EXPHER (EXperimental PHysics ERror analysis): a Declaration Language and a Program Generator for the Treatment of Experimental Data

Philippe Weber, Daniel Taupin

▶ **To cite this version:**

Philippe Weber, Daniel Taupin. EXPHER (EXperimental PHysics ERror analysis): a Declaration Language and a Program Generator for the Treatment of Experimental Data. Journal de Physique III, 1995, 5 (5), pp.605-622. 10.1051/jp3:1995149 . jpa-00249334

HAL Id: jpa-00249334
https://hal.science/jpa-00249334

Submitted on 4 Feb 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

J. Phys. III France **5** (1995) 605–622 MAY **1995**, PAGE 605

Classification
Physics Abstracts
02.90P

EXPHER (EXperimental PHysics ERror analysis): a Declaration Language and a Program Generator for the Treatment of Experimental Data

Philippe Weber and Daniel Taupin

Laboratoire de Physique des Solides, Université de Paris-Sud, 91405 Orsay cedex, France

(Received 10 June 1994, revised 4 January 1995, accepted 24 January 1995)

Résumé. — EXPHER est à la fois un logiciel — compilateur et générateur de programme, avec une bibliothèque associée — facilitant le traitement d'une série de données expérimentales associée à un modèle physique et un langage de spécification du comportement de l'appareillage expérimental. À partir de cette description de l'appareillage, EXPHER produit un programme compilable dont l'exécution détermine les valeurs les plus probables des paramètres du modèle physique, leur barre d'erreur et la matrice d'erreurs finale.

Abstract. — EXPHER refers to both a package — program generating compiler and library — which facilitates the treatment of a set of experimental data associated to a physical model, and a specification language describing the behaviour of the experimental device. Starting from this description of the experimental devices, EXPHER yields a compilable program whose execution determines the most likely values of the parameters involved in the physical model, but also their error bars and their comprehensive error matrix.

1. Introduction

Any physicist, chemist, biologist and even a sociologist, who wants to deduce scientific information from a set of experimental measurements he has performed, is faced to three problems:

i) Establish a *theoretical* model describing the behaviour of his experimental device and of his samples, that is, establish a relationship between the unknown values $\mathbf{X}$ he is looking for and the data $\mathbf{E}$ he can observe.

ii) Establish an algorithm to determinate the *best* values of $\mathbf{X}$ knowing the measurements $\mathbf{E}_0$ actually observed.

iii) Build a program — nomatter the language, Fortran, C, Pascal or ADA — which performs this algorithm.

© Les Editions de Physique 1995

Any scientist knows that no experimental measurement is perfectly precise and that each measure *deviates* from the expected value (from its *expectation* in probabilistic language) by a quantity which in turn depends on the physical phenomena involved. This uncertainty may be due to an imprecise reading, to intrinsic noises (electromagnetic noises in transmission wires), to internal instabilities (electronic white noise during signal amplification) or to a fundamental uncertainty of the phenomena (quantic phenomena or particle counting noise). The consequence is that the theoretical model which describes an experiment is intrinsically probabilistic.

No apparatus gives the final probability law of the measured quantities but, fortunately, physical experiments and measurement devices exhibit a behaviour which is rather well known and which can be modellized... provided the experimentalist is not too lazy. Thus, owing to both the phenomenologic knowledge of the apparatus and the knowledge of the phenomenon under study, one can nearly always establish a theory of the information flow, i.e. establish a relationship which describes the probability laws of the outputs (that is, the measurements $\mathbf{E}$) *knowing* the inputs (i.e. the quantities $\mathbf{X}$ one wants to determine).

In other words, the experimentalist is nearly always able to establish the probability law of the measures $\mathbf{E}$ (of which he knows a sampling set $\mathbf{E}_0$) *knowing* the values of the *unknowns* $\mathbf{X}$ (which he does not know, obviously!). The whole of the problem is to obtain the inverse probabilities, i.e. to determine the probability law of the inputs $\mathbf{X}$ knowing the actual outputs $\mathbf{E}_0$.

Solving such a problem is sometimes trivial but often infeasible, so that physicists often revert to heuristic and questionable methods or to statistical inconsistencies which would frighten many a specialist of probability calculus. And when a rigorous treatment is established, the resulting algorithm — that is, the program — is usually valid only under a limited range of conditions, so that a slight change, either in the parameters or in the formulation, can lead to many changes in the program, which may be not only tedious but the cause fo further errors.

Describing the behaviour of an experimental device, i.e. the *probability law*[1] $\mathcal{P}(\mathbf{E}|\mathbf{X})$, is if the exclusive responsibility of the experimentalist, of course with the possible collaboration of a theoretician, but this can obviously not be automatised.

Conversely, once the behaviour of the device has been clearly described with some specification language, the algorithm yielding the inverse probability $\mathcal{P}(\mathbf{X}|\mathbf{E}_0)$ or its main characteristics can be derived — in most cases — in a quasi-automatic manner, provided one takes account of some elementary theorems of the probability calculus. Then, if the initial problem is correctly specified, programming this algorithm can usually be considered and it often appears quite feasible.

Thus, summarizing these three stages:

- we created a specification language (EXPHER) to specify the physical and statistical behaviour of an experimental measuring device;
- we created a software package of the same name, which compiles the description — in the EXPHER language — of this apparatus to produce a usual program — provisionally in Fortran 77 which is the programing language usually understood by physicists — which determines the most likely values of the unknowns $\mathbf{X}$, *knowing* the vector of the measurements actually recorded $\mathbf{E}_0$, and computes the resulting *uncertainties* of these unknowns (in the common language of physicists: their *error bars*);

(1) In general, we denote $\mathcal{P}(A|B)$ the probability of having the event A *knowing* the event B. By extension A or B may be the values of random variables or current vectors in a space of random variables.

- after that, one only needs to compile and execute the program produced by the EXPHER package, giving to it the experimental measurements as input data.

Remark: At a first look of this paper, many a scientist might object: "All that is just curve fitting, which is provided in most mathematical packages!" Indeed we looked at some of these packages, such as Mathematica [1], ACE/gr [2] or Kaleidagraph [3], and we found out that the proposed curve fitting does neither handle non-constant data variances, nor experimental data organised into several curves or spectra, nor does it make an error estimation of the coefficients output from the fit.

2. Experimental Measurements: a Probabilistic Phenomenon

2.1. THE TRANSFER FUNCTION AND THE UNIQUENESS OF THE "OUTPUT". — By definition, the *transfer function* $q_E(\mathbf{e}|\mathbf{X})$ of an experimental device describes the probability of observing the measurements(2) $\mathbf{e} \in \{\mathbf{e}\}_N$ "knowing" the M parameters $\mathbf{X} \in \{\mathbf{x}\}_M$ ruling the phenomenon(3).

The uniqueness of the "measurements" vector $\mathbf{e}$ deserves being emphasised: in fact, all the experimental data related to correlated experiments must be considered *as a whole* in the probabilistic sense; therefore, they must be recorded (as distinct elements) in a unique vector. This applies not only to the whole set of the measurements building — for example — a "spectrum", but this also applies to multiple and/or successive measurements of the same quantity. The reason is that — whatever the pertinence of the derivation algorithm — the resulting probability law of the unknowns $\mathbf{X}$ will depend on the whole of all the measures used for that determination.

