Object-Oriented Model for Partially Separable Functions in Parameter Estimation*

Jaakko Järvi[†]

Abstract

In parameter estimation, a model function depending on adjustable parameters is fitted to a set of observed data. The parameter estimation task is an optimisation problem, which needs a computational kernel for evaluating the model function values and derivatives. This article presents an object-oriented framework for representing model functions, which are partially separable, or *structural*. Such functions are commonly encountered, e.g., in spectroscopy.

The model is general, being able to cover a range of varying model functions. It offers flexibility at runtime allowing the construction of the model functions from predefined component functions. The mathematical expressions are encapsulated and a close mapping between mathematics and program code is preserved. Also, all interfacing code can be written independently of the particular mathematical formula. These properties together make it easy to adapt the model to different problem domains: only tightly controlled changes to the program code are required.

The paper shows how derivatives of the model function can be computed using automatic differentiation relieving the programmer from writing explicit analytical derivative codes.

The persistence of the objects involved is discussed and finally the computational efficiency of the function and derivative evaluation is addressed. It is shown that the benefits of the object-oriented model, namely the higher abstraction level and increased flexibility, are achieved with a very moderate loss of performance. This is demonstrated by comparing the performance with low-level tailored C-code.

1 Introduction

Even though object-orientation (OO) has become the dominating programming paradigm, it is quite slowly adopted to numerical applications, mainly because of the poor efficiency of OO programs in numerical codes. The progress in programming techniques and compilers is changing this situation and makes it possible to

*This work was supported by the Academy of Finland, grant 37178.

[†]Turku Centre for Computer Science, Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, email: jaakko.jarvi@cs.utu.fi

take advantage of OO in numerical codes without a significant performance penalty [16]. This is demonstrated in this paper describing an OO model for parameter estimation of structural, partially separable functions.

The task of modelling data is commonly encountered in numerous application fields. The goal is to fit a model that depends on adjustable parameters to a set of observed data. A cost function, such as the sum of squared differences, is chosen to measure the agreement between the model and data. This function is minimised by adjusting the parameters of the model according to some optimisation algorithm.

The model can be based on some underlying theory about the data or be just a sum of convenient functions, such as polynomials. This article focuses on partially separable model functions, where the function is a sum of *component functions*, e.g., a spectrum consisting of a sum of spectral lines. The OO model presented in this article was developed while working on nuclear magnetic resonance (NMR) spectra estimation. Hence, the article includes a case study of NMR spectral fitting to make the ideas presented more concrete.

Numerous algorithms have been described for model fitting tasks in the literature [2, 14]. They are usually presented from the numerical analysis viewpoint, treating the model as a plain vector of parameters and a function for evaluating values and derivatives. However, this *flat* representation of the model function is not necessarily natural. The model may be structural consisting of several component functions, which possibly correspond to some real life entities. The function representation should be flexible. It should be possible to specify the composition of the component functions at runtime, rather than fix them in the program code. Furthermore, the function representation should be able to handle dependencies between parameters of different component functions. The flat model representation is therefore inconvenient for the user and it is the application developer's task to provide a conversion to and from the structural representation.

This article presents an OO model to serve as an intermediate link between the two representations described above. The model provides simultaneously an efficient computational kernel for the optimisation algorithms and the structured view for the user. It is a collection of classes comprising a core to represent structured model functions. These core classes implement the basic structural and flat views to the model function, as well as the mechanisms for function value and derivative calculations.

The extension of the core model for a specific application is done by providing a simple class for each type of component function. Essentially only member functions specifying the mathematical formulae of the component functions are required in these classes. Consequently, the particular mathematical expressions are encapsulated and the mathematical structures of the problem domain are preserved in the program code. This means that the necessary changes to program code are minor and well controlled if the model is applied to a different application area.

The model utilises the concept of *automatic differentiation* [15] for derivative computations. This relieves the programmer from writing analytical derivative codes. Automatic differentiation is made transparent to the programmer with operator overloading.

The core classes implement all the functionality needed for constructing component functions and their parameters. The user interface for this task can therefore be built solely based on the core classes. The addition of new classes to the model hierarchy does not cause any need for changes in the interfacing code. In section 3.5 we give an example of a user interface built in this manner.

This paper also discusses the computational efficiency and shows that the overhead arising from the higher abstraction level and greater runtime flexibility of the OO model is very moderate compared with a low-level C-code implementation. Persistence, i.e., the ability to store and retrieve the objects of the model is also considered.

The crucial parts of the model are presented using C++ language, but the model can be implemented in any language supporting inheritance, dynamic binding and operator overloading. However, the test runs were performed using a C++ implementation.

There are few descriptions of using object orientation together with parameter estimation in the literature. Related work can be found from [11, 17] containing general descriptions of computer systems sharing some similarities with our model. For description of an NMR analysis software built using a variant of the object oriented model presented here, see [10].

