RUSSIAN ACADEMY OF SCIENCES
PETERSBURG NUCLEAR PHYSICS INSTITUTE

Preprint 2421

I. B. Smirnov

LikelihoodLib – Fitting, Function Maximization, and Numerical Analysis

Gatchina-2001

LikelihoodLib – Fitting, Function Maximization, and Numerical Analysis

I. B. Smirnov
e-mail: Igor.Smirnov@cern.ch

Abstract

A new class library is designed for function maximization, minimization, solution of equations and for other problems related to mathematical analysis of multi-parameter functions by numerical iterative methods. When we search the maximum or another special point of a function, we may change and fit all parameters simultaneously, sequentially, recursively, or by any combination of these methods. The discussion is focused on the first the most complicated method, although the others are also supported by the library. For this method we apply: control of precision by interval computations; the calculation of derivatives either by differential arithmetic, or by the method of finite differences with the step lengths which provide suppression of the influence of numerical noise; possible synchronization of the subjective function calls with minimization of the number of iterations; competitive application of various methods for step calculation, and converging to the solution by many trajectories.

1 Introduction

A fitting program is a prerequisite component of any analytical system for data treatment or analysis in any problem domain. The term "fitting" usually means an iterative procedure of adjustment of the parameters of a model which describes the data of any sort. We usually determine a function which estimates numerically how close the model fits the data. Then we search either the minimum or maximum of this function using ordinary methods of mathematical analysis, which usually give an approximate solution or just a hint regarding its position and are applied in the iterative procedures together with strategies of trials. A similar approach, an iterative procedure upon parameters of one or many functions aimed at satisfying some conditions regarding their values, is used for solution of simultaneous equations (see, for example [1]), as well as for many other problems. These functions do not necessarily represent a model. Nevertheless, by the methods of solution, the iterative adjustment of parameters, this is also the fitting in broad sense. In any such problem we deal with similar mathematical abstractions: imaginary objects and algorithms. In object-oriented programming these abstractions can be expressed via a set of foundation classes and functions, which can be utilized for solution of the most of such problems. The idea to compose such class library is especially attractive from the point of view of the scalability and re-use[2], because the tools designed for one particular problem and made available inside this framework may be eventually demanded by other problems.

It is supposed that the objective function or functions will be determined by the user of this library in the form of the programming function which returns their values and optionally the partial derivatives. The program knows nothing more about the objective function except that the program can request the function value for any allowed point in the parameter space. This determines the principal difference between approaches to one-parameter problems, for which this point information usually allows to determine a closed region, an interval, where the solution resides, from many-parameter ones, for which this is impossible.

Of course, if it is known that the parameters are not correlated, the many-parameter problem can be reduced to many one-parameter ones. If the parameters are slightly correlated, the problem can be solved with the reasonable precision by the multiple application of the previous procedure. The correlated many-parameter problem with not too many parameters can sometimes be efficiently solved by the recursive application of the one-parameter fit, when for each investigated value of the first parameter the best value of the second is fitted, and for each combination of the first and second, the third is also fitted, and so on. However, this guarantees the presence of the solution inside the found interval only for the first parameter fitted on the uppermost level, and provided that the precision of fitting the other parameters adjusted on the internal levels is absolute. Otherwise, the subjective function for the uppermost level fitting will have little discontinuities or little random admixture. In fact, such numerical noise in the subjective function may appear at the application of any adaptive computational procedure with adjustable steps, intervals, etc.

Thus, although the subjective function is usually assumed smooth in the mathematical sense, its computer representation always has rounding errors and often has the admixture of numerical noise. It is desirable that the fitting algorithms would be tolerant regarding these errors. Our algorithms for one-parameter fit are not quite perfect in this sense, but they are not the subject of this paper. Here we will discuss the maximization of an arbitrary multi-parameter function with simultaneous adjustment of its parameters, for which the toleration to the numerical noise is also a crucial issue.

We are especially interested in fitting "heavy" functions, with large computer time consumptions, singular behavior, many correlated parameters varied in a restricted domain, having the significant third and larger order terms in the Taylor-series expansions and computed with the finite numerical precision.

This set of problems is eventually combined with the large time span between ordering and receiving the function value (usually due to remote allocation of data) and with the possibility to order many function values at once waiting for their receiving nearly the same time. The large time span can effectively prohibit the solution of some rare but important problems unless the program can synchronize the function calls. The synchronization is not entirely the technical problem. Whether it is possible or not is determined by the fitting strategy.

In general, all studies on the fitting strategy fall into two partially overlapped groups and differ mostly by their attitude to the use of function derivatives. The use of derivatives implies that their values at any requested point should be either supplied by the user or, when it is impossible, computed by the program. The latter is done either by the finite differences, or, for the second derivatives, by the use of features of a particular form of the subjective functions (linearization for the minimum squares)[3, 4, 5, 6]. Also the second derivatives are sometimes evaluated by observing how the first ones vary from iteration to iteration, but this is just a variant of the finite differences (the references to such works can be found in [7]). Then, the derivatives are used either to localize the solution, or to try the next approximation or trial, and to evaluate the precision of the current iteration.

The adequate utilization of derivatives, as well as their correct calculation, is not so simple as it sometimes seems. This led other authors[1, 8, 9] to the conclusion that "the returns from this activity are not commensurate with the effort required"[8] and motivated developing the adaptive trial algorithms which do not need derivatives at all, or use only the first ones. They typically choose the next step depending whether the previous one was successful or not, which is determined by the comparison of the function values. The initial trials are more or less random, but then the special procedures adapt the farther movement to the particular landscape and gradually converge to the solution. Of course, this strategy is worth-while only if the user does not supply derivatives, which could otherwise make the trials much more purposeful. However, even if the user does not supply derivatives, it is unclear whether this strategy excels the opposite method based on the numerical calculation of derivatives by finite differences and on their correct utilization.

The discussions about which approach for this case is better are usually "rich in opinions but somewhat deficient in facts"[8]. To make up this deficiency we remark that the adaptive trial algorithms inherently deny synchronization of the function calls, while the algorithms with derivatives are well compatible with it.

Indeed, the computations by finite differences consist in exploring how much the function value at the current "central" point differs from that at the set of neighboring ones. Obviously, all these points can be requested simultaneously. Of course, there is a problem of choosing adequate distances between them or differential step sizes. To suppress the influence of the large-order derivatives these differential steps should not be too large. Simultaneously they should not be too small so as to suppress the numerical noise and rounding errors[1, 10]. However, these problems can be solved. In particular, the latter problem is solved by interval computations.