In fact, a common practice in experimental sciences consists in treating separately the *groups of measurements* and, after that, to take the average of the results obtained; such a separate treatment is statistically questionable since it postulates the independence of the various groups of measures, leading to erroneous estimations of the final uncertainties and, even more, to erroneous estimations of the unknowns.

In general $\mathbf{E}_0 \in \{\mathbf{e}\}_N$ has many more elements than $\mathbf{X} \in \{\mathbf{x}\}_M$ but $\mathbf{X}$ may also have more; this will require additional hypotheses, sometimes "fuzzy", about $\mathbf{X}$.

Remarks:

- *We presently assume that the choice of the vectorial space $\{\mathbf{x}\}_M$ of the unknowns is obvious. In fact, this is a severe difficulty in most physical problems, and there the scientific knowledge of the physicist is of major importance.*
- *The function $q_E(\mathbf{e}|\mathbf{X})$ is a probability law (possibly a density) ruling $\{\mathbf{e}\}_N$, not $\{\mathbf{x}\}_M$. Therefore, a coordinate change of $\mathbf{X}$ does not change $q_E(\mathbf{e}|\mathbf{X})$ (no Jacobian); this is not the case of a coordinate change of $\mathbf{e}$.*

2.2. FROM THE TRANSFER FUNCTION TO THE PROBABILITY LAW OF THE UNKNOWNS
This is the central problem of scientific experiments. It can be expressed as follows: once the probability of $\mathbf{E}$ knowing $\mathbf{X}$ has been established, how to get the probability law of $\mathbf{X}$ knowing the actual measurements $\mathbf{E} = \mathbf{E}_0$? The answer comes from the famous Bayes Formula, i.e.:

$$\mathcal{P}(\mathbf{E}, \mathbf{X}) = \mathcal{P}(\mathbf{E}|\mathbf{X})\mathcal{P}(\mathbf{X}) = \mathcal{P}(\mathbf{X}|\mathbf{E})\,\mathcal{P}(\mathbf{E}) \tag{1}$$

(2) **Bold** lower case letters denote any column vector. $\mathbf{e}$ denotes here the current vector of the N-dimensional space $\{\mathbf{e}\}_N$ of the possible measurements.

(3) **BOLD** upper case letters denote a column vector of random variables.

2.2.1. General Case. — The Bayes Formula (1) needs introducing an *a priori* probability law of $\mathbf{X}$, i.e. the information we know about $\mathbf{X}$ before the experiment is done; we denote it $t_{\mathbf{X}}(\mathbf{x})$. Then, classic probability calculus yields the real goal of an scientific experiment:

$$\mathcal{P}(\mathbf{X} = \mathbf{x}|\mathbf{E} = \mathbf{E}_0) \equiv r_{\mathbf{X}}(\mathbf{x}|\mathbf{E}_0) = \frac{q_{\mathbf{E}}(\mathbf{E}_0|\mathbf{x})\, t_{\mathbf{X}}(\mathbf{x})}{\int q_{\mathbf{E}}(\mathbf{E}_0|\mathbf{x})\, t_{\mathbf{X}}(\mathbf{x})\, \mathrm{d}\mathbf{x}} \tag{2}$$

The whole of the data treatment, of the data reduction and of the error analysis is contained in equation (2) above.

2.2.2. The Case of $t_{\mathbf{X}}(\mathbf{x})$ "Flat". — If nothing or nearly nothing is *a priori* known about $\mathbf{X}$, then its *a priori* probability law is uniform, i.e. $t_{\mathbf{X}}(\mathbf{x})$ is constant, at least in the interesting domain of $\mathbf{x}$. Then:

$$\mathcal{P}(\mathbf{X} = \mathbf{x}|\mathbf{E} = \mathbf{E}_0) \equiv r_{\mathbf{X}}(\mathbf{x}|\mathbf{E}_0) = \frac{q_{\mathbf{E}}(\mathbf{E}_0|\mathbf{x})}{\int q_{\mathbf{E}}(\mathbf{E}_0|\mathbf{x})\, \mathrm{d}\mathbf{x}} \tag{3}$$

2.3. Studying the Post-Experimental Probability Laws. — This merely consists in studying the mathematical properties of $r_{\mathbf{X}}(\mathbf{x}|\mathbf{E}_0)$, in two steps in practice:

i) Seeking its maximum with respect to $\mathbf{x}$ in order to localise the region where the probability of having $\mathbf{X}$ is important. This is called "seeking the *maximum likelihood*".

ii) Estimating the "width" of this maximum, i.e. estimating the *confidence region* or — more simply speaking — estimating the uncertainties.

2.3.1. The Maximum Likelihood. — Seeking the maximum of the *a posteriori* priority law of $\mathbf{X}$ (i.e. the *maximum likelihood*) deserves a comment: since this maximum is not invariant under a non-linear space change of $\mathbf{X}$, the relevance of such a seek may be questioned. But in fact, the interesting thing is not its accurate location, but the region of $\{\mathbf{x}\}_M$ where the *a posteriori* probability law is significant. Conversely, this non-invariance makes it useless to determine the position of that maximum with a precision 100 times better than the width of the region, since final uncertainty of $\mathbf{X}$ is roughly the sum of the width of this significant region, plus the inaccuracy of the location of its maximum.

For practical and numeric reasons, one usually deals with the logarithms of these probabilities([4]) so that the general equation of the maximum likelihood $\mathbf{x}_0$ is derived from equation (2):

$$\left[\frac{\partial}{\partial \mathbf{x}}[\log q_{\mathbf{E}}(\mathbf{E}_0|\mathbf{x})] + \frac{\partial}{\partial \mathbf{x}}[\log t_{\mathbf{X}}(\mathbf{x})]\right]_{\mathbf{X}=\mathbf{X}_0} = \mathbf{0}^{\bullet} \tag{4}$$

2.3.2. Estimating the Errors. — Although this is an unfortunate common practice, determining the maximum likelihood without estimating the errors i.e. without evaluating the size of the *confidence region*, is irrelevant. This *error computation* — in common language — is done by evaluating the matrix([5]) of the *second derivatives* of the *a posteriori* probability law [8].

$$\overline{\Sigma^2} = -\left[\frac{\partial^2}{\partial \mathbf{x}^2} \log r_{\mathbf{X}}(\mathbf{x}|\mathbf{E}_0)\right]^{-1}_{\mathbf{X}=\mathbf{X}_0} \tag{5}$$

Then, the *independent error* of the i-th component of $\mathbf{X}$ is merely the i-th diagonal element of $\overline{\Sigma^2}$.

([4]) The symbol $^{\bullet}$ represents the transposition of a matrix or a vector.

([5]) We denote *matrices* with upright characters with an over-bar; e.g.: $\overline{\mathrm{C}}$, $\overline{\Sigma^2}$.