2 Parameter estimation problem

The task of fitting a parametric model function to a set of observed data points can be seen as minimisation of a cost function describing the distance between the model and the data. A common choice for the cost function is the sum of squares function. This least-squares model fitting problem can be stated as follows:

Let $y(x_i), i = 1, \ldots, m$ be a set of observed data points, $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of model parameters and $\hat{y}(x, \mathbf{p})$ a parameter-dependent model function. The maximum likelihood estimate of the parameters is obtained by minimising the chi-square function

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{m} \left(\frac{y(x_i) - \hat{y}(x_i, \mathbf{p})}{\sigma_i} \right)^2, \tag{1}$$

where σ_i is the standard deviation of the measurement error of the ith data point. This formulation leads to a possibly non-linear optimisation problem which can be solved with iterative methods, most commonly with Levenberg-Marquardt or Gauss-Newton algorithms [2, 14]. The idea is to improve iteratively the trial solution

$$\mathbf{p}_{\mathbf{new}} = \mathbf{p}_{\mathbf{current}} + \mathbf{\Delta p} \tag{2}$$

until an acceptable solution is found. The change $\mathbf{\Delta p}$ is determined using the gradient and usually an approximation of the Hessian of the cost function. These in turn require calculation of the partial derivatives $\frac{\partial \hat{y}(x, \mathbf{p})}{\partial p_s}, s = 1, \ldots, k$ of the

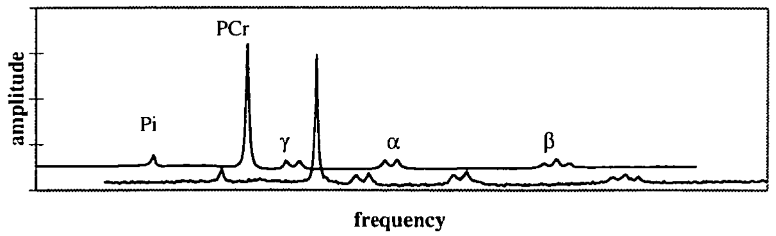


Figure 1: Example of a ^{31}P NMR spectrum (lower curve) and a model function (upper curve) fitted to the spectrum. α, β and γ peak groups originate from ATP molecules. The measured spectrum is shifted rightwards for clarity.

model function. Even though each iteration typically involves additional costs, such as solving a linear system of equations, the calculation of the model function and derivative values often dominate the overall cost.

The above clarifies the numerical view to the parametric estimation problem. The algorithms developed for the estimation must be supplied with the parametric model function, functions for the partial derivatives and the vector of modifiable parameters. Furthermore, $\hat{y}(x, \mathbf{p})$ is typically calculated at several points with constant $\mathbf{p}$. In cases we are interested in, $\hat{y}(x, \mathbf{p})$ is partially separable, that is, $\hat{y}$ can be represented as a sum of component functions $\hat{y}_j, j = 1, \ldots, n$, each being dependent on only r_j parameters, where $r_j << k$.

2.1 NMR spectroscopy case

In NMR spectroscopy, a signal of damping oscillations (FID) emitted by certain atomic nuclei (e.g. ^{31}P) is observed. An NMR spectrum is a Fourier transform of this signal. The spectrum contains peaks or resonance lines corresponding to nuclei in various compounds. The amplitude of a single peak is proportional to the number of equivalent nuclei resonating at that frequency. [6]

A typical ^{31}P NMR spectrum is shown in Fig. 1. Signals of inorganic phosphate (Pi), phosphocreatine (PCr) and adenosine triphosphate (ATP) can be identified from the spectrum. The aim is to find the amplitudes and frequencies of the identified compounds. This is done by quantifying the spectrum or the FID, which is represented as a superposition of parametric functions, each corresponding to a single resonance line. This parametric model function is fitted to the measured signal and the results, peak intensities and frequencies, are calculated from the model parameters. Fig. 1 also shows a fitted model.

Basically we have a structural model function consisting of a sum of component functions, the resonance lines. Several lineshapes are encountered, the most common being the Lorenz function described by amplitude A, frequency f, phase ϕ and damping factor d. A model of n reconance lines in a somewhat simplified form in time domain is then

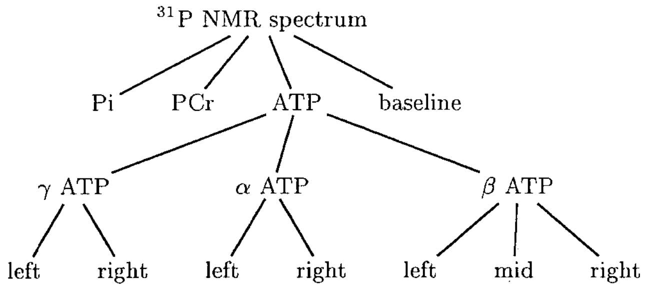


Figure 2: Example of a model function instance.

$$\hat{y}(t, \mathbf{p}) = \sum_{j=1}^{n} A_j \cos(2\pi f_j t + \phi_j) e^{-d_j t}, \tag{3}$$

where $\mathbf{p} = (A_1, f_1, d_1, \phi_1, \ldots, A_n, f_n, d_n, \phi_n)$. As can be seen, the sum function is partially separable. Note that, contrary to this simplified expression, the NMR signal can contain different lineshapes and there may be additional terms in the sum. [5]

Dependencies between parameters of different component functions are typical for NMR models. Consider the ATP molecule. It is known a priori that 7 peaks altogether originate from the ATP molecules. The peaks come in three groups: α, β and γ. These groups have equal amplitudes. The groups α and γ consist of two peaks each having again equal amplitudes. The β-group consists of three peaks with relative amplitudes 1 : 2 : 1. The frequency differences between the peaks inside the groups are known and it is reasonable to assume that the damping factors of all the peaks are equal. Taking these into consideration, the amplitudes, damping factors and frequencies of 7 peaks are actually defined by only one amplitude, one damping factor and three frequency parameters. The hierarchical structure of ATP and other peaks in the NMR example spectrum is depicted in Fig. 2.