Although the derivatives allow to make the trials more purposeful, they does not guarantee the success of each. Meanwhile, from each point we may try many steps determined by different assumptions regarding the function behaviour and expressed by different formulas or algorithms. Moreover, we may search the solution simultaneously by many trajectories, considering at every iteration many current points, steps, and exploiting different step generators, even those which are not compatible with the synchronization if taken alone. If the synchronization does matter, all necessary points at each iteration can be ordered synchronously. Such an approach may be faster and is expected to be very resistant against converging to local maxima, minima and other detrimental phenomena.

To our knowledge, none of the available programs (see, for example, FUMILI[4] and MINUIT[7]) support these requirements.

The known programs also do not support recursive fitting, in which at the each step of one fitting procedure another fitting task is executed inside the same program. This problem may seem exotic, but really it is not so. It has already been mentioned that if the fitted variables are strongly correlated, the recursive fitting is sometimes the most efficient way of problem solution. For problems like Rosenbrock's valley[1] the number of function calls may be reduced by the order of magnitude. Moreover, even for classical simultaneous fitting all parameters, the calculation of the next step from each current point can anyway be performed by an iterative fitting procedure, usually by the analysis of the Taylor expansion obtained at the current point[5, 6]. Obviously, this local iterative procedure is recursive with respect to the global one. To provide the recursive fitting, the data of each fitting process should be well isolated from the others. This influences the choice of the programming language and the program design. In particular, this consideration effectively prohibits FORTRAN77 with its static data and favors the use of C++ and object-oriented programming.

These considerations have motivated the development of a new class library. However, our work has been stipulated by our practical needs and by real possibilities. Therefore a set of foundation classes for numerical analysis, appearing applicable for wide range of problems, is supplemented only by a few realized applications, from which we describe here only one: the maximization of a multi-parameter function. This description is rather illustrative and touches on mainly the principal aspects of the program design, to the extent in which we were able to recognize them among the vast amount of less important information.

The program is coded in C++. We included in this text some notations used in the program and given in the syntax of C++. The class diagram is presented in notations similar to that of G. Booch[11]. Our notations differ from these conventions only in the notation of class itself, for which we use more compact rectangular box with rounded corners, and in the notation of objects controlled by the pointers, the original concept explained in the appendix and marked by the open crossed box at the side of the addressed object.

2 Structure of program

In order to make the program scalable and re-usable, we need to determine the classes as the representations of the abstractions of our subject area, the numerical analysis of functions. However, the most of its abstractions represent not static objects but actions: procedures, or algorithms which should be represented in programming as functions. The difference between an object and an action is usually clear: the object is a real or imaginary body which can be created, moved, copied and destroyed (deleted); the action is a process which can be just started and finished. For example, the maximization is an action, the subjective function is also an action. These actions are logically connected and use many other accessory functions and parameters which assist in maximization. Although all these functions and other attributes do not represent the body, even imaginary, they can not work without each other, and hence should be initialized and annulled synchronously. This motivates their declaration as the members of a single class. The principal members of this class are the subjective function and the function solving the fitting problem. The subjective function is usually called here `virtual void calc_function_event()`, but it has different arguments depending on the problem. The function solving the fitting problem is usually called `execute(void)`. This function is responsible for initialization, processing the necessary number of iterations, termination, and for debug printing. It turns out that such function is necessary for solution of any problem, although this function is not always expected to be directly called by the user. This means that we can define this common function, as well as many other ones, in a common base class, here the class `ModelFunction` (see fig. 1), from which we can derive classes adapted for any particular problems.

The standard procedure implemented in the function `execute()` consists of the iterations at which we compute the function values in the ordered points, calculate next steps, compute the functions at the end of the steps, choose the best steps and initialize the new points. Instead of points we sometimes use the term "nodes" to stress the fact that one node may represent several current points, although we can also consider many current nodes at each stage. Thus we will need a few more member functions which

> compute the subjective function values in the ordered points, `void calc_fun(...)`;
>
> compute the next steps, `int calc_steps(void)`;
>
> choose the best steps and initialize the new points, `int new_nodes(void)`.

The last two functions should check the finishing conditions and issue an order to finish when necessary. The calculations of the subjective function values are specially gathered into a single function `void calc_fun(...)` so as to provide the synchronization, the possibility to order several function values at once. Besides synchronization this function should assist in handling many events, numerical calculation of the derivatives, evaluation of the precision of

