Computational Thermodynamics Library: **TDLIB'00**

E.B. Rudnyi
Chemistry Department, Moscow State University, 119899, Moscow, Russia,
rudnyi@comp.chem.msu.su
http://www.chem.msu.su/~rudnyi/tdlib/

1. Introduction 2

1.1. Legal: GNU Public license 2

1.2. Final goal 2

2. Library Design 4

2.1. Background for Phase Models Library (PHASE) 4

2.2. Background for Thermodynamic Algorithms Library (TD_ALGO) 7

2.3. Background for Inverse Problem Library (VARCOMP) 8

2.4. Some Thoughts on Possible Implementations (a single muster-function, interpreting, object-oriented approach) 10

2.5. External Representation of the library objects: XML (SGML) 15

3. Thermodynamicist Guide 18

3.1. Getting started 20
3.1.1. Simple data fitting – a linear regression (tdlib/ex/line - Ref. [8]) 20
3.1.2. The Ba-Cu binary system (tdlib/ex/bacu - Ref. [9]) 22

3.2. Describing phases. I. Simple objects in the PHASE library (tdlib/ex/phase) 23
3.2.1. Object coef 24
3.2.2. Objects elem and formula 25
3.2.3. Object PointPhase (tdlib/ex/phase/pp) 25
3.2.4. Objects in category func_Tp (tdlib/ex/phase/pp) 26
3.2.5. Object species (tdlib/ex/phase/pp/) 28
3.2.6. Object SimpleSolution (tdlib/ex/phase/simple/) 30
3.2.7. Objects in category FuncTpx (tdlib/ex/phase/simple) 30
3.2.8. Object CuOx_ordered_plane (tdlib/ex/phase/y247/) 34
3.2.9. Object NumericalDerivatives (tdlib/ex/phase/y247/) 34

3.3. Describing phases. II. Solution models with the internal parameters in the PHASE library (tdlib/ex/phase) 35
3.3.1. Object CuOx_plane (tdlib/ex/phase/y123) 37
3.3.2. Objects AssociatedSolution and associated_solution (tdlib/ex/phase/ass) 38

3.4. Describing phase equilibria. Objects in the TD_ALGO library 42
3.4.1. Object PassThrough (tdlib/ex/algo) 43
3.4.2. Object OutputFile 43
3.4.3. Object convert 44
3.4.4. Object ComputeOutput (tdlib/ex/algo) 44
3.4.5. Object compute 44
3.4.6. Object PhaseProperty (tdlib/ex/phase) 45
3.4.7. Object PhaseEquilibrium (tdlib/ex/bise) 46
3.4.8. Object reaction (tdlib/ex/bise) 52

3.5. Describing residuals. Objects and formats in the VARCOMP library 52
3.5.1. Object residual (tdlib/ex/bacu and tdlib/ex/line) 52
3.5.2. Objects in the category output 54

3.5.3. Describing experimental values (tdlib/ex/bacu/ and tdlib/ex/line/) 54
3.5.4. Describing variance components (tdlib/ex/bacu/ and tdlib/ex/line/) 55
3.5.5. Describing unknowns (tdlib/ex/bacu/ and tdlib/ex/line/) 56
3.5.6. Describing optimizers 57

4. Case studies 59

4.1. Fictitious A-B binary system from the ThermoCalc Parrot guide (tdlib/ex/tc_ab) 59

5. Information for programmers 66

6. Question, discussion, bugs, etc 67

7. Acknowledgment 69

8. Literature 69

1. Introduction

Modern thermodynamics heavily depends on computing. Most practical tasks do not have a solution in the closed form, and thus, the use of thermodynamics is tightly bound with numerical optimization methods. The term “computational thermodynamics”, coined as far as I know by Bo Sundman, reflects this rather well. There are several integrated commercial packages available as well as public domain and shareware software (for example, see lists [1, 2]). Still in my view, the situation is quite far from the satisfactory one.

There are many unsolved fundamental problems that lead to phenomena in thermodynamics circles referred to as "manual optimization" (for example, see [3]). Practically speaking, this looks like as follows. A researcher set up a problem, runs the software and receives that "convergence has not been achieved", "task has not been solved" or just "division by zero". The main reason is usually tied with the unsatisfactory initial guess that has been chosen automatically by the software. Then, a researcher has to meditate for a while and to find out the right initial guess by himself.