2.3.3. The Case of $t_X(\mathbf{x})$ Implying Constraints. — The previous treatment assumes $t_X(\mathbf{x})$ to be derivable, but it may include constraints represented with δ functions:

$$t_X(\mathbf{x}) = t_X^o(\mathbf{x}) \prod_{k=1}^{K} \delta(c_k(\mathbf{x})) \tag{6}$$

Then the equation (4) of the maximum likelihood becomes:

$$\left[\frac{\partial}{\partial \mathbf{x}} \log q_E(\mathbf{E}_0|\mathbf{x}) + \frac{\partial}{\partial \mathbf{x}} \log t_X^o(\mathbf{x}) - \mathbf{\Lambda}^\bullet \frac{\partial}{\partial \mathbf{x}} \mathbf{C}(\mathbf{x})\right]_{\mathbf{x}=\mathbf{x}_0} = \mathbf{0}^\bullet$$
$$\mathbf{C}(\mathbf{x}_0) \equiv \{c_k(\mathbf{x}_0)\} = \mathbf{0} \tag{7}$$

where the $\mathbf{\Lambda}$ are Lagrange multiplicators, and the error matrix $\overline{\Sigma^2}$ is now:

$$\overline{\Sigma^2} = \left[\begin{pmatrix} \overline{\Sigma^2}_{r^o}^{-1} & \frac{\partial \mathbf{C}}{\partial \mathbf{x}}^\bullet \\ \frac{\partial \mathbf{C}}{\partial \mathbf{x}} & \overline{\mathbf{0}} \end{pmatrix}^{-1}\right]_{terms[1..M,1..M]} ; \quad \overline{\Sigma^2}_{r^o}^{-1} = -\left[\frac{\partial^2}{\partial \mathbf{x}^2} \log r_X^o(\mathbf{x}|\mathbf{E}_0)\right]_{\mathbf{x}=\mathbf{x}_0}^{-1} \tag{8}$$

where r_X^o is the derivable part of r_X. This matrix is trimmed to terms $[1..M, 1..M]$ which are the only ones of physical meaning.

3. The Algorithms of the Maximum Likelihood Determination

3.1. The "Quasi-General" Case. — In theory, one has just to solve the equation system (4) or (7) to get the maximum likelihood $\mathbf{x}_0$, and then compute $\overline{\Sigma^2}$ using equations (5) or (8). When $\log q_E(\mathbf{E}_0|\mathbf{x})$, and $\log t_X^o(\mathbf{x})$ are quadratic functions of $\mathbf{x}$, and when $\mathbf{C}(\mathbf{x})$ is a linear function of $\mathbf{x}$, one retrieves the classical *least square method*. However, a more general algorithm applies when they are " quasi-quadratic", i.e. in a domain of $\mathbf{x}$ far from any mathematical singularity so that the *expectation* of $\mathbf{E}$ is *locally* a "quasi-linear" function of $\mathbf{x}$. This gives an iteration process giving at each step a "better" approximation $\tilde{\mathbf{x}}$ of $\mathbf{x}_0$ [8], whose speed of convergence depends on the linearity or quasi-linearity of $\mathbf{Ec}(\mathbf{x}_1)$:

$$\begin{pmatrix} \tilde{\mathbf{x}} \\ \mathbf{\Lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \overline{\Sigma^2}_{r^o}^{-1} & \frac{\partial \mathbf{C}}{\partial \mathbf{x}}^\bullet \\ \frac{\partial \mathbf{C}}{\partial \mathbf{x}} & \overline{\mathbf{0}} \end{pmatrix}_{\mathbf{x}=\mathbf{x}_0}^{-1} \begin{pmatrix} \nabla_X^\bullet \log r^o \\ -\mathbf{C} \end{pmatrix}_{\mathbf{x}=\mathbf{x}_o} \tag{9}$$

3.2. The Particular Case of Weighted Least Squares. — In most physical situations, the deviations of the observed measurements $\mathbf{E}$ with respect to the expectations $\mathbf{Ec}(\mathbf{x})$ are *statistically independent*, i.e. the observed measurement is equal to its *expectation* $\mathbf{Ec}(\mathbf{x})$ *plus* a *non-correlated additive noise* which means that the probability of observable measures only depends on the difference $\mathbf{E} - \mathbf{Ec}(\mathbf{x})$:

$$q_E(\mathbf{e}|\mathbf{x})_{\text{additive noise}} = \aleph(\mathbf{e} - \mathbf{Ec}(\mathbf{x}), \mathbf{x}) \tag{10}$$

where $\aleph$ becomes a *normal law*([6]) if the noise is *Gaussian*:

$$q_E(\mathbf{e}|\mathbf{x})_{\text{additive noise}} = \mathbb{N}(\mathbf{e} - \mathbf{Ec}(\mathbf{x}), \overline{\sigma^2(\mathbf{x})}) \tag{11}$$

([6]) $\mathbb{N}(\mathbf{x}, \overline{\sigma^2}) \equiv \frac{1}{\sqrt{2\pi|\overline{\sigma^2}|}} \exp\left[-\mathbf{x}\overline{\sigma^2}^{-1}\mathbf{x}\right]$.

where $\overline{\sigma^2(\mathbf{x})}$ is the variance matrix of the experimental measures, which can be in turn a function of the unknowns $\mathbf{x}$.

3.2.1. Case of no Previous Knowledge, i.e. $t_X(\mathbf{x})$ "Flat". — Since the matrix $\overline{\sigma^2(\mathbf{x})}$ is either constant of slowly varies with the unknowns, a good approximation for seeking the maximum likelihood consists in seeking the minimum of the argument $\Psi(\mathbf{x})$ of the exponential

$$\Psi(\mathbf{x}) \equiv (\mathbf{E}_0 - \mathbf{Ec}(\mathbf{x}))^\bullet \left[\overline{\sigma^2(\mathbf{x})}^{-1}\right] (\mathbf{E}_0 - \mathbf{Ec}(\mathbf{x})) \tag{12}$$

rather than the maximum of $q_E(\mathbf{e}|\mathbf{x})$. This leads to an iterative algorithm giving at each step a better approximation $\tilde{\mathbf{x}}$ from a previous approximation $\mathbf{x}_1$ of the maximum likelihood location:

$$\begin{aligned} \overline{\mathbf{Q}} &\Leftarrow \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]^\bullet_{\mathbf{x}=\mathbf{x}_1} \overline{\sigma^2(\mathbf{x}_1)}^{-1} \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]_{\mathbf{x}=\mathbf{x}_1} \\ \mathbf{R} &\Leftarrow \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]^\bullet_{\mathbf{x}=\mathbf{x}_1} \overline{\sigma^2(\mathbf{x}_1)}^{-1} (\mathbf{E}_0 - \mathbf{E}_c(\mathbf{x}_1)) \\ \tilde{\mathbf{x}} &\Leftarrow \mathbf{x}_1 + \overline{\mathbf{Q}}^{-1}\mathbf{R} \end{aligned} \tag{13}$$

where $\mathbf{Ec}$ and $\mathbf{E}_0$ are $[1..N]$-vectors (N = number of measures), $\mathbf{x}$ is a $[1..M]$-vector (M = number of unknowns) so that $\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}$ is a $[1..N, 1..M]$-matrix, $\overline{\mathbf{Q}}$ a $[1..M, 1..M]$-matrix and $\mathbf{R}$ a $[1..M]$-vector. After convergence, the error matrix is:

$$\overline{\Sigma^2} = -\left[\frac{\partial^2 \log q_E(\mathbf{E}_0|\mathbf{x})}{\partial \mathbf{x}^2}\right]^{-1}_{\mathbf{x}=\mathbf{x}_0} \approx \left[\left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]^\bullet \overline{\sigma^2(\mathbf{x})}^{-1} \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]\right]^{-1}_{\mathbf{x}=\mathbf{x}_0} = \overline{\mathbf{Q}}^{-1} \tag{14}$$