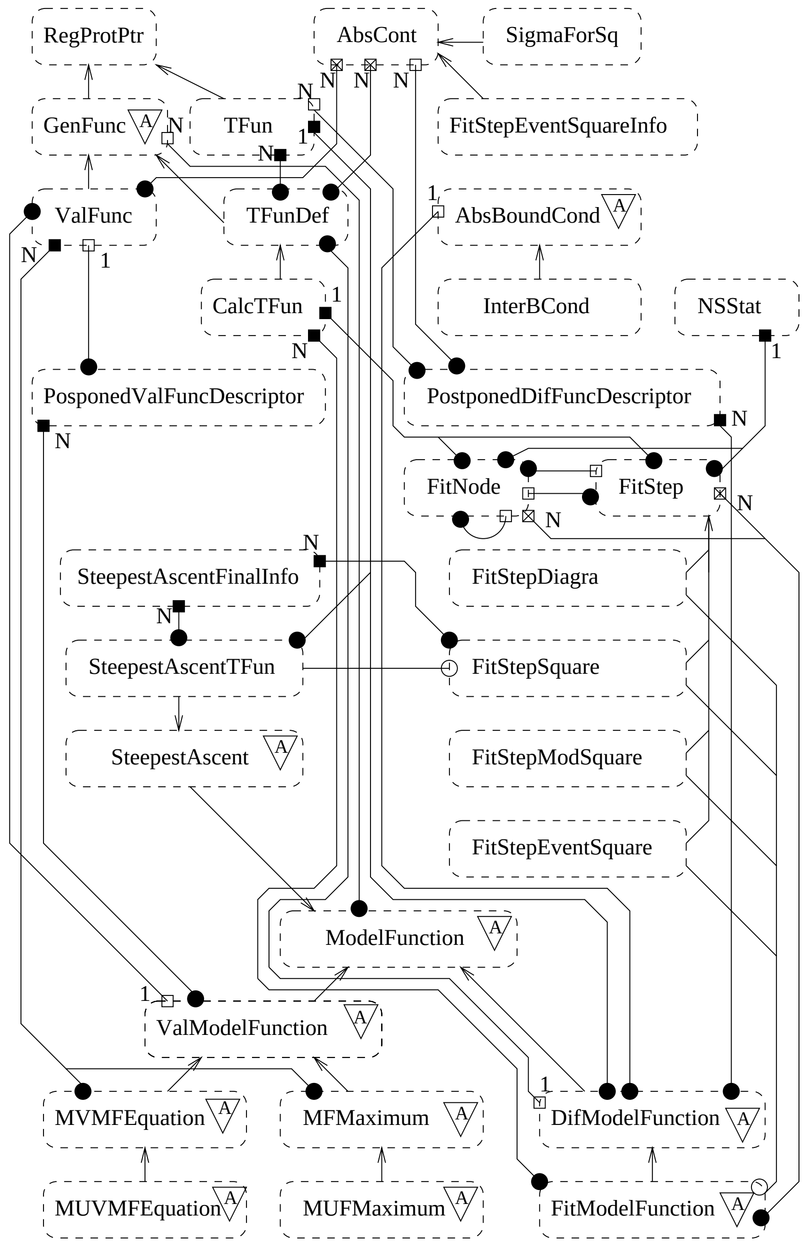

Figure 1: The class diagram of LikelihoodLib.

the derivatives, increasing the precision if necessary, and repetition of calculation of the derivatives. Of course, there are problems in which all the additional features: synchronization, handling many events and derivatives, are not necessary. For example, sometimes we need merely calculate the function value in one single point. Then we overload this virtual function by a special version defined in derived class, as we did it in class `SteepestAscent`.

The subjective function is to be declared and defined in the sub-classes of `ModelFunction`. Of course, it can have very different headers even for the problems which are solved with the base version of `void calc_fun(...)`. To call the subjective function, as well as to check and to increase the precision, `calc_fun()` should appeal to intermediate agents with fixed headers, pure member functions of an abstract base class, here `GenFunc`. Each class derived from it should contain current parameters at which the function is to be computed, the results of this computation, the definition of that agent function as a call of the real subjective function determined in the corresponding object of the sub-class of `ModelFunction`, and also a function which checks and increase the precision.

In detail, the class `GenFunc` declares the following headers:

`virtual void calc_fun_event(...)=0` – computes the function for one event.

`virtual void calc_fun(void)=0` – computes the derivatives if necessary.

`virtual int increase_precision(void) {return 0;}` – increases the differential steps if necessary and returns 1 if increased.

The function `ModelFunction::calc_fun(...)` accesses all these methods via the protected pointers to `GenFunc` (The sense of the protected pointers `ProtPtr<>` is explained in the appendix). The class `ModelFunction` contains an array with variable length `DinLinArr< ProtPtr<GenFunc> > genfunc` with these pointers. They should be initialized during the calls of `calc_steps()` and `new_nodes()`. If the derivatives are computed by finite differences, the real subjective function may be called many times in each call of `GenFunc::calc_fun_event(...)`. Its argument may be either the total sub-class of `GenFunc` (if the derivatives are not necessary) or smaller class determining a particular point.

Two auxiliary member functions of `ModelFunction` are used for handling the events or bins. The synchronization may be implemented either by reading the next event and multiple calling the user function related to one event and one point, or by calling a more complicated user function which itself should manage reading all events and calculations for all necessary points. The latter is more convenient when the reading program is stand-alone. It may even run on a different computer.

Thus, we need two auxiliary member functions:

> `fix_ev(...)` – fixes the next event or issues a signature of finishing. Fixing the event may consists in its reading from a disk file, or conducting preliminary mathematical calculations common for all points.
>
> `end_ev(...)` – finishes calculations of the current event. Called after the calls of the studied function for all requested points. Useful for the second type of synchronization, when the program should issue the request for processing all events and points simultaneously.

We determined two sub-classes of `GenFunc`:

> The class `ValFunc` – a many-parameter many-value function without derivatives.
>
> The class `TFunDef` – a many-parameter single-value function with derivatives (some operations are currently determined only up to the second order).

They correspond to the following sub-classes of `ModelFunction`:

> The class `ValModelFunction` oriented for solution of one-parameter problems or many-parameter problems by sequential solution of one-parameter problems.

> The class `DifModelFunction` oriented for solution of many-parameter problems by free motion in the parameter space.

The both sub-classes may be invoked recursively in any combinations and in any depth.

The both sub-classes also have derived classes. The class `ValModelFunction` is specialized in the classes `MVMFEquation` and `MFMaximum`, which adjust only one parameter, and in `MUVMFEquation` and `MUFMaximum`, which arrange sequential adjusting of all parameters. The latter classes are efficient for the functions with the little correlation between the parameters. Otherwise, a very efficient procedure is the recursive application of two first classes. Note that when dealing with the latter classes the user should call the function `total_execute()` instead of standard `execute()`. The function `total_execute()` calls `execute()` for each parameter in a loop until the solution stops migrating.

The class `DifModelFunction` has only one sub-class designed for fitting, `FitModelFunction`.

The class `ValFunc` has a very simple structure. The function `ValFunc::calc_fun_event(...)` just calls the subjective function declared in `ValModelFunction`. The function `ValFunc::calc_fun(...)` does nothing.

The structure of the class `TFunDef` is much more sophisticated. If the user does not supply derivatives, this class computes them numerically. For this purpose besides the main point it should determine and keep the function value and the available derivatives in some neighboring points dispersed around the main one by differential steps. To handle these points we need a simpler class functioning as a container for the function and its derivatives but unable to compute them by finite differences. Nevertheless, as it is shown in section 4, such class, called here `TFun`, can compute the derivatives semi-analytically.