To sum up the problem setting, the estimation of the parameters of the function $\hat{y}$ is the task to be performed. This is done by minimising the chi-square error with respect to the measured signal, where the partial derivatives of $\hat{y}$ must be calculated repeatedly. Function $\hat{y}$ has a hierarchical structure corresponding to the peaks in the spectrum.

3 Object-oriented model

Significant savings in development time can be achieved with careful design of the model function representation. In the case of structural model functions, the utmost goal is flexibility. The number and type of the component functions may vary and there may be common or related parameters between the component functions.

The model function representation ought to be able to handle these situations with ease and yet be able to compute the function value and derivatives efficiently.

An important issue is the user interface for managing the model functions. The user constructs the model functions and observes or edits the model parameters. The programmer's task to provide this interface for varying models is considerably alleviated if the interface can be implemented without the need to know the actual types or number of the component functions. The term *user* refers to a human operator of a computer program whereas by *client* we denote the programmer or code calling the functions or using other services of the object-oriented model.

The object-oriented approach provides a convenient means to build a function representation to meet the requirements detailed above. The model consists of two separate class hierarchies, the *function hierarchy* and the *parameter hierarchy*. An essential component is also a library for automatic differentiation. The hierarchies are first discussed accentuating the client view to the classes and then the process of function value and derivative evaluation is clarified. While reading, the reader may consult the object diagram in Fig. 6 representing the NMR example as objects from function and parameter hierarchy.

3.1 Function hierarchy

The classes of the function hierarchy (Fig. 3) represent the component functions of the structural model function (the nodes of the tree in Fig. 2). The base of the hierachy is the abstract base class base_model, which defines the interface for the function classes; each function can compute the value and derivatives at a given point. The base_class maintains a vector of parameters and defines member functions for accessing them. Different component functions are derived from the base_model class. These can be either *elementary* or *composite* functions.

Composite functions maintain a list of other component functions. They simply group other components. A composite function computes its values and derivatives by calling the evaluation functions of its *child* functions. Each composite model *owns* the models in its child list. The top_model class represents the whole model function to be fitted and implements the interface to the client code. It also maintains the vector of the adjustable parameters used by the optimisation algorithm.

The generic elementary_model class encapsulates the common features of the component functions to make the derived classes as simple as possible. The template parameter of the generic class specifies the number of parameters in the function. We will return to the details of this template in section 4. Now it suffices to say that the elementary model holds the parameters of the mathematical function to be calculated as *automatically differentiable numbers* in the proxy data member.

Fig. 4 shows a complete class definition of an example class derived from elementary_model. These derived classes contain the actual mathematical formulae of the model function (the eval function). In addition, only two simple utility functions (create and get_class_name) are needed. These are the only requirements for each elementary function class and it is thus very easy to extend the function hierarchy to cover new function types.

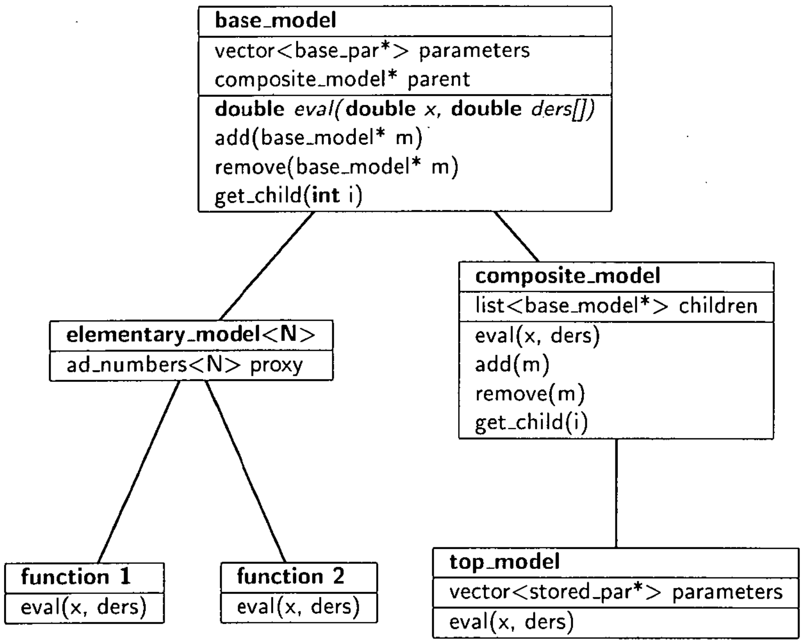


Figure 3: Model function class hierarchy.

Gamma et al. [8] have proposed some general methods for representing hierarchical structures in an object-oriented language. This model function hierarchy can be seen as a version of the *Composite design pattern*. Regarding the implementation issues of this pattern discussed by Gamma et al. we have chosen to maintain explicit parent references implemented as a pointer in the base_model class. We also chose to maximise the interface of the base_model. This means that, e.g., operations for manipulating the list of children of the composite models (add, remove) are declared and defined in base_model. This gives transparency for the client but on the other hand the operations do not have a meaning for elementary models. Therefore, by default, the operations add and remove fail (e.g. by raising an exception) and the functions are overridden in the composite_model class to give them meaningful definitions.