Remarks:

- *It can be shown [8] that the above algorithms* (13) *and* (14) *are valid for any additive noise, provided the expectation* $\mathbf{Ec}(\mathbf{x})$ *and the variance* $\overline{\sigma^2}$ *of* $\mathbf{E}$ *are known.*

- *These algorithms imply* $\overline{\sigma^2}^{-1}$, *which means one has to invert* $N \times N$*-matrix* $\overline{\sigma^2}$, *which may be enormous. Fortunately, it becomes diagonal in the case of non-correlated noise.. But one should resist to the temptation of smoothing the experimental data.*

3.2.2. Case of the a priori Probability Law $t_X(\mathbf{x})$ not "Flat". — Using previous denotations, (6) for t^o_X and (7) for the constraints, the algorithm — still valid for non-Gaussian noises — becomes:

$$\begin{aligned} \overline{\mathbf{Q}}^0 &\Leftarrow \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]^\bullet_{\mathbf{x}=\mathbf{x}_1} \overline{\sigma^2(\mathbf{x}_1)}^{-1} \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]_{\mathbf{x}=\mathbf{x}_1} + \overline{\Sigma^2}^{-1}_{t^0} \\ \mathbf{R}^0 &\Leftarrow \left[\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}\right]^\bullet_{\mathbf{x}=\mathbf{x}_1} \overline{\sigma^2(\mathbf{x}_1)}^{-1} (\mathbf{E} - \mathbf{Ec}(\mathbf{x}_1)) + \overline{\Sigma^2}^{-1}_{t^0} (\mathbf{x}_t - \mathbf{x}_1) \\ \overline{\mathbf{Q}} &\Leftarrow \begin{pmatrix} \overline{\mathbf{Q}}^0 & \frac{\partial \mathbf{C}^\bullet}{\partial \mathbf{x}} \\ \frac{\partial \mathbf{C}}{\partial \mathbf{x}} & \overline{0} \end{pmatrix}_{\mathbf{x}=\mathbf{x}_1} \\ \mathbf{R} &\Leftarrow \begin{pmatrix} \mathbf{R}^0 \\ -\mathbf{C}(\mathbf{x}_1) \end{pmatrix} \\ \begin{pmatrix} \tilde{\mathbf{x}} \\ \mathbf{\Lambda} \end{pmatrix} &\Leftarrow \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{0} \end{pmatrix} + \overline{\mathbf{Q}}^{-1}\mathbf{R} \end{aligned} \tag{15}$$

3.3. The Practical Difficulties of Computer Treatment of Experiments

3.3.1. Computing the Derivatives of all the Expected Measures against all Unknowns
All the maximum likelihood seeking algorithms exhibit the term $\left[\frac{\partial \mathbf{E}}{\partial \mathbf{x}}\right]_{\mathbf{X}=\mathbf{X}_1}$ whose numerical computation by means of finite differences is infeasible for computer times and accuracy reasons. Therefore, we reverted to *formal derivation.*

3.3.2. Choosing the Minimisation Process of the χ^2. — This is a main issue but, fortunately, iterative gradient methods seem to work in usual physical cases([7]).

3.3.3. Validity Tests. — We mention this point, not because of its difficulty — in fact it is just a side product of the maximum likelihood seeking — but because it is often forgotten by many an experimentalist. In fact the χ^2-test merely consists in computing the variance weighted sum of the squared deviations, which is simply:

$$\Psi(\mathbf{x}_0) \equiv (\mathbf{E}_0 - \mathbf{Ec}(\mathbf{x}_0))^* \left[\overline{\sigma^2}^{-1}(\mathbf{x}_0)\right] (\mathbf{E}_0 - \mathbf{Ec}(\mathbf{x}_0)) \tag{16}$$

that is, nothing but the quantity $\Psi(\mathbf{x})$ of equation (12) one has just minimized to find $\mathbf{X} = \mathbf{x}_0$. Usual χ^2 theory now says that the model is not disagreed — "not disagreed" does not mean "agreed"! — by the experiment if:

$$1 - \frac{3}{\sqrt{N-M}} < \chi^2_r \equiv \frac{\Psi(\mathbf{x}_0)}{N-M} < 1 + \frac{3}{\sqrt{N-M}} \tag{17}$$

4. The Information Needed by EXPHER to Do its Job

The aim of EXPHER is to produce a program whose execution seeks the *maximum likelihood* (algorithms 9, 13 or 15) and computes the *error matrix* $\overline{\Sigma^2}$ (Eq. 14), thus relieving the experimentalist/programmer from many straightforward although tedious tasks which can be automated. To do that, EXPHER needs three types of information:

- the list of the unknowns $\mathbf{X} \in \{\mathbf{x}\}_M$, including the "subsidiary" ones, those needed in the calculation but of no interest for the physicist;
- the expressions of the *expectation* of the *measures* and their *variances* as functions of the unknowns;
- specifications about the file *storage* of experimental data, i.e. *actual measurements* $\mathbf{E}_0$.

All this information is given to the EXPHER package by means of a specification language of the same name, EXPHER. In contradiction to its reading aspect, EXPHER is not an algorithmic language but a *specification language* giving the package the necessary information to write an output program in a traditional algorithmic language, Fortran in a first stage, whose compilation and execution will yield the interesting physical quantities. Thus, using EXPHER implies three steps:

i) translating the language EXPHER into Fortran, done by the EXPHER compiler,

ii) compiling this Fortran program,

iii) executing this Fortran program reading the experimental data.

([7]) But more sophisticated methods are still under consideration and under investigation.

5. EXPHER and its Specification Language

Many syntactic features of EXPHER are shared with many other languages. Since this paper is published in a review dedicated to physics rather than computer science, we shall not describe the whole of the language (more thoroughly described in the EXPHERmanual) and we only insist on those specific to its character of *specification language*.

5.1. THE SOURCE TEXT OF EXPHER. — Like Pascal, C or ADA, instruction are separated with ";" and "·" indicates a *declaration* with the declared object at left and the description on the right.

The symbol "=" is not an *affectation* — an empty concept in a *specification language* — but an identity (like the `PARAMETER` of Fortran or the constant statements of Pascal).

Except in character string constants, EXPHER does not distinguish between upper and lower case letter. But in the following text we chose to denote *reserved keywords* using `UPPER CASE` letters, and other objects (i.e. examples) using `lower case` letters.