At the calculation of the differential steps the program should take into account some restrictions determined by the user as the minimal or maximum differential steps. The program should choose the differential steps so that they would meet these restrictions and provide adequate numerical precision. This can be done by the analysis of the values and the precision of the derivatives calculated for some neighboring point usually investigated at the previous iteration. The natural assumption is that the second derivative does not vary significantly. Since this is not guaranteed, after the current point is calculated, the precision should be checked. If the precision is not adequate, we may not be able to infer the correct value of the differential steps. Therefore they are merely increased by 10 times or till the maximum allowed value, and the repetition of the calculations is requested. The requirements on the precision depend on methods which generate steps. Therefore the class `TFunDef` and its sub-class `CalcTFun` call some virtual methods of the class `DifModelFunction`, which are specified in `FitModelFunction` and call some functions of the classes derived from `FitStep`.

The boundaries of the valid space may be either strict or not, which means that the differential steps either can step out of the boundaries or can not (the ban is useful if the function is singular outside). In the latter case the configuration of the location of the additional points depends on whether the main point is near the boundary. If the standard allocation of the additional points around the main one leads to violation of the strict boundary, the program puts them from one side of the main point toward internal space and use other formulas for differentiation.

The boundaries of the working area for the parameters of `TFunDef` are to be determined by a class derived from the class `AbsBoundCond`. Currently the class `AbsBoundCond` has only one specific instance `InterBCond` determining independent intervals for every parameter. The object of this class is accessible from `TFunDef` via the protected pointer to `DifModelFunction` and from the latter to `AbsBoundCond`. A principal method of this class restricts by the components the vector of parameters, which is usually the next step, by such a way that it finishes at the boundary.

The fitting program may not need the function value and the derivatives at once. At the beginning the program usually needs only the function value to check if the trial is successful. The derivatives will be needed only for the successful trial. To optimize the speed of computing, the user may choose to order the subjective function either in the main and all additional points at once, or to calculate only the main point and to postpone the additional ones until the trial is accepted.

In addition, some of the parameters may be fixed at fitting, and the corresponding derivatives should just be assigned zeros without calculations of the corresponding additional points.

The class `TFunDef` contains a lot of useful data members and functions which we will not describe in detail.

For both classes `ValModelFunction` and `DifModelFunction` the second type of synchronization is supported by the similar data members (the class names are omitted):

```
DinLinArr<PosponedValFuncDescriptor> pfd;
DinLinArr<PosponedDifFuncDescriptor> pfd;
```

and by the possibility to specify the subjective function which does not calculate anything but merely copies its argument to `pfd`. The actual calculations should be done inside overloaded `end_ev()` which should use `pfd`.

The class `FitModelFunction` is designed by the principle of competitive application of many available methods for step generation from one or many current points. After the steps are proposed by all available methods, it selects the best steps by ratings. The rating is meant to represent the reliability of the step method and the expected function value at the end. It is calculated as the weighted sum of the current and expected function values with weights depending on the method. The number of steps selected at this stage is not more than `int FitModelFunction::q_watch_step`. Then the class `FitModelFunction` requests the function values at the end points and chooses a few points with the largest function, not more than `int FitModelFunction::q_watch_point`. The function value at the chosen end point of the step should be more than its value at the point from which the step originates. Otherwise, the step is not accepted. If `q_watch_point` is not accumulated, the class `FitModelFunction` asks the particular method to correct the step, taking into account the obtained function value. This approach resembles the method of "information board" used in some researches on artificial intelligence[11].

This procedure starts from one or many initial points, but finishes only at one. Even if a few methods simultaneously detect the finishing conditions, the point with the largest function value is chosen. Not only the particular methods define the finishing conditions, but the class `FitModelFunction` participates in this also. Even if a particular method announces finishing, this class may not accept the announcement if it detects that the maximal value of the function ever found is more that $f_{\rm finish} + |f_{\rm finish} \cdot \epsilon_y|$, where ϵ_y stands for `double FitModelFunction::rel_accuracy_y`

If the solution is found, the associated statistical error can be found on the basis of the inverse Hessian matrix and checked by shifting of any one parameter by $\pm\sigma$ from the solution and fitting the others. This is done in the member function `report(...)`.

3 Numerical inaccuracy and interval computations

The rounding errors and the numerical noise often appearing in the subjective function values corrupt the derivatives computed by finite differences unless the differential steps are large enough and the natural variation of the function exceeds its numerical inaccuracy. The inaccurate derivatives corrupt the computation of the next step, deviate the expected distance to solution, compromise the finishing conditions and also the statistical uncertainty, if the fitting is based on statistics. Although the precision of the subjective function is assumed to be known by the order of magnitude, even the mere extrapolation of its uncertainty to these values at the given differential steps is not trivial. The matrix calculations involved in these procedures are known to have trend to accumulate the numerical errors, especially if the parameters are correlated or not quite independent.

The simplest way of tracing the numerical precision is interval computations, when each number x is accompanied or even substituted by the lowest and the largest limits $[x_l, x_r]$ in which the main value may float due to the numerical inaccuracy. These limits are processed through every arithmetic operation and the call of each standard function. For example, if $x_l > 0$ and $y_l > 0$, $(xy)_l = x_l \cdot y_l$. But if $x_l > 0$ and $y_r < 0$, $(xy)_l = x_r \cdot y_l$, and so on. There is a theory of interval computations aimed at minimization and tracing the rounding errors[12]. In this theory and in the corresponding programming languages the main or the most probable value is not watched. We currently prefer to watch it, since it is not necessarily equal to the medium value of the interval, which can be large and exaggerated.

Indeed, this estimate usually exaggerates the inaccuracy since it always assumes the worst case when the operands are anti-correlated. In particular, this is clearly exposed in operations in which the same variable appears twice or more: $x \cdot x$, etc. Of course, the multiplication by itself can be expressed as raising to the second power by the specially written function which takes into account correlations. But one should keep in mind that such operations can appear in an indirect or hidden form. Therefore the manipulations with such intervals and their interpretation should be carried out with some caution.

Of course, there is no necessity to create the special programming language for interval computations (to the contrary of what is advocated in [12]), since interval arithmetic in any variant can be included in an object-oriented language as especial class with the overloaded arithmetic operations and algebraic and other mathematical functions. Our version of such class is called `DoubleAc`. It manipulates with double precision floating point numbers. Many calculations during fitting are executed with the objects of this class. We remark that the class `DoubleAc` is not complete implementation of the theory [12]. It has many simplifications compared with this theory.