Not all functions are shown in the class diagram of Fig. 3. The base_model class also defines functions for adding and removing parameters as well as functions for naming the models. The *virtual constructor* [8, 1] mechanism is utilised in the object construction, requiring the two virtual functions, create and get_class_name, to be overridden in each derived class.

```
class lorenz : public elementary_model<4> {
public:
  lorenz* create() {return new lorenz(); }
  string get_class_name() {return "lorenz"; }
  enum {amp, freq, damp, ph };
  double eval(double x, vector<double>& ders) {
    return store_derivatives( ders,
      par(amp)*cos(2*pi*par(freq)*x + par(ph)) * exp(-x*par(damp))); }
};
```

Figure 4: Definition of an example function derived from the elementary_model class.

3.2 The parameter hierarchy

The parameter of a model function is basically just a value of some floating point type. However, the same parameter value may be shared by several component functions or there may be other dependencies between parameters. Hence, not all parameters of the component functions store a value. As a consequence, just representing a parameter as a floating point number is not sufficient to allow the component models to use the parameters in a uniform way. Therefore parameters are represented as classes from the parameter hierarchy (Fig. 5).

The base_par class is the topmost class of the hierachy and provides the common interface, the functions get_value and get_derivative for retrieving the value and initial derivative of the parameter. The stored_par class represents actually stored, adjustable parameters. The dependent_par class is the base class for dependent parameters and linear_par is for expressing linear relations between parameters. Other dependencies may be implemented by deriving new classes from the dependent_par class.

Each dependent parameter holds a pointer to another parameter, a *parent* parameter. The value is resolved by asking the value of the parent recursively until finally an instance of a stored_par class will end the recursion. The same mechanism applies for derivatives. The get_derivative function evaluates the derivative with respect to the underlying stored parameter. For stored_par this is 1 (the derivative of a variable with respect to itself is 1), while for linear parameters we get it by multiplying the derivative of the parent with the linear factor (see the code outlined in Fig. 5).

All parameters also maintain a child list and a pointer to the model function owning the parameter. The dependent parameters contain a vector of *parameter modifiers* such as the coefficients of the linear relation. The number of these modifiers is fixed for each derived class and given in the constructor.

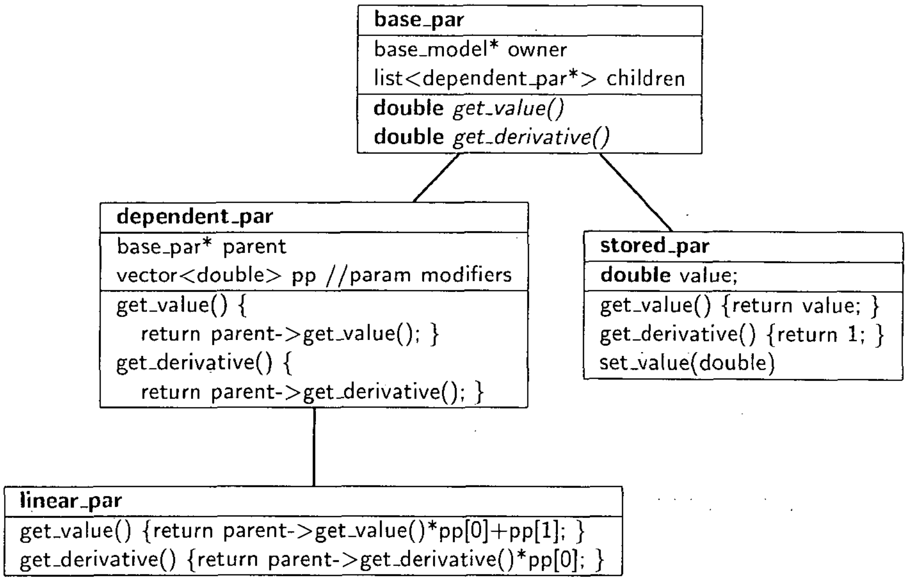

base_par
base_model* owner
list<dependent_par*> children
double get_value()
double get_derivative()
dependent_par
base_par* parent
vector<double> pp //param modifiers
get_value() {
return parent->get_value(); }
get_derivative() {
return parent->get_derivative(); }
stored_par
double value;
get_value() {return value; }
get_derivative() {return 1; }
set_value(double)
linear_par
get_value() {return parent->get_value()*pp[0]+pp[1]; }
get_derivative() {return parent->get_derivative()*pp[0]; }

Figure 5: Parameter hierarchy.

3.3 Enforcing the consistency

The data structure for representing structural functions consists of several objects from the function and parameter hierarchies (Fig. 6). It is a combination of two object trees, both maintaining child node lists and parent pointers. In addition, the nodes of the model tree may own nodes of the parameter tree. This relation is represented as a list in the model tree node and a corresponding owner pointer in the parameter tree node. Furthermore, a vector of references to the adjustable parameters in the parameter hierarchy is maintained in the topmost model function.

To be able to guarantee the consistency of such a complex structure the construction and manipulation of the objects involved in the data structure must be controlled tightly. Though not shown in the class definitions, the creation and destruction of models is not part of the public interface of the classes, instead the creation of the objects is delegated to a special *creator* object and the destruction is performed from within the member functions of the classes of the hierarchy.