As far as semantics are concerned, a source text EXPHER only contains declarations which must be unique for each objet([8]). It is also worth emphasising that, since EXPHER is a *declaration language* and not a *procedural language*, the order of the declarations is of no importance([9]). They may be of two classes:

i) value declarations,

ii) type declarations.

Unless exception (arguments and formal results of procedures — see 5.4.2), the scope of the declarations is the whole of the EXPHER text.

5.2. VALUE DECLARATIONS. — This is the ordinary statement, which says that a given object (referred to by an *identifier*) is *equivalent* to another object or expression; this means: not only equal, but identified with the mathematical relationship symbol "≡". Then the obvious rule is that any object must have been declared one time, not less, not more, and that it must have a *type*. The normal syntax([10]) of a *declaration* is derived from Pascal or ADA, i.e.:

⟨*value-declaration*⟩ : ⟨*identifier*⟩ { : ⟨*type*⟩ }^{0,1} { = ⟨*value*⟩ }^{0,1} ;

where ⟨*type*⟩ is either a declared *type* identifier or a type description, in the same was as is done in Pascal, ADA or even Fortran 90, and which is not worth a long description in this kind of paper. Conversely, the ⟨*value*⟩([11]) notion is more specific to EXPHER, in the sense it may be of three fundamental kinds:

⟨*value*⟩ :. ⟨*expression*⟩ | ⟨*unknown-value*⟩ | ⟨*constrained-value*⟩
⟨*unknown-value*⟩ :. `UNKNOWN.` { , ⟨*initial-value*⟩ }^{0,1}
⟨*constrained-value*⟩ : `CONSTRAINT.` , ⟨*constraint-expression*⟩

The case where an identifier is assigned to be identical to an expression if of straightforward meaning, but the two other cases deserve some comments:

([8]) With possible exceptions due to *scope* rules.

([9]) Obvious for computer scientists, but against tradition in physics programming...

([10]) Where {...}$^{0+}$ means an optional and repeatable item, {...}$^{1+}$ a compulsory repeatable item, {...}$^{1,+}$ a compulsory item, repeatable with commas inbetween, {...}^{0,1} an optional but not repeatable item.

([11]) Optional as the ⟨*type*⟩ in the syntax, but this only means that the *type* and *value* can be given to an object in distinct statements.

5.2.1. The Objects of Unknown Value. — Unknown objects — in the usual sense of an equation to solve — must be of type `REAL` or arrays of `REAL`. They are given the formal value `.UNKNOWN.` which is duly recorded by EXPHER. However, as their determination might be iterative, the formal `.UNKNOWN.` statement may be followed by the indication of an "initial" value which will be used as a starting point in the iterating process; it may also be preceeded by an integer argument which specifies the "group" of unknowns it belongs to, that is, a pragmatic hint to the EXPHER compiler concerning the iteration algorithm to be implemented.

5.2.2. Constrained Values. — Objects which are *constrained* to be equal to another value — which usually depends on the *unknowns* — must also be `REAL` or arrays of `REAL`. They are explicitly declared which the formal value `.CONSTRAINT.`, followed by a comma and the expression which represents the constraint which this object must fulfill. For example

```
y    REAL =  CONSTRAINT., SIN(x)-2*x ;
```

constrains y≡`SIN(x)-2*x` to be zero, where x might be either an unknown by itself, or a value bound to unknowns.

5.3. The Types of EXPHER

5.3.1. Primitive Types. — In addition to `REAL`, EXPHER possesses three other primitive types, `INTEGER`, `LOGICAL` (booleans) and `CHARACTER` (in fact "strings"). However, these three latter types can only represent constants, not unknowns or values depending on unknowns.

5.3.2. The Structures. — A *structured type* is described as a sequence of typed items, each of which has a selector. For example:

```
e . STRUCT(a:INTEGER, b:REAL) ;          % declaration of the type of e
a.OF.e = 12 ; b.OF.f = 5*SQRT(2.225); % assigning values to both fields of e
```

or

```
e   STRUCT(a,b:REAL) ;
e = (12.15, 5*SQRT(2.225)) ;             % assigning a multiple value to e
```

5.3.3. The Arrays. — Except some details concerning syntax, arrays are declared and handled in a classical way, at least for people knowing Algol 68, ADA, C or Fortran 90 [7].

The important point concerning arrays is due to the fact that EXPHER is a specification language, in which each object — i.e. each element of each array, array of arrays, etc. — must be declared *once* and *only once*; then, of course, giving a value to only a part of an array is such a common mistake that EXPHER has to be extremely strict about ensuring a comprehensive definition of all the elements of any array. This is why EXPHER forbids([12]) putting a *sub-array* at the left of an "=" sign. Thus

```
x:[1..3]INTEGER ;
x[1]=12; x[2. 3]=(5,8);
```

is forbidden by EXPHER, but a much more clever, more understandable and-legal way of doing is:

```
x = (12,5,8) ;
```

([12]) Usual physicists, chemists and programmers will complain, just like car drivers complain about compulsory seat belts or prosecution when they do not stop at red lights.

5.4. FUNCTIONS (OR PROCEDURES). — Although the word "procedure" is more common in the computer world, one should rather speak of "functions" which often are a simple way avoiding repetitions, but in some other cases (the FILL primitive) they are necessary to describe some typical definition schemes of EXPHER. Although this would be disappointing for computer scientists, the functions are not recursive in EXPHER, and we think the usual physicist seldom needs this feature.

5.4.1. Procedural Types. — Like Algol 68 [5] but not like ADA [4] or Pascal, *procedures* are considered as values by EXPHER, with some restrictions however. In particular, EXPHER procedures possess a type which completely specifies the types of their *arguments* and of their result and — provided the types agree — values of procedural type can be assigned to be equal to other values of a procedural type defined elsewhere.

5.4.2. Assigning a Procedural Constant to an Object of Procedural Type. — In other words, this is the usual declaration assigning a name to a procedure whose text is given. Normally a simple statement like

```
ft: FUNCTION = FUNCTION(i,j:INTEGER, k:REAL  > REAL): i+j*k;
```

is sufficient to do that, but the body of the procedure may require internal declarations (of restricted scope); then the procedure may also be declared like:

```
tf: FUNCTION = FUNCTION(i   INTEGER  > resu   INTEGER):
BEGIN
  n   INTEGER = INPUT(1,*); % Read one data on file 1
  resu = n+i ;              % Increase it by i to build the result
END ;
```

5.4.3. Vectorising Procedure Calls. — Normally, a procedure call gives a result of the type stated by its declaration; however, a *function* whose argument is stated as a scalar can be given an array, or an array of arrays instead. Then EXPHER considers that the result is in turn an array obtained by invoking the function for each element of the input argument.

