-XSC function. In order to illustrate the two possible evaluation methods discussed in [7], we have implemented the functions which do a single evaluation by using a variable of `GradType`, as in `gradValueScalar()`, while implementing the functions evaluating both the function and its derivatives we have used the straightforward method, as in `CalcScalarGrad()`, see the distribution.

An important aspect of *MathLink* programming is the error-checking, respectively triggering error messages. Some error messages are issued by the *Mathematica* code, but most errors can only be detected inside the functions in the external communication module or in the external C-XSC module. The error messages are communicated via the exception class `MLCXSCFunctionException` of `ADExpressions`. The source file `gradAri.cpp` contains a separate function `PrintErrorMessage()` which triggers the error messages from the exception class to a *Mathematica* session. This function implements *MathLink* programming discussed in [1]. Note that the message names, defined in the template files, are associated with dummy *Mathematica* symbols, e.g. `MLCXSCErrorGrad`. These symbols are different in the four implemented packages so that the latter be loaded and used simultaneously.

4.2. Using the packages. Once the external *MathLink* programmes have been compiled to produce executable files, the latter (containing the corresponding package) are ready to be launched in any *Mathematica* session. This is done with the *Mathematica* function `Install`. All four packages can be used simultaneously. The *Mathematica* function `Names` gives the names of all public symbols from the specified context and the online *Mathematica* help provides their syntax and purpose.

Now, we initialize a scalar function $f = 2 - \sum_{i=1}^{10} x_i^2$ in 10 variables and evaluate it in the interval $[-1, 1]$ for each variable. We use indexed variables and *Mathematica* functions for generating the functional expression and the lists of variable names and variable values.

```
In[5]:= SetFScalarGrad[Table[x[i],{i,10}], 2 - Sum[x[i]^2, {i,10}]]
        fValueScalarG[Table[{-1, 1}, {i, 10}]]
Out[6] = {-8., 2.}
```

The output `Null` of the successful evaluation of `SetFScalarGrad` is not visible. For the evaluation of vector-valued functions, C-XSC AD modules impose

a restriction followed by the interface: the number of specified variables must be equal to the function dimension.

```
In[7] := SetFVectorGrad[{x}, {x, 1 - x, x^2}]
MLCXSCErrorGrad::illargs : Illegal arguments: Init:The number
of variables must be equal to the dimension of the function.
Out[7] = $Failed
```

However, one variable name can be repeated many times in the list of variable names. The list of variable names facilitates the functional expression parsing, it is not used by the C-XSC modules. The evaluation stage uses the list of interval values for the initialization of `GTVariable` or `HTVariable` vectors. For a variable involved in the functional expression, the implementation takes the interval value corresponding to that variable and the correspondence is established by the variable number in the list. If a list of variable names involves more than one occurrence of the same variable name, the evaluation stage takes that interval from the list of variable values which corresponds to the first occurrence of the variable name in the list.

```
In[8]:= SetFVectorGrad[{x,x,y}, {x,y-1,x^2}]; JacValueG[Table[{-1,1},
{i,3}]]
Out[8]={{{1.,1.},{0.,0.},{0.,0.}},{{0.,0.},{0.,0.},{1.,1.}},{{-2.,2.},
{0.,0.},{0.,0.}}}
```

Dummy variables can also be used which makes possible the guaranteed evaluation of constants. The implementation of C-XSC AD modules uses the standard error handling (runtime error) of the interval library in case of some error during the function or the derivatives evaluation, cf. [7]. C-XSC runtime errors cannot be checked and cause closing the connection but the corresponding C-XSC message is displayed in a separate `stderr` window.

The *Mathematica* interface for C-XSC routines provides more possibilities for controlling the conversion input errors. Inexact quantities can be enclosed either by the *Mathematica* function `Interval`, or by an interval enclosure in C-XSC using the techniques discussed in [7] and wrapping in the `Hold` function. Evaluation of functions in different forms is also possible via the `Hold` function.

```
In[9]:= SetFScalarGrad[{x}, Hold[Divide[23,Interval[{10,10}]]]*x]
        fValueScalarG[{{1, 2}}] // FullForm
Out[10]//FullForm = List[2.3‘,4.6000000000000005‘]
```

Table 4 enlists the *Mathematica* functions interfacing the corresponding solvers from the C-XSC modules in Table 2. The names of the functions are identical in both environments.

Below we find all zeroes of the function $(3x-1)(10x-1)/30$ in the interval $[0,4]$ and use `InputForm` function to see the full precision of the enclosing intervals.

```
In[11]:=AllZeros[x,(3x-1)Hold[Divide[10x-1,30]],{0,4},1.*^-14]//InputForm
Out[11]={{0.09999999999999999, 0.1},{0.33333333333333326,
0.3333333333333334},{1,1}}
```

Table 4

Functions in the *Mathematica* package `nlADproblems`

`AllZeros`	`[varName, f, start, eps]`
`AllGOp1`	`[varName, f, start, eps]`
`AllNLSS`	`[{varName`$_1$`,...}, {f`$_1$`,...}, {start`$_1$`,...}, eps]`
`AllGOp`	`[{varName`$_1$`,...}, f, {start`$_1$`,...}, eps]`

The list `{1, 1}` in `Out[19]` contains information about the uniqueness (1) of the roots in the enclosing intervals. More examples for the use of functions from `nlADproblems` package can be found in a *Mathematica* notebook involved in the distribution of the package.

5. Conclusion. The embedding of C-XSC nonlinear problem solvers in *Mathematica* provides an integrated environment which combines the interactivity, symbolic and visualization tools of *Mathematica* with the rigorousness and the speed of C-XSC functionality. Because the communication between the two environments is based on *MathLink* protocol, round-off errors in transferring numerical data are avoided and C-XSC programme execution is dynamic without any precompilation. Providing syntactic and semantic interoperability between the two environments, the presented *MathLink* programmes involve a rather complicated interaction between the external C-XSC code, the parsing software communicating functional expressions, the *Mathematica* package code, and the *Mathematica* kernel. Therefore the present work may serve as a framework for future developments of new numerical methods based on the C-XSC automatic differentiation.

REFERENCES

[1] Popova E. Lect. Notes Comp. Sci., **5492**, 2009, 117–132.

[2] Klatte R., U. Kulisch, C. Lawo, M. Rauch, A. Wiethoff. C-XSC, A C++ Class Library for Extended Scientific Computing, Springer, Heidelberg, 1993.

[3] Hofschuster W., W. Krämer, M. Neher. Lect. Notes Comp. Sci., **5492**, 2009, 68–102.

[4] Wolfram Research Inc.: *Mathematica*, Version 5.2, Champaign, IL, 2005.

[5] Gayley T. A *MathLink* Tutorial. Wolfram Research, Champaign, IL, 2002.

[6] Popova E., W. Krämer. Lect. Notes Comp. Sci., **6327**, 2010, 354—365.

[7] Hammer R., M. Hocks, U. Kulisch, D. Ratz. C++ Toolbox for Verified Computing: Basic Numerical Problems, Springer, Heidelberg, 1995.

Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Acad. G. Bonchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: `epopova@bio.bas.bg`

**WRSWT, Bergische Universität Wuppertal*
Faculty of Mathematics and Natural Sciences
Gaußstr. 20
D-42097 Wuppertal, Germany,
e-mail: `kraemer@math.uni-wuppertal.de`