The final data structure maintains several invariances. The child list of a composite model is kept consistent with the parent references of the children. The same applies to child/parent relation in the parameter hierarchy as well as the parameter/owner relation between model functions and parameters.

The relation between parameters and models is further restricted. A parameter and its descendant can not be owned by different function hierarchies. Furthermore, the owner function of a dependent parameter must be a descendant of the owner of the parent parameter. Some of the invariances are guaranteed automatically by

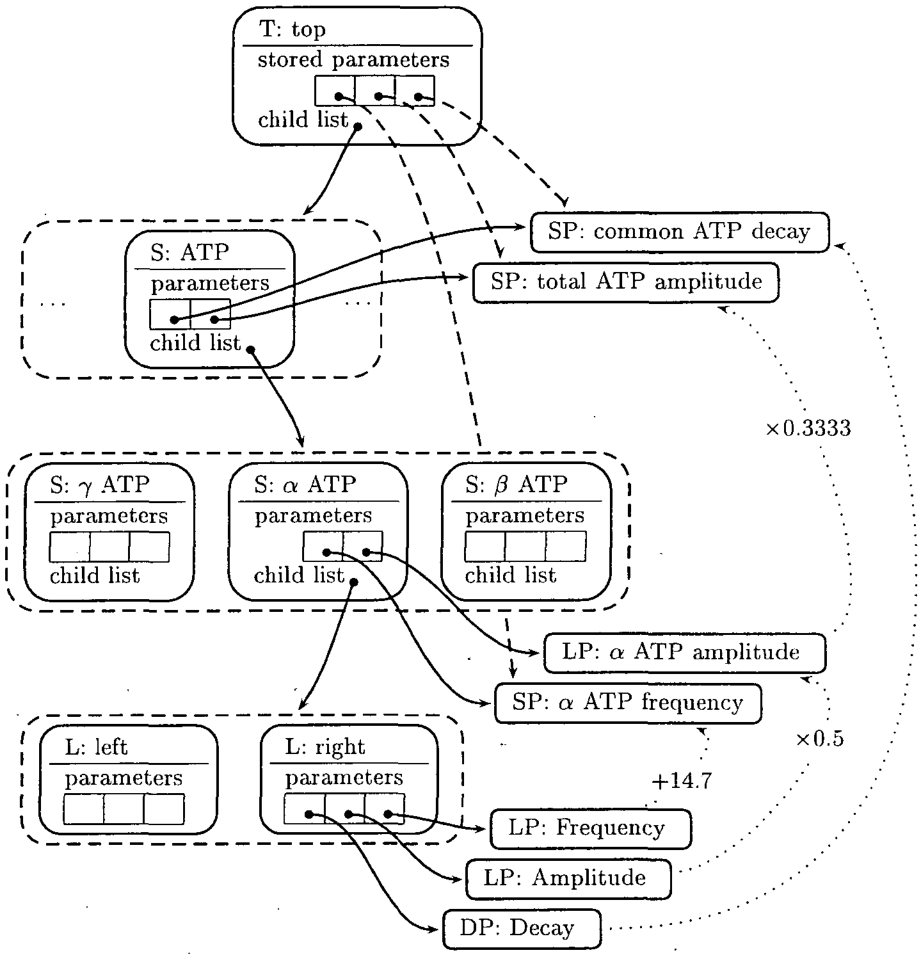


Figure 6: Instantiated objects and their relations illustrated in the NMR case (part of the ATP molecule). Solid lines represent ownership relation, while dashed lines are non-owning pointers. Dotted lines are parent links. The class of each object is given in parenthesis (C=composite, L=lorenz, LP=linear parameter, DP=dependent parameter, SP=stored parameter). Along the parent links of the parameters are the formulas for computing the values of linear parameters.

the restricted object construction. Others are enforced by raising an exception if a user tries to perform an operation which conflicts with an invariance condition.

3.4 Constructing model functions

The starting point of a model function is an instance of the top_model class. After it has been created, component models can be added to its child list.

The construction of objects is delegated to a special creator object implementing the virtual construction mechanism. The purpose of this is to make the client code, which initiates the creation of objects, independent of the changes in the elementary function classes.

The class of the object to be created is specified as a class name string at runtime. This is a convenient way of initiating object construction. Since the object creation task is most likely initiated by a user command, it is quite natural to specify the class as a class name string. The user may, e.g., have selected the class from a selection list.

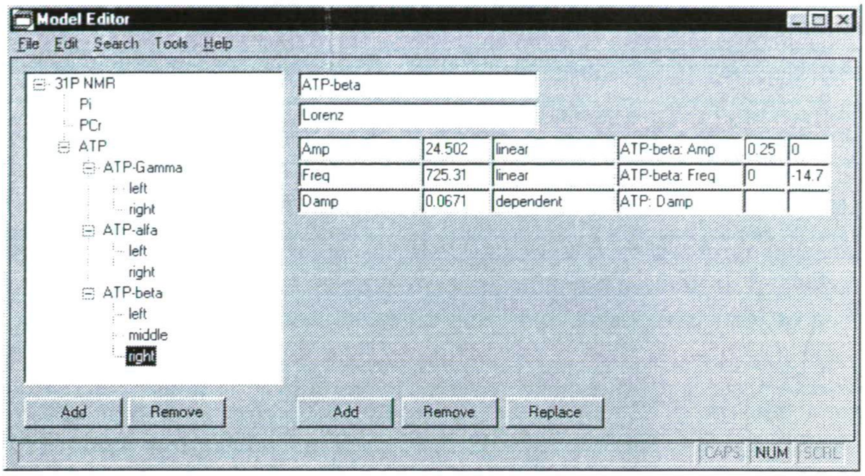
The creation mechanism requires each class to *register* itself (one line of code) and define the virtual functions get_class_name and create. Otherwise the creation mechanism is totally independent of the derived classes: e.g., adding new elementary functions to the model hierarchy does not have any effect on the client code. For details of the virtual construction mechanism, see [8] describing several creational design patterns.