Of course, such a feature could be replaced in usual procedural languages with a loop over all element of the given array, but this is no more possible in a declaration language like EXPHER. For example, using the SIN (*sine*) function whose type is FUNCTION(REAL > REAL), the statements

```
x:[2]REAL; y:[2]REAL = SIN(x);
```

assign the array of the sines of the elements of x to the array y, i.e. the equivalent of the following (forbidden) assignments:

```
y:[2]REAL; y[1] = SIN(x[1]); y[2] = SIN(x[2]);
```

5.5. "PRIMITIVE" FUNCTIONS. — As we already say in section 5.4.3, many usual classiques (filling arrays, sums, various products, etc.) are performed in traditional languages — Fortran, C, Pascal — by means of *loops* which require *updating* a *variable*. Such a logic is incompatible with a *declaration language* like EXPHER, unless reverting to *recursivity* which is just in fact another way of presenting the problem. Besides, such loops are hardly compatible with other required operations such as *formal derivations*. This is the reason why EXPHER offers some *primitives* which perform these operations in a clearly controlled manner. We present the most significative of them below.

5.5.1. The FILL *Primitive.* — This extremely useful primitive enables filling an array of any dimension or bounds with values depending on the indices, or on functions of the indices. In short, FILL is called with as many bound arguments as in the resulting array, *plus* one *function argument* which will be invoked with arguments equal to each valid set of indices, and whose result will fill the corresponding element of the resulting array. By the way, we think it strange that no common programming language provides such a feature which would save much tedious programming of the initialisation of arrays. For example:

```
x:[1..2]LOGICAL = FILL(1. 3,1..3,ft);
ft:FUNCTION = FUNCTION(i,j:INTEGER  > LOGICAL): i=(4-j);
```

will build a 3×3 logical array named x whose second diagonal is *true* while all other elements are *false*.

5.5.2. The SUM *and* PRODUCT *Primitives.* — For the same reasons — i.e. avoiding loops — EXPHER has a SUM primitive which computes the sum of the values of an expression depending on a summation index. For example:

```
SUM(i, z-3. y+5, w(i)+x[i+2]+SQRT(x[i]))
```

will compute the *sum* of all the values of `w(i)+x[i+2]+SQRT(x[i])` when the summation index i varies from z-3 to y+5. The primitive PRODUCT has the same syntax, except that it performs the product of the given terms.

5.5.3. Properties of Arrays. — Other obvious primitives yield the lower and upper bounds and the number of dimensions of an array.

5.6. EXPRESSIONS. — An expression may consist of a single object, or it may result from an operation, or it may be a *conditional expression*.

5.6.1. Operations and Operators. — Operator symbols have an obvious meaning; they are: +, -, *, /, **, <*>([13]), =, <>, <, >, <=, >=, .AND., .OR. and .NOT..

5.6.2. Conditional Expressions. — Their syntax looks like Fortran 77 constructs, i.e.:

⟨*conditional-expression*⟩ : IF ⟨*boolean-expression*⟩ THEN ⟨*expression*⟩
{ ELSEIF ⟨*boolean-expression*⟩
THEN ⟨*expression*⟩ }$^{0+}$
ELSE ⟨*expression*⟩
ENDIF

but it is to be emphasized that, although the keywords look similar, this kind of construction generating a *conditional value* rather than *conditionally performing an instruction* appears in very few languages (Algol 68, TEX [6], C). Although useful, it is not necessary in procedural languages, but compulsory in a declaration language like EXPHER. Example:

```
x = 3+ IF n=1 THEN 2 ELSE n+1 ENDIF * 5 ;
```

5.7. VECTORIAL FACILITIES. — In order to facilitate operations on arrays, EXPHER assumes that, if an operator ⊙ can operate between all *fields* of two structured values a and b of the same type, then a ⊙ b exists, and it consists in letting ⊙ operate between all the fields of both structured values. The same principle holds between two arrays of the same type.

In the same way, a structure or an array can be multiplied or divided by a scalar, if all fields of the structure or all elements of the array can.

([13]) For matrix products.

5.8. REFERENCING FILES. — To produce a relevant executable program, EXPHER obviously needs to know the names of the files to read at execution time and their data structure.

5.8.1. Declaring a Data File. — This uses a predefined type named `FILE`; for example:

```
xdat   FILE = ('mydata.dat',2) ;
```

assigns the external file `mydata.dat` to the Fortran input unit number 2 and calls it `xdat` for references inside the EXPHER source text.

5.8.2. Reading Data. — This is performed by the `INPUT` primitive whose syntax refers to a *logical unit* and to one or several Fortran *format specifications* (see examples in sections 7.4.1 and 5.4.2):

`INPUT ( ⟨logical-unit⟩ , ⟨formats⟩ )`

5.8.3. The Case of Array of Undefined Bounds. — EXPHER allows input arrays and procedure arguments to have bounds specified with a question mark ("?") rather than an explicit expression. In the case of a procedure argument, this obviously means that the missing bound will be taken from the actual given argument but, in the case of an input array, this means that the missing bound(s) will be read in the input file before the contents of the array. This clearly enables arrays whose bounds depend on the data given at execution time.

6. How EXPHER Performs its Job

Although it may look like, EXPHER is not a compiler. Indeed compilers actually record a minority of declarations (values, variables, types) but their main job consists in reading and translating a *source text* step by step. Conversely, EXPHER first stores all the information pertaining to each object that is, not only its type but the whole tree of the operations defining its value. Only after that does it analyse their logical structure of the objects and of the unknowns to build a *maximum likelihood* seeking algorithm and the computation of the *error matrices*.

6.1. THE VARIOUS STEPS OF THE TREATMENT BY EXPHER

6.1.1. Reading and Compiling the Source Files. — EXPHER reads two source files, namely the description of the predefined objects (functions, types and constants) and the user source file where the actual experiment is described in the EXPHER language. Then each object — explicitly declared or not — is stored in a temporary data base with a number of qualifying items: *name* (a character string), *class* (i.e. *operator*, *type* or *sub-type*, *value*), *type* (*pointer* to the description of its type), *kind* (function, structure or array), *bounds* (for arrays) or *arguments* (for procedures) or *fields* (for structures), *description* (a pointer to the description of that object, namely an operator with its operands or the components of a type).

In addition, some boolean flags tell whether it is an *unknown* or a *constraint*, etc.

6.1.2. Creating the Objects Involved in the Data Treatment. — EXPHER analyses the various fields of the created objects in order to establish an appropriate algorithm to seek the *maximum likelihood*; then it creates the fundamental objects involved in the algorithm, such as $\mathbf{Ec}(\mathbf{x})$, $\mathbf{x}$, $\overline{\mathbf{Q}}$, $\mathbf{R}$, etc.

6.1.3. Treating the Objects of Stated Value. — EXPHER examines their value and, if possible, it decomposes this value into elementary expressions. Then it looks whether this elementary expression exists; if it does, a pointer to this existing expression is made, otherwise a new object is created. Thus, the description of any object is build as a tree of elementary expressions.

6.1.4. Formal Derivation. — Computing the error matrix $\overline{\Sigma^2}$ (Equ. (14)) implicitly requires formally calculating the derivatives $\frac{\partial \mathbf{Ec}}{\partial \mathbf{x}}$. To do that, EXPHER has to internally perform formal derivations. This is not very difficult since the values of the object have been represented as a *tree of operations* whose formal derivation is easy, although manually tedious.

6.2. Writing the User's Program Fit for Experimental Data Treatment

6.2.1. Data Storage. — EXPHER generates static storage only for static data, i.e. those whose size is known at EXPHER compile time. The others — mainly arrays — are dynamically stored, with types separation (integer, real, character, boolean) for compatibility reasons due to the final programming language([14]).