4 Differential arithmetic

The idea of differential arithmetic is borrowed from one of the examples in [12] and based on the fact that the derivative of the result of an arithmetic operation applied to two functions of one variable is expressed via the values and derivatives of these functions taken alone. This complicated statement, in fact, means well known rules: $(u \pm v)' = u' \pm v'$, $(u \cdot v)' = u'v + uv'$, and $(u/v)' = u'/v - uv'/v^2$. If we have an arbitrary algebraic expression with operands whose numerical values are known as well as the numerical values of their derivatives, we can calculate not only the numerical value of its result but also the value of derivative of the result. If the operand is a constant, its derivative is known, it is zero. Similarly, if the operand is the variable parameter, its derivative is unit. We also know the derivative of any power of the variable parameter. From these elements we can compose any algebraic expression. Obviously, this idea can be extended for the partial derivatives of any order of multi-parameter functions. This allows to avoid analytical differentiation as well as the numerical differentiation by finite differences, but to exploit some third semi-analytical semi-numerical method. We will identify it by the name "differential arithmetic" following [12], although it is clear that this is not only arithmetic about. By the similar way one may process any elementary functions: algebraic, trigonometric, logarithmic, etc. If $F(x)$ is any elementary function of one argument, for example, sin, cos etc. , and F' denotes its derivative relative to the argument, its application to f gives the following: $(F(f))'_i = F' f'_i$, $(F(f))''_{ij} = F'' f'_i f'_j + F' f''_{ij}$, and so on. Thus, the derivatives of the result are again expressed via the derivatives of these functions taken alone.

This method of computing derivatives is especially convenient if implemented in an object-oriented style. Indeed, we can consider the function value together with all values of its derivatives as a single object of a class. We can define the arithmetic operations and elementary functions for this class as procedures returning objects of the same class. Obviously, the returned objects should contain the function values and derivatives computed according to these ideas. Then, we can write the ordinary mathematical expressions and even the total programs operating not with the regular floating point numbers, but with the objects of such class. In this case the derivatives of the expressions will be automatically computed as some by-product of computation of the values of functions.

This approach is implemented in the class `TFun`. Currently we have encoded the differential arithmetic only up to the second order, which is the necessary minimum for our purposes.

5 Numerical calculation of derivatives by finite differences

The first partial derivative is normally computed by

$$f'_x = \frac{f(x+h) - f(x-h)}{2h}, \tag{1}$$

where h is differential step. If the point x is near the strict boundary on the left side from it, we use other points and formula:

$$f'_x = \frac{1.5f(x) - 2f(x-h) + 0.5f(x-2h)}{h}, \tag{2}$$

Here we assume that the second derivative is more or less constant and take into account its influence on the first one.

If the second partial derivative is not supplied, but the first one is given, the second one is computed by

$$f''_{xx} = \frac{f'_x(x+h) - f'_x(x-h)}{2h}. \tag{3}$$

The similar formula is used near the strict boundary (the boundary is to the right side):

$$f''_{xx} = \frac{f'_x(x) - f'_x(x-2h)}{2h}. \tag{4}$$

The point $f(x-h)$ is not calculated in this case, which allows to reduce the time consumption.

If the first derivative is not provided, the second one is calculated by the function:

$$f''_{xx} = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \tag{5}$$

It is assumed to be constant and determined by the similar formula for shifted points.

To calculate the second derivatives taken by two different parameters, we check which first derivatives are supplied and use either

$$f''_{x_1 x_2}(x_1, x_2) = \frac{f'_{x_1}(x_1, x_2 + h_2) - f'_{x_1}(x_1, x_2 - h_2)}{2h_2} \tag{6}$$

or

$$f''_{x_1 x_2}(x_1, x_2) = \frac{f'_{x_2}(x_1 + h_1, x_2) - f'_{x_2}(x_1 - h_1, x_2)}{2h_1} \tag{7}$$

The same formulas are used for shifted points.

If none of the first derivatives are supplied, we use the function values:

$$\begin{aligned} f''_{x_1 x_2}(x_1, x_2) \quad = \quad & \frac{1}{4h_1 h_2} \cdot \\ & (f(x_1 + h_1, x_2 + h_2) - f(x_1 - h_1, x_2 + h_2) - \\ & f(x_1 + h_1, x_2 - h_2) + f(x_1 - h_1, x_2 - h_2)) \end{aligned} \tag{8}$$

The same formula is used for shifted points.

It is seen that for each combination of two parameters we need to calculate the function in four additional shifted points: $x_1 \pm h_1$ and $x_2 \pm h_2$. We can reduce this number to one, if we use, for example, the following formula:

$$\begin{aligned} f''_{x_1 x_2}(x_1, x_2) \quad = \quad & \frac{1}{h_1 h_2} \cdot \\ & (f(x_1 + h_1, x_2 + h_2) - f(x_1, x_2 + h_2) - \\ & f(x_1 + h_1, x_2) + f(x_1, x_2)) \end{aligned} \tag{9}$$

Here we have only one additional point $(x_1 + h_1, x_2 + h_2)$. This not symmetrical approach can be inaccurate if there are large third derivatives. Otherwise, if this approach is acceptable, the user can assign `int DifModelFunction::s_sddpf=1`, and this formula will be used for not restricted case. If any of the three shifted points are out of the boundary, the program tries other combinations following the same principle.

In some applications the second derivative can be approximately computed from the first one. The example is minimization of the sum of squares by a linearization procedure [3, 4]. This method is implemented in the classes `FitStepEventSquare` and `FitStepEventSquareInfo`. The subjective function should have the form

$$f = \sum_i f_i = \sum_i -\frac{(y_i - m_i(\vec{x}))^2}{\sigma_i^2}. \tag{10}$$

We search its maximum at the constant parameters σ_i. The first derivative of (10) will be

$$\frac{\partial f}{\partial x_j} = 2\sum_i \frac{(y_i - m_i(\vec{x}))}{\sigma_i^2} \cdot \frac{\partial m_i(\vec{x})}{\partial x_j}. \tag{11}$$

If we ignore the second derivative of $m_i(\vec{x})$ by $\vec{x}$, then the second derivative of f is

$$\frac{\partial^2 f}{\partial x_j \partial x_k} = -2\sum_i \frac{1}{\sigma_i^2} \cdot \frac{\partial m_i(\vec{x})}{\partial x_j}\frac{\partial m_i(\vec{x})}{\partial x_k}, \tag{12}$$

So as not to write a special program and an interface for computing and transferring $m_i'(\vec{x})$, we use f_i' which is transferred or computed by default, if the event information is provided by the user in the function `calc_function_event_detailed()`. Then the $m_i'(\vec{x})$ is restored from (11) and (10):

$$\frac{\partial^2 f}{\partial x_j \partial x_k} = \frac{1}{2}\sum_i \frac{1}{f_i} \cdot \frac{\partial f_i}{\partial x_j}\frac{\partial f_i}{\partial x_k}, \tag{13}$$

6 Maximization of multi-parameter function

Since the maximum of any smooth function of the parameters $\vec{x}$ coincides with zero of its gradient or the first partial derivatives

$\vec{f'}(\vec{x})$, and behavior of the gradient is controlled by the second partial derivatives or Hessian $f''(\vec{x})$:

$$f''_{ij}(\vec{x}_0) = \left[\frac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j}\right]_{\vec{x}_0}, \tag{14}$$

the maximum will coincide with the solution of an equation:

$$\vec{f'}(\vec{x}) = \vec{f'}(\vec{x}_0) + f''(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0) = 0, \tag{15}$$

provided that the function $f(\vec{x})$ really has maximum and its third derivatives are everywhere zero. The latter is not necessarily true. Therefore, even if the function really has maximum, the solution of (15)

$$\vec{x} - \vec{x}_0 = -f''(\vec{x}_0)^{-1}\vec{f'}(\vec{x}_0) \tag{16}$$

may point to a minimum or just to a stationary point of the corresponding quadratic form, where the function value may be even less than $f(\vec{x}_0)$. Therefore the single or repetitive application of (16), which is sometimes called the method of Newton, may lead to confusing results. To cope with this problem we need first to restrict the step length $\vec{x}-\vec{x}_0$ by a maximal allowed value which reflects our expectations or knowledge about the influence of the higher-order derivatives and the range at which they manifest themselves. Then, instead of or in addition to the restricted general solution by (16) we can apply any variant of the steepest descent (for minimization) or steepest ascent methods[1]. By their base variant the trajectory of search is directed along the gradient and propagated until the function maximum (or minimum) is found. Then the trajectory is rotated to be along the new gradient, which is perpendicular to the old one, and the procedure is repeated. To provide synchronization we avoid the iterative procedure of searching the maximum along the gradient, but compute its expected position by the Hessian. The most powerful and sophisticated method is moving along the gradient by little steps and re-computing the gradient by the left equation of (15). The move may proceed until some simple conditions are satisfied, perhaps, until the total distance travelled is less than the maximum allowed step, or the ordered number of iterations is executed. Then the new function value and derivatives are requested at this end point and the procedure is repeated. This method allows to take into account the third and higher order derivatives, if they are supplied by the user, (then, instead of the left part of (15) we will use the more general form) and to move exactly by the sharp ridge or valley if they are. The differential arithmetic of the class `TFun` would allow in principle to compute the derivative of any order.

Note that these manipulations with the derivatives are not necessary for the case of one parameter function, where we can isolate the maximum in an interval which will exactly determine the current precision. Therefore for the latter problem it is better to use the class `MFMaximum`.

6.1 General solution

The general solution by Newton's formula (16) is done by the class `FitStepSquare`. The similar algorithm for minimization of the sum of squares with the second derivatives calculated by (13) is implemented in the class `FitStepSquareEvent`. However, the latter class is not currently able to apply functionality of `SteepestAscentTFun` (see below).

At the calculations of differential steps these methods assume that the second derivative is not zero and that it is similar by the order of magnitude at the previous and at the current point. The differential step is calculated to provide the necessary precision of the diagonal elements of the Hessian matrix and of its inverse matrix, and the precision of the next step. Initially the differential step is offered to be approximately by an order of magnitude less than the previous real step. If the considerations given above dictate increasing the differential step length, it is also required that the differential step should not be larger than the final accuracy. Everything is controlled by the parameters which can be changed by the user.

The constructors of these classes first calculate the next step by (16) and restrict it by the maximal allowed step. The latter is possible by two different ways: separately by components and by proportional reducing. The first way changes the direction of step, while the second one preserves it. The both steps are then restricted by boundaries through the call of `AbsBoundCond::restrict(...)`. Here only one type of restriction is provided, the reducing by components, since the proportional reducing may result in zero length step. This does not support travelling along the boundary which is necessary if the the function increases to the outside direction.

Then the program evaluates the expected function values for one or two different steps by the Taylor expansion

$$f_{\text{exp}} = f(\vec{x}_0) + \vec{f'}(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0) + \frac{1}{2}(\vec{x} - \vec{x}_0)^\top f''(\vec{x}_0)(\vec{x} - \vec{x}_0) \quad (17)$$

and chooses the step related to the largest value. If this value is less than the current one $f(\vec{x}_0)$, or if the actual function value $f(\vec{x})$ obtained at this step is less than $f(\vec{x}_0)$, the program abandons the Newton's formula and applies the integration of the simultaneous equations

$$\frac{d\vec{x}(l)}{dl} = \vec{f'}(\vec{x}_0) + f''(\vec{x}_0)(\vec{x}(l) - \vec{x}_0), \quad (18)$$

where l means the trajectory length. This is performed in the class `SteepestAscentTFun` and gives the steepest ascent trajectory by the Taylor series. Of course, this procedure is computationally worth-while only if the subjective function is computed much longer than its Taylor series. The integration is performed until the maximal step is accumulated by any one parameter or until the ordered number of steps is done (currently 100) with lengths

$$0.2 \cdot \vec{h} = 0.2 \cdot \frac{\vec{f'}(\vec{x}_0)|\vec{f'}(\vec{x}_0)|^2}{|\vec{f'}^\top(\vec{x}_0) f''(\vec{x}_0) \vec{f'}(\vec{x}_0)|}. \quad (19)$$

$|\vec{h}|$ is the expected distance to either the maximum or minimum of the quadratic form along or counter the direction of the gradient, see the next section. The program memorizes not only the end point of this trajectory, but also all its previous points. This information allows to recede by the trajectory if the function value at the end point appears less than $f(\vec{x}_0)$. If the investigated point is point number n, the new considered point will be just point number $n/4$ unless the integer part of $n/4$ is already equal to 0. In the latter case the correction is declined.

This method is the most precise. Its rating is assumed to be equal to the expected function value f_{exp}.

For the case of many parameters this method is the only one which takes into account the correlations between the parameters and only its next step evaluates correctly the real precision. Therefore for many parameters only this method is allowed to finish the iterations. The conditions of finishing are the following:

- The class `SteepestAscentTFun` is not used for obtaining the current step.
- The current and previous steps are less by components than the required accuracy `DinLinArr<double> FitModelFunction::accuracy`.
- The new function value at the end of the current step is larger than the function value at the original point minus its absolute value multiplied by `double FitModelFunction::rel_accuracy_y`.
- The diagonal elements of the error matrix $f''(\vec{x}_0)^{-1}$ are negative (to avoid considering the next step and long computations of the matrix determinants).
- The signs of the main minors of the Hessian indicate that the corresponding quadratic form is negatively-determined.

6.2 Steepest ascent with Hessian

Originally, the steepest descent method is proposed for minimization at the cases when either the Hessian is not known or the application of the Newton's formula leads to confusing results. However, the numerical calculation of the Hessian is not more difficult than computing the gradient (both by differential arithmetic and by finite differences). If the Newton's formula fails, we can use the integration of (18). Nevertheless, if the subjective function is computed much faster than its Taylor expansion, it is reasonable to integrate the subject function itself. Moreover, if the Hessian is computed not quite precisely, it is reasonable to make a trial according with this method together with the trial by the previous general method so as to choose the best step. Although our modification uses the Hessian, its inaccuracy affects only the step length and does not affect its direction. If, nevertheless, this method fails as well (the new function value is less than the old one), we can always correct (reduce) the step length by quite intelligent way. Also this method is more natural for one-parameter functions. Therefore in the latter case it is used instead of the previous one, while for many-parameter functions it is used as optional.

The original method consists in making steps along the function gradient exactly or approximately to the maximum appeared along this direction. As we have already mentioned, to provide synchronization we avoid the iterative search of this maximum, but evaluate its position by the current Hessian. We denote by $\vec{u}$ the unit vector along the gradient $\vec{u} = \vec{f'}(\vec{x}_0)/|\vec{f'}(\vec{x}_0)|$. If to put the beginning of this vector in the point $\vec{x}_0$, the Taylor expansion T will vary along this vector according to

$$T(r) = f(\vec{x}_0) + r\vec{f'}^{\top}(\vec{x}_0)\vec{u} + \frac{1}{2}r^2\vec{u}^{\top}f''(\vec{x}_0)\vec{u}. \tag{20}$$

Its derivative $T'_r(r)$ has zero at the point

$$r = -\frac{|\vec{f'}(\vec{x}_0)|}{\vec{u}^{\top}f''(\vec{x}_0)\vec{u}} = -\frac{|\vec{f'}(\vec{x}_0)|^3}{\vec{f'}^{\top}(\vec{x}_0)f''(\vec{x}_0)\vec{f'}(\vec{x}_0)}. \tag{21}$$

Whether this is the maximum or minimum of $T(r)$ depends on the sign of the denominator. Normally it should be negative, and the value of r is positive. Otherwise, r will point to the minimum and there is no any sense to make such step. But we may anyway try step along the gradient with the same length $|r|$ assuming that we approach to the maximum. Owing to the reasons discussed in [1], it is reasonable to reduce this step slightly, by a factor of 0.9 or so. Therefore the searched step is

$$\vec{h} = 0.9 \cdot \frac{\vec{f}'(\vec{x}_0)|\vec{f}'(\vec{x}_0)|^2}{|\vec{f}'^{\top}(\vec{x}_0) f''(\vec{x}_0) \vec{f}'(\vec{x}_0)|}. \tag{22}$$

Similarly to the previous method, this step is restricted by the maximum allowed step and by the boundaries. But since the idea is to step along the gradient, the changing of the step direction is considered as not desirable action. Therefore if the step reduced proportionally is not zero, the program uses only it. Otherwise it uses the step reduced by the components.

The expected function is calculated by (17). This method is usually precise and its rating is assumed equal to the expected function.

If the actual value of the function at the end point is found to be less than the initial $f(\vec{x}_0)$, we can correct the step by reducing it. The program does this by drawing an imaginary parabola by the values of the function and gradient at the initial point and by the value of the function at the end point. The maximum of the parabola gives the next approximation.

The finishing conditions are similar to the previous method except the last condition, which is substituted by the demand of the negative denominator in (21).

This method is implemented in the class `FitStepModSquare`.

6.3 Steepest ascent according to variation of gradient

This method is designed for the cases when the Hessian is not available. Since it is usually available, this method is utilized as one more optional generator of the trials useful for the functions with erratic behaviour. For the regular functions this method is far less efficient with respect to the previous ones.

This is also a variant of the steepest ascent method. But to calculate the step length we take here into account the difference of the gradients obtained at the previous and at the current iteration.

To explain this method we first consider the 1-parameter function $f(x)$ with the Taylor expansion around the point x_n obtained at iteration number n:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2. \tag{23}$$

If to substitute the second derivative by

$$f''(x_n) \approx \frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}, \tag{24}$$

the zero of the first derivative will appear at the distance

$$h = x - x_n = -\left(\frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}\right)^{-1} \cdot f'(x_n). \tag{25}$$

We generalize it to many parameters by

$$\vec{h} = -k \cdot \left(\frac{|\vec{f}'(\vec{x}_n) - \vec{f}'(\vec{x}_{n-1})|}{|\vec{x}_n - \vec{x}_{n-1}|}\right)^{-1} \cdot \vec{f}'(\vec{x}_n), \tag{26}$$

where k is the parameter which should indirectly take into account the angle between the function gradients. This method gives more or less good approximation if $\vec{f}'(\vec{x}_n)$ is parallel or anti-parallel to $\vec{f}'(\vec{x}_{n-1})$. But if there is a large angle between them, it is better to reduce the step. Therefore we multiply it by the absolute value of the cosines of the angle between the last step and the current gradient:

$$k = \max\left(0.1, \left|\frac{\vec{h}_{n-1}}{|\vec{h}_{n-1}|} \cdot \frac{\vec{f}'(\vec{x}_n)}{|\vec{f}'(\vec{x}_n)|}\right|\right). \tag{27}$$

By this way we reduce the confusing straggling and make the steps more gradually approaching the maximum. Another applied way to smooth the erratic behavior of this method is the use of a variable which follows the variation of the step length but always remains a little bit larger than the actual step length and resists its sharp increase. This variable has sense of the current maximal allowed step length. In the occasion when the offered step length is sharply increased and appears more than the current value of this variable, the latter forces its reduction to the currently allowed maximum, although this variable itself becomes a little bit larger. Of course, the step is also restricted by the maximal allowed step determined by the user and by the boundaries.

The expected function is evaluated by

$$f(\vec{x})_{\text{exp}} = f(\vec{x}_n) + |f'(\vec{x}_n)||(\vec{h}_n)| + \frac{1}{2}\left(\frac{|\vec{f}'(\vec{x}_n) - \vec{f}'(\vec{x}_{n-1})|}{|\vec{x}_n - \vec{x}_{n-1}|}\right) \cdot \vec{h}_n^2. \tag{28}$$

The method's rating is calculated by

$$R = 0.5 f(\vec{x}_n) + 0.5 f(\vec{x})_{\text{exp}}. \tag{29}$$

If the new function value is less than the current one, the correction of the step is performed similarly with the previous method.

This method is not precise and it is not considered as finishing.

This method is implemented in the class `FitStepDiagra`.

A Safe use of pointers

The pointers play the principal role in object-oriented programming in C++. Simultaneously they represent the most unreliable element of the language. The manipulations with the pointers do not automatically force adequate manipulations with addressed objects and vice versa. If the pointer is used for establishing certain relations between objects, its inertness can eventually corrupt the program. However, it turns out that the logic of these relationships almost always falls into one of two categories, and both of them may be determined and automated by the well protected template and regular classes. Although the use of such classes gives computer some additional work, it reduces time of programming, provides certain scalability and re-use of the program, and, hence, seems reasonable.

Shortly, the pointers support such relations between the objects when one object can access or use another object but the latter is not the physical or logical component of the first. Another application of the pointers is the support of polymorphism, when one object accesses another one whose type is not completely determined (we mean handling the derived types with virtual functions; the term polymorphism is sometimes used for other purposes, but here we use the meaning suggested in [13]). What is missed is the support of polymorphism for the logical components, although such relation occurs very frequently.

Moreover, even the supported relation, the reference to the alien object, is not supported in total and prone to programming errors, since the deletion of the addressed object does not result in annulling or clearing the pointer and in denying the further access to it. The attempt to simulate the relation of logical inclusion frequently leads to errors by the same reason. In general, any regular pointer appearing as the class member of an application class (except well debugged accessory library classes) means a programming error, either now or in future. This makes the result of computing random, depending on whether the hardware detects "segmentation fault" or what will be found at the place of the deleted object.

This motivates substitution of the regular pointers, when they are used as the class members, by the objects of template classes. Since we have only two main types of relations: the reference to an alien object and the logical inclusion, we need just two base templates simulating these relations. As follows from the discussion above, the first one should just mimic the regular pointer and protect it from the errors appearing at careless manipulations with the addressed object. This type is called the protected pointer and denoted by `ProtPtr<>`. The addressed object should know the addresses of all protected pointers to it and clear them at its deletion. As an optional feature it can clear them at the substitution of itself by another object through the operator of assignment. Technically, the class of the addressed object should be derived from a special base class called `RegProtPtr`, "Register of Protected Pointers".

The second type should support the logical inclusion of one object into another. If the pointer is copied or deleted either together with the object to which it belongs or separately, this action should force similar operation with the addressed object. By the other words, this pointer should control the addressed object. Technically, such object should be allocated in the free memory. To provide correct copying of the objects of derived classes each such class should define the same virtual member function `copy(void)` which merely allocates the new exemplar of the object, a copy of itself, in the free memory. The controlling pointer is denoted by `AutoCont<>`.

The protected pointers do not have significant additional functionality with respect to the regular pointers and perhaps they do not need special notation in the class diagrams. The controlling pointers need a special notation. They are marked by the open crossed box at the side of the addressed object. Thus, if the connection is marked by the closed circle at one end and by the closed box at the other end, this means that the object of the class marked by the box is the member of the object of the class marked by the circle. The same with the open box at the other end means, that the object with the circle knows only the address of the object with the box, but does not control it. The intermediate crossed open box means that the addressed object is logically the member of the object with the circle and is controlled by it. In the two latter cases the actual type of the addressed object may be either that which is denoted, or any derived from it.

Author thanks A. V. Kravtsov for reading the draft of this text and for useful comments on it.

References

[1] *H. H. Rosenbrock*. An automatic method for finding the greatest or least value of a function. Comput. J. **3**, 175 (1960).

[2] *I.B. Smirnov*. Track Reconstruction for Forward Spectrometer of SPES4-π Experiment. Preprint PNPI-2345, Gatchina, 2000.

[3] *S. N. Sokolov, I.N. Silin*. Search of minimums of functionals by method of linearization. Preprint JINR D-810, Dubna, 1961, (Rus.).

[4] *I.N. Silin*. The standard program for solution of problems by the method of minimum squares. Preprint JINR 11-3362, Dubna, 1967, (Rus.).

[5] *A. Jones*. Spiral - a new algorithm for non-linear parameter estimation using least squares. Comput. J. **13**, 301 (1970).

[6] *D. W. Marquardt*. An algorithm for least-squares of non-linear parameters. J. of the Society for Industrial and Applied Math. **2**, 431 (1963). (inavailable for us, referred from [5])

[7] *F. James*. MINUIT - function minimization and error analysis. CERN program library long writeup D506.

[8] *R. Hooke and T. A. Jeeves*. "Direct Search" solution of numerical and statistical problems. J. Assoc. Comput. Mach. **8**, 212 (1961).

[9] *J. A. Nelder and R. Mead*. A simplex method for function minimization. Comput. J. **7**, 308 (1965).

[10] *J. Pumplin, D. R. Stump, and W. K. Tung*. Multivariate fitting and the error matrix in global analysis of data. MSU-HEP-07100, CERN-TH/2000-249.

[11] *Grady Booch*. Object-Oriented Analysis and Design with Applications, second edition. Addison-Wesley Publishing Comp., Russian edition: "Binom", "Nevsky Dialekt".

[12] *R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich*. PASCAL-XSC, Language reference with examples. DMK, Moscow-2000 (Rus., translation from Springer-Verlag).

[13] *B. Stroustrup*. The C++ programming language. Binom publishers, Moskow, 1999, (Rus., translation from Addison-Wesley).