3.5 Model editor

As an example of a user interface for specifying structural model functions, Fig. 7 shows a snapshot of the model editor we have written. The structural function tree is visible on the left and the parameters of the currently selected model on the right. The names of the functions as well as the parameter names and values can be edited freely on the spot. There are buttons and menu commands for adding and removing functions and parameters, defining relations between parameters and storing and retrieving models. As pointed out above, the code of the model editor is totally independent of the particular elementary function classes derived from the model function hierarchy.

3.6 Persistence of model objects

In addition to methods for creating and modifying the structural model functions, means for their storage and retrieval are needed. In object-oriented systems, the ability of objects to live beyond the lifetime of the program is called persistence. It can be achieved using *object serialisation*, the common approach used in commercial class libraries such as MFC [12] and OWL [13]. This approach relies on virtual construction mechanism and requires the programmer to specify reading and writing methods for each persistent class.



ATP-beta

Lorenz

Amp	24.502	linear	ATP-beta: Amp	0.25	0
Freq	725.31	linear	ATP-beta: Freq	0	-14.7
Damp	0.0671	dependent	ATP: Damp		

Figure 7: Snapshot of our model editor.

Virtual construction is already included in the model and parameter hierarchies. Furthermore, when the model hierarchy is extended, no new data members need to be introduced in the derived elementary function classes. Therefore the read and write functions can be inherited and need not be specified. Consequently, the implementation of persistence can be encapsulated entirely into the core classes of the model and no changes are required when new classes are added to the hierarchies.

4 Evaluation of function values and derivatives

In this section, the function value and derivative computation in our model is explained. The concept of automatic differentiation is described and it is shown how to calculate the derivatives of structural functions easily and yet effectively with this technique.

4.1 Automatic differentiation

The derivatives are traditionally calculated either symbolically or by using divided differences. The former may be quite difficult and error-prone while the latter introduces truncation errors and may be inaccurate and inefficient. Automatic differentiation provides an appealing alternative.

In automatic differentiation, the derivatives are computed by the well-known chain rule, but instead of propagating symbolic functions, numerical values are propagated along the computation. The evaluation of the function and its derivatives are calculated simultaneously using the same expressions. There are several descriptions about automatic differentiation [15, 1, 3] and also software packages available [9, 4]. Some packages preprocess the source code to add the necessary statements for computing the derivatives. Other packages, using programming languages that support operator overloading, implement the differentiation as a class library without the need for a separate precompilation.

There are interesting computational issues concerning the implementation of automatic differentiation. The chain rule can be used either in *forward* or *backward* mode or in something between. The implementation involves a tradeoff between time and space complexity. In this article the forward mode automatic differentiation is used. It is simple and fits very well in this particular application as will become clear below.

In forward mode automatic differentiation, instead of computing with scalar values, we compute with automatically differentiable numbers (ADN) $\langle \mathbf{f}, \nabla \mathbf{f} \rangle$. An ADN consists of a value and a vector of partial derivatives of a function at a given point. When building expressions with these objects, at the leaf level of the expression tree $\mathbf{f}$ is either a variable or a constant. When differentiating with respect to N variables, the derivative of the ith variable is represented as the ith canonical unit vector of length N and the derivative of a constant with a zero vector. For example, when differentiating with respect to three variables x, y, z the constant 3.14 is expressed as $\langle 3.14, (0,0,0) \rangle$ and the variable y as $\langle y, (0,1,0) \rangle$. Computation with these objects utilises the chain rule of derivatives.

$$\frac{\partial}{\partial t} f(g(t)) \bigg|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s) \bigg|_{s=g(t_0)} \right) \left(\frac{\partial}{\partial t} g(t) \bigg|_{t=t_0} \right) \tag{4}$$

As an example, consider the two-derivative case for function $y + \sin(x^2)$. Starting with $\langle y, (0,1) \rangle + \sin\langle x, (1,0) \rangle^2)$ by squaring x, we get $\langle y, (0,1) \rangle + \sin(\langle x^2, (2x, 0) \rangle)$. Taking the sine gives $\langle y, (0,1) \rangle + \langle \sin(x^2), (2x\cos(x^2), 0) \rangle$ and finally the addition with y gives $\langle y + \sin(x^2), (2x \cos x^2), 1) \rangle$. For numerical work, the computation is not done symbolically, rather the actual values of the function and its derivatives are calculated and propagated through the expression. Given $x = 2, y = 4$ the same example becomes

$$\begin{aligned} \langle 4, (0,1) \rangle + \sin(\langle 2, (1,0) \rangle^2) &= \langle 4, (0,1) \rangle + \sin(\langle 4, (4,0) \rangle) = \\ \langle 4, (0,1) \rangle + \langle 0.06976, (1.9951, 0) \rangle &= \langle 4.06976, (1.9951, 1) \rangle. \end{aligned}$$

The method can be applied to any machine-computable function. All that is needed is to code the differentiation rules for simple functions and operations. Then any function composed of those elementary functions can be differentiated automatically. In C++ this means overloading common functions and operators for objects described above.