7. Using EXPHER

7.1. Availability. — EXPHER is a *freeware* product, of course not to be modified without the authors' agreement. The current version is available through anonymous `ftp` at `hprib.lps.u-psud.fr` (a HP9000/UNIX), in the subdirectory "`pub/expher`" which contains the EXPHER manual in LaTeX and in PostScript. Second level directories provide the various versions of EXPHER depending on the target system (VAX/VMS, RS6000/AIX(UNIX), HP/HP_UX(UNIX), SUN/SunOS(UNIX) and the PC/MS-DOS version which uses the `f2c` [11] converter and the `gcc/djgpp` C compiler by D.J. Delorie). In addition, another sub-directory named `master` provides the "master" files and programs needed to generate all these versions, and possibly some others.

7.2. Installing EXPHER . — Just compile the relevant source files and link them using the provided macros and put the executable in a convenient directory. Then EXPHER has just to be invoked with the name of the description of the experiment in the EXPHER language as a command parameter.

7.3. Compiling with EXPHER. — Unless errors occur, the use of EXPHER produces an output Fortran program which has just to be compiled using the Fortran 77 compiler. It is to be noticed that EXPHER running on a given computer can yield a program fit for another system: this is just a specific option to give it.

Then, output Fortran program has just to be executed, provided it has access to the data files specified in the EXPHER source text.

7.4. Some Experience with EXPHER

7.4.1. An Example in Solid State Physics: Dielectric Constant Measurement. — Both the real and the imaginary parts of the dielectric constant of $(TMTSF)_2(AsF_6)_{0.93}(SbF_6)_{0.07}$ have been measured, at a temperature of 2 K and as a function of the frequency (in fact the pulsation ω) in the range [1000—20000 Hz] with a step of 1000 Hz [9]. These sets of measurements $\{\varepsilon_r\}$ and $\{\varepsilon_i\}$ are recorded in files `eps_r.dat` and `eps_i.dat`, respectively, and the uncertainty of these measures is estimated to 2%, i.e. a constant relative error.

([14]) Presently Fortran 77, but Pascal or C would pose similar problem

In this domain of low frequencies, the dielectric constant is empirically assumed of the form :

$$\varepsilon(\omega) = \varepsilon_r(\omega) + j\,\varepsilon_i(\omega) = \frac{\varepsilon(0)}{1+(\omega\tau_0)^{2\alpha}} + j\,\frac{\varepsilon(0)(\omega\tau_0)^{\alpha}}{1+(\omega\tau_0)^{2\alpha}} \tag{18}$$

where $0 \leq \alpha \leq 1$ is a coefficient describing the width of the distribution of the relaxation time constant τ around its central value τ_0.

Starting from these experimental data, one wants to determine three unknowns, namely $\varepsilon(0)$, τ_0 et α. Since $\varepsilon(\omega)$ is not a linear function of the unknowns τ_0 and α, starting values to initiate the necessary iteration are needed, and we chose $\varepsilon(0)_{\text{start}} = 1$, $\tau_{0\text{start}} = 0.0001$ and $\alpha_{\text{start}} = 0.7$.

The source file describing the experiment in EXPHER language is then:

```
% Basic utility declarations
  freq_deb   real = 1000.0 ;  % declaring the starting frequency
  freq_fin   real = 20000.0 ; % declaring the ending frequency
  ecart . real = 1000.0      ; % the frequency step
  n   integer = nint((freq_fin-freq_deb)/ecart)+1; % size of experimental data
  Spectre   TYPE = [n]real ;  % a "type" declaration

  eps0   real =  unknown., 1.0 ;  % ε(0) is an unknown of initial value 1.0
  tau    real =  unknown., 0.0001 ;% τ0 is an unknown of initial value 0.0001
  alpha   real =  unknown., 0.7 ; % α is an unknown of initial value 0.7

% Declare what are the data, what was expected and the expected deviations
  tok    [2][n]experim ;      % we call tok the [2,n]-array of the experimental
                              % measurements; experim is a predefined structured
                              % type consisting of a measure, an expectation
                              % and an experimental deviation (expected deviation).
  measure.of.tok = fill(2,fillm) ; % (introduce the measured data; see below 5.5.1)
  expect.of.tok = fill(2,experience) ; % (say what is expected; see below 5.5.1)
  error.of.tok = 0.02*absr(measure.of.tok) ; % uncertainty of 2% in magnitude

% Define the procedures used before
  fillm   function = function(nexp:integer  > Spectre):
    if nexp=1 then Spectre(input(f1,*)) % One part of data is in a file,
    else Spectre(input(f2,*))           % the other in another file,
    endif ;                             % and both coerced to type Spectre.

  experience   function = function(nexp:integer  > Spectre) .
    fill(n,expectation,nexp);           % (expectation has an additional
                                        % argument,nexp).

  expectation   function = function(i,nexp:integer  > resu:real):
    begin
      x   real = freq_deb+(i-1)*ecart ;
      resu = if nexp=1 then er(x) else ei(x) endif ;
    end;

  er   function = function(x:real .> real)
       eps0/(1.0+(2*pi*x*tau)**(2*alpha));

  ei   function = function(x:real .> real) .
       eps0*(2*pi*x*tau)**alpha/(1.0+(2*pi*x*tau)**(2*alpha));
```

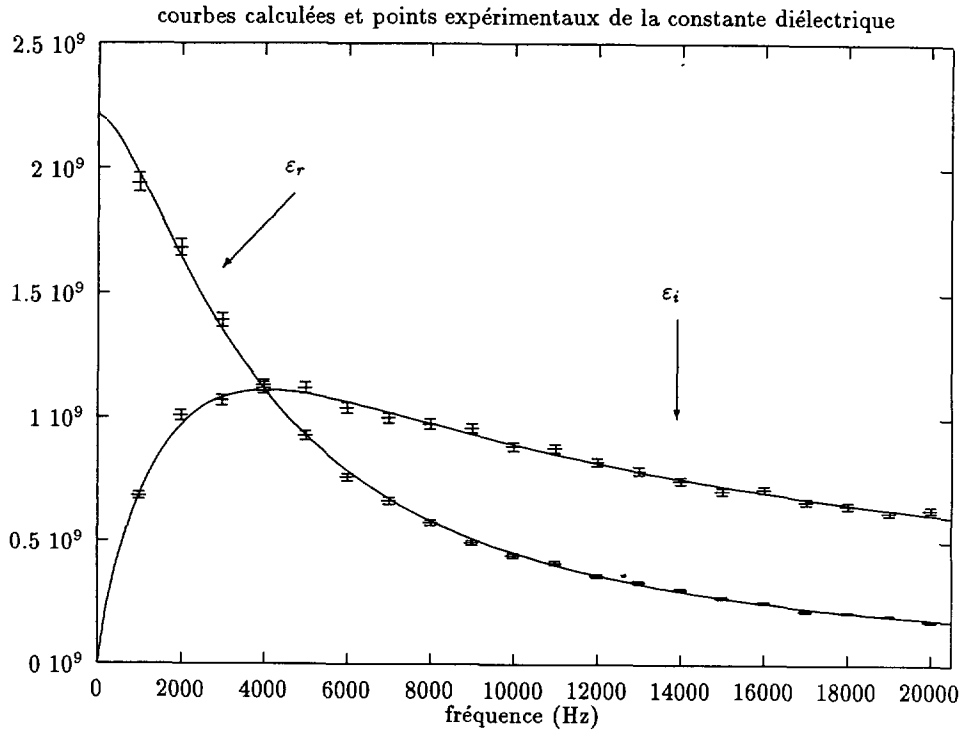

Fig. 1. — Computed and recorded values of ε_r and ε_i

```
% Declaring the input data files
  f1   file = ('epsr.dat',1) ; % contains experimental data {εr}
  f2   file = ('epsi.dat',2) ; % contains experimental data {εi}

% (end of EXPHER source file)
```