Another problem with the most current software is that it is available in the binary form only. This means that with this software a researcher can solve mostly routine tasks and he/she can change nothing within the software. If you are going to work at frontiers of science, this is unacceptable, because in this case you cannot take part in the development of new thermodynamics ideas.

1.1. Legal: GNU Public license

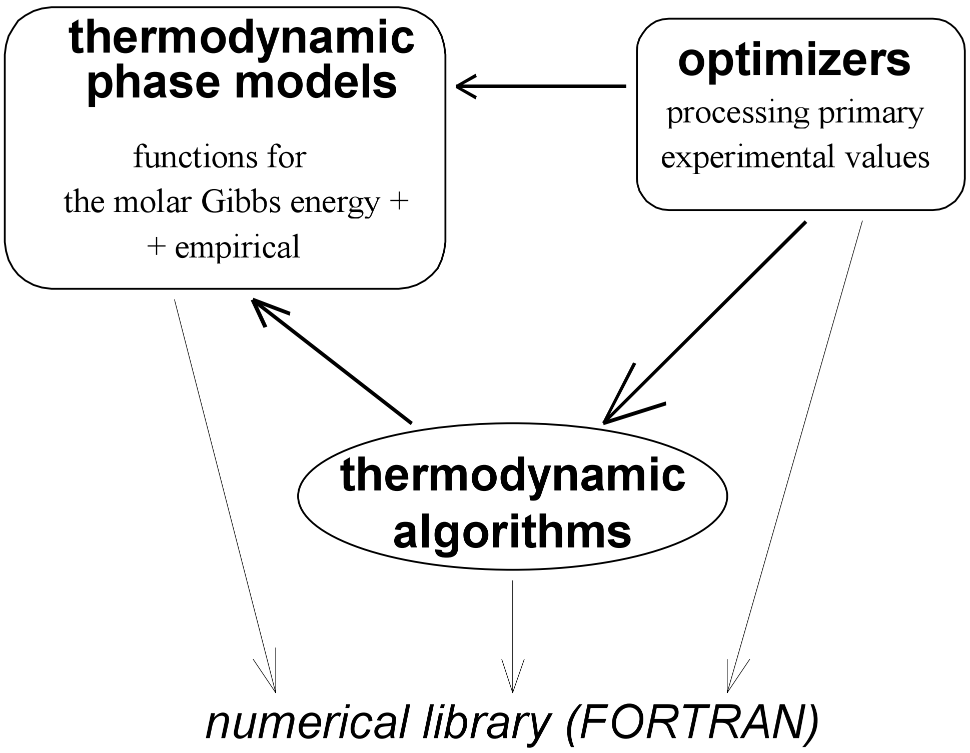
Free Software Foundation (FSF) advocates freedom for the distribution of software (see GNU manifesto [4]). In my view, this is exactly what the thermodynamics community needs. The main requirement is that the software must be distributed with the source code and an individual must have full rights to modify the source code in accordance with his/her necessities. In order to achieve this goal, the GNU public license [5] has been developed by FSF and there is a lot of software already available under these terms [6]. Note that free means freedom, not price.

1.2. Final goal

Fig. 1 presents an overview of the computational thermodynamics library. It should be mentioned, that this idea is not new. The most commercial packages already have similar structure.

A researcher starts with a thermodynamic model - he/she suggests molar Gibbs energies for all the phases in the system in question. For many phases the Gibbs energy can be found in the reference books or thermodynamics databases (see [1, 2]). If this is not the case, the researcher should guess the functional form of the Gibbs energy and to leave there some unknown empirical parameters.

When the thermodynamic model is ready, then it is possible to compute equilibrium properties of the system. This is achieved by means of thermodynamics algorithms and will be referred to as the direct problem. However, when the empirical parameters within the Gibbs energy are unknown, then first it is necessary to solve the inverse problem (optimizers in Fig. 1), that is, to process available experimental values to estimate unknown parameters within the Gibbs energies.


thermodynamic phase models
functions for the molar Gibbs energy + + empirical
optimizers
processing primary experimental values
thermodynamic algorithms
numerical library (FORTRAN)
Fig. 1. A sketch for the computational thermodynamics library.

Solid arrows show dependencies between different pieces of the library. Thermodynamics algorithms depend on phase models while optimizers rely on both phase models and thermodynamic algorithms. After all, the inverse problem is consecutive applications of the direct problem while unknown empirical