The forward mode automatic differentiation for calculating gradients can be computationally unattractive if applied blindly. If the gradient has n elements, the computation may require up to order of n as much time as computating the value of the same expression. However, in the case of partially separable functions the forward mode can be applied efficiently. If we consider the model function as a whole, it may have quite a number of parameters, but the number of parameters of the individual elementary functions is typically rather low and known beforehand. Furthermore, different elementary functions are only related via a summation expression, which means that also the derivatives are just summed together. Consequently, we use automatic differentiation in computing the local gradients of the elementary functions and update the calculated values via pointers to the common derivative vector.

The computing time of the local gradients can be further reduced. By using C++ templates, moderate size derivative vectors of ADNs can be replaced with special sparse vectors to yield very efficient code [10]. This method was utilised in the test runs described in section 4.4.

4.2 The function evaluation process

The model function is evaluated by calling the eval function of the topmost class, which will traverse all the contained models and calculate their cumulative values at a given point. The derivatives are computed simultaneously using automatic differentiation. The derivative vector is passed as a parameter to the eval function. First the resulting derivative vector or gradient is initialised to zero. Each elementary function reads the values and initial derivatives of the parameters (with the get_value and get_derivative functions) and constructs ADNs from them. If the elementary function has n parameters, ADNs having n-dimensional derivative vectors are used. The mathematical expression is then evaluated using ADNs and each elementary function updates the resulting derivative values to the actual gradient vector. This is accomplished with a call to store_derivatives function defined in the elementary_model template (see Fig. 4), which adds the derivative values to the right positions of the gradient.

After the whole function tree has been traversed, the function value is returned and the gradient is available as the derivative vector passed to the eval function.

4.3 Computational efficiency

With regard to the computational efficiency the evaluation strategy includes a few pitfalls. Firstly, dynamic binding is applied in the eval function invocations. There is an inherent additional cost in a call to a dynamically bound virtual function compared with a statically bound function [7]. Furthermore dynamic binding precludes the use of inlined functions. Inline expansion can speed up function calls and is beneficial for small functions. However, in this case the computational cost of the function call is probably minor compared to the cost arising from the evaluation of the actual mathematical formulae of the elementary functions, recalling that the derivatives are also calculated in the same function. Considering this, the relative cost of the slightly slower function call is most likely insignificant in this case.

Secondly, dynamic binding is also applied between parameters in the get_value and get_derivative functions. In this case the extra cost may be notable. The evaluation of an elementary function having n parameters would yield at least n virtual function calls to fetch the parameter values. The number of calls is larger if dependent parameters are involved. However, in model fitting tasks, the model function is evaluated repeatedly at several points, without changing the parameter values. Taken the example from NMR spectroscopy, the region of interest may contain thousands of points. Therefore the parameter values can be cached and only when the parameter values are changed, each elementary function reads the values and derivatives with the virtual get_value and get_derivative functions and stores the values to local proxy variables (ADNs). With this approach the relative cost of retrieving the parameter values via virtual functions is of little consequence. The caching is made transparent to the client code by maintaining a flag in the topmost class indicating whether the values in the proxy variables are valid or not.

Also, the updating of the local gradients to the global derivative vector must be efficient. This is implemented in the elementary_model template by maintaining a mapping from each local parameter index to an index in the global derivative vector. These mappings can be constructed prior to the first model evaluation. In this task the function tree must be traversed once, but this causes no efficiency problems, since the indices only change if the model function changes, i.e., new component functions are added or removed. At evaluation time the only additional cost is an extra indirection for each parameter.

Some cost may also arise if the composite models in the function tree contain many levels (e.g. in the ATP compound). From the computational point of view, it is not necessary to traverse all composite functions during the evaluation, rather it is sufficient to call the evaluation functions of the elementary models directly and save the cost of a few virtual calls. This is easy to implement by maintaining a separate list of the leaf nodes in the top_model class, which we did in the test runs.

4.4 Test runs

To assess the efficiency of the model some test runs were performed. As a test case, we used formulae from the NMR case consisting of 10 component functions having 24 adjustable parameters altogether. Five different alternatives to perform the function and derivative computations were programmed:

1. A tailored low-level C-code with analytical derivatives.
2. The presented OO model with analytical derivatives.
3. The OO model with automatic differentiation.
4. A straightforward OO implementation, without any caching.
5. A low-level implementation of the function with finite difference value approximations of the derivatives.

In the tailored low-level implementation, the model function was totally fixed at design time, so any change in the function requires changes in the code. The code

Implementation	Relative time
1. Tailored low level C-code	1.00
2. OO model with analytical derivatives	1.07
3. OO model with automatic differentiation	1.29
4. Straightforward OO implementation	2.48
5. Divided difference approximations	16.52

Table 1: Relative evaluation times of the different methods for computing the value and derivatives of the NMR model.

was hand-optimised to a reasonable level (not making any processor specific tricks). All subexpressions were calculated only once and all relations between parameters were directly written into the code as effectively as possible. It is fair to say that the code used was as fast as possible.