After reading that, EXPHER generates a rather serious program of hundreds of Fortran statements, which in turn refers to a specific library. After 15 iterations (5 seconds of CPU on a HP9000/755) the execution of the program produced by EXPHER gives the following results:

$\varepsilon(0) = (2.222 \pm 0.011) * 10^9$
$\tau_0 = (3.95 \pm 0.04) * 10^{-5}$ s
$\alpha = 0.752 \pm 0.005$

and the reduced Chi-2 value is $\chi_r^2 = 1.155$ which is pretty good. The comparison of computed (expected) data with experimental ones is given in Figure 1.

7.4.2. A Geological Application. — While one of the authors (P.W.) worked in tuning the software itself, the other (D.T.) tried it on examples provided by Jaoul and Raterron [10]. Writing in EXPHER language the initial code took half an hour. This consisted in fitting 38 data values with the formula :

$$\dot{\epsilon}_k = A\sigma_k^n \exp \frac{-E}{RT_k} \tag{19}$$

where A, n and E were three couples of unknowns (depending on the T temperature range) and the values with subscript k experimental data. The EXPHER source code was:

```
npts : integer = input(9,*);
coupleex : type = struct(tk, sigma, epsil, erreps: real);
exp_data : [npts]coupleex =   fill(1..npts, read_exp);

read_exp : function=function(i: integer:> in_val:coupleex):
 input(9, *, *, *, *);

epsilon : [npts]experim;

measure  of. epsilon = ln(epsil .of. exp_data);

error .of. epsilon = 0.01 * (erreps .of. exp_data);

expect  of. epsilon =  fill(1. npts, calc_expect);

calc_expect   function = function(kk:integer :> real):
 if  tk .of. exp_data[kk] < 1413
 then
    log_grand_A[1] + petit_n[1]*ln(sigma .of. exp_data[kk])
     -grand_E[1]/(R * tk .of. exp_data[kk])
 else
    log_grand_A[2] + petit_n[2]*ln(sigma .of. exp_data[kk])
     -grand_E[2]/(R * tk .of. exp_data[kk])
 endif;
log_grand_A : [2]real = 1, .unknown. ; % these unknowns in same group #1
grand_E : [2]real = 1,  unknown. ;     % these unknowns in the same group
petit_n : [2]real = 1,  unknown. ;     % these unknowns in the same group

xxx: file = ('jaoul.dat',9);
r: real = 8.32;
```

Unfortunately, at the first trial the resulting (reduced) χ^2_r was awful, namely more than 400. This was due to the fact that the various σ_k were subjected to experimental errors, of the order to 10%, i.e. one should consider two classes of values of σ, the σ_k which were measured (or posted) and the actual values of the same parameters, namely σ_{unk_k}.

If we had directly programmed the "regression", this change would have needed several days of work, but using EXPHER, it was done in half an hour, giving:

```
npts : integer = input(9,*);
nsigma : integer = input(9,*);

coupleex : type = struct(tk: real, i_sigma: integer, epsil, erreps: real);
exp_data : [npts]coupleex =   fill(1..npts, read_exp);

read_exp : function=function(i: integer:> coupleex):
 input(9, *, *, *, *);

couplesig : type = struct(i_sigma: integer, sigma: real);
exp_sigma : [nsigma]couplesig = fill(1. nsigma, read_sig);
```

```
read_sig : function=function(i: integer:> couplesig):
 input (9, *, *);

epsilon : [npts]experim;
sigma_eff   [nsigma]experim;

measure .of. epsilon = ln(epsil .of. exp_data);
measure .of. sigma_eff = sigma .of. exp_sigma;

error .of. epsilon = 0.01 * (erreps .of. exp_data);
error .of. sigma_eff = 0.1 * (sigma .of. exp_sigma);

expect  of. epsilon =  fill(1. npts, calc_expect);
expect  of. sigma_eff =  sigma_unk;

calc_expect   function = function(kk:integer :> real):
 if  tk .of. exp_data[kk] < 1413
 then
    log_grand_A[1] + petit_n[1]*ln(sigma_unk[i_sigma .of. exp_data[kk]])
     -grand_E[1]/(R * tk .of. exp_data[kk])
 else
    log_grand_A[2] + petit_n[2]*ln(sigma_unk[i_sigma .of. exp_data[kk]])
     -grand_E[2]/(R * tk .of. exp_data[kk])
 endif;

log_grand_A : [2]real = 1, .unknown. ;
grand_E : [2]real = 1,  unknown. ;
petit_n : [2]real = 1, .unknown. ;
sigma_unk : [nsigma]real = 2,  unknown., sigma .of. exp_sigma ;

xxx: file = ('jaoulxxz.dat',9);
r: real = 8.32;
```

With this new description, the resulting reduced χ_r^2 was of the order of 60. In fact, this is of little importance since we are not presently dealing with geophysics but, afterwards, several other tests were made, each of them requiring about ten minutes of coding, plus some dozens of minutes getting valuable output data.

References

[1] Wolfram S., Mathematica, second edition (Addison-Wesley, 1991) pp. 172-173.
[2] Turner P.J., ACE/gr, User's Manual, Software Documentation Series (1993) pp. 47-49.
[3] Kaleidagraph learning Guide, version 2.0, Synergy Software.
[4] Le Verrand D., Le langage ADA, manuel d'évaluation (Bordas, 1982).
[5] Buffet J., Arnal P. and Quéré A., Définition du langage algorithmique Algol 68 (Hermann, 1972).
[6] Knuth D.E., The TEXbook, (Addison-Wesley, 1984).
[7] Metcalf M.and Reid J., Fortran 90 Explained (Oxford University Press, 1990/1992).
[8] Taupin D., Probabilities, data reduction and error analysis in the Physical Sciences (Les éditions de physique, 1988) pp. 57-58 and 63-64.

[9] Traetteberg O., Thèse (Orsay, 1993).

[10] Raterron P. and Jaoul O., *J. Geoph. Res.* **96** (1991) 14.277-14.286.

[11] Feldman S.I., Gay D.M., Maimone M.W. and Schryer N.L., "A Fortran-to-C Converter", Computing Science Tech. Rep. **149** (AT&T Bell Labs, 1993).