In the second case, the OO model presented was used, but the derivatives of the elementary functions were calculated analytically. This case should roughly represent the extra cost originating from the dynamic binding of the model functions, as well as the cost arising from not coding the dependencies between the parameters directly.

In the third case, the OO model was used with derivatives computed using automatic differentiation. Table 1 shows the results and confirms the extra cost being quite acceptable compared with the flexibility the model offers.

In the fourth case, no proxies for parameters were used, rather the initial values and derivatives were retrieved during each evaluation using the virtual function invocations. This demonstrates that the performance may drop significantly if the programmer is not aware of the principles affecting efficiency in OO programs.

In the fifth case, derivatives were approximated with divided difference values. The benefit of this alternative is that only the code for evaluating the value of the model function is needed. The performance is, however, very poor requiring n evaluations of the model function, where n is the number of elements in the gradient. Also, the accuracy is harder to assess.

The test runs were performed under Linux on Intel Pentium processor. The C++ compiler used was KAI C++ 3.2.d with optimisation flags +K2 -O3.

5 Conclusions

An object-oriented model for parameter estimation of partially separable function was described. The model achieves two goals. Firstly it gives an easily extendible OO framework for representing partially separable functions in a structured way, resembling the physical real-life interpretation and mathematical structure of the functions. Secondly, it offers an interface to an optimisation algorithm, namely a vector of adjustable parameters and a function capable of computing the value and derivatives of the model function efficiently.

To achieve the first goal, the model separates the commonalities of partially separable functions from the specific mathematical formulae. The formulae are encapsulated to a few very simple classes. It is therefore easy to apply the model to different problem domains, since changing these classes or adding new ones to the model does not affect the client code using the model. Furthermore, relations between parameters are handled by the model and they do not complicate the mathematical expressions of the component functions.

The derivatives needed in the parameter estimation are obtained using automatic differentiation. Hence, there is no need to hand-code analytical derivatives or use divided difference values.

Considering the second goal, the calculation of function values and derivatives is efficient. In our example case from NMR spectroscopy, the evaluation of the OO model required only 29% more time than a low-level tailored implementation of the same function. As a compensation, in the OO model the final function as well as relations between parameters can be specified at run-time, the model is easily extendible to cover new component functions and no hand-coded derivatives are required.

To sum up, the paper gives practical guidelines for implementing an efficient OO computational kernel for partially separable functions. With an example, we showed that OO programming offers substantial benefits, such as higher abstraction level, code reuse, flexibility and handling of complexity for numerical programming as well. Furthermore, the advantages can be achieved with a moderate loss of performance.

References

[1] Barton J. J., Nackman L. R.: Scientific and Engineering C++, Addison-Wesley, Reading Massachusetts 1994.

[2] Bazaraa M.S., Sherali H.D., Shetty C. M.: Nonlinear Programming: Theory and Algorithms, 2nd Edition, Wiley 1993.

[3] Editors: Berz M., Bischof C. H., Corliss G. F., and Griewank A.: Computational Differentiation - Techniques, Applications, and Tools,.SIAM, Philadelphia Pennsylvania 1996.

[4] Bischof C. H., Carle A., Corliss G. F., Griewank A., Hovland P.: ADIFOR: Generating derivative codes from Fortran programs, Scientific Programming, 1 (1992) 1-29.

[5] Bovée W. M. M. J.: Quantification in in vivo NMR, Spectral editing, in: Magnetic Resonance Spectroscopy in Biology and Medicine, eds. de Certaines J. D., Bovée W. M. M. J., Podo F., 181-207, Pergamon, Oxford 1992.

[6] Derome A. E.: Modern NMR Techniques for Chemistry Research, 63-90, Pergamon, Oxford 1991.

[7] Driesen K., Hölzle U.: The Direct Cost of Virtual Function Calls in C++, ACM Sigplan Notices, OOPSLA'96 Proceedings, **31** (1996) 306-323.

[8] Gamma E., Helm R., Johnson R., Vlissides J: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading Massachusetts 1995.

[9] Griewank A., Juedes D., Utke J.: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, ACM Transactions on Mathematical Software, **22** (1996) no.2, 31-167.

[10] Järvi J.: A PC program for automatic analysis of NMR spectrum series, Computer Methods and Programs in Biomedicine **52** (1997) 213-222.

[11] Majoras R. E., Richardson W. M., Seymour R. S.: An object-oriented approach to evaluating multiple spectral models, Journal of Radioanalytical and Nuclear Chemistry **193** (1995) 207-210.

[12] Microsoft, Microsoft Foundation Class Library, Microsoft Corporation.

[13] Borland, Borland C++ 5 Programmer's guide, Borland International, 1996.

[14] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P.: Numerical Recipes in C: the Art of Scientific Computing, 2nd Edition, Cambridge University Press, New York 1992.

[15] Rall L.B.: Automatic differentiation: Techniques and Applications, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin 1981.

[16] Robinson A.D.: C++ Gets Faster for Scientific Computing, Computers in Physics **10** (1996) 458-462.

[17] van Tongeren B. P. O., Boxman R. D. C., Deumens J. W., van Leeuwen J. P., Mehlkopf A. F., van Ormondt D., de Beer R.: QUANSIS, An object-oriented data-analysis system for in vivo NMR signals, Journal of Magnetic Resonance Analysis, **2** (1996) 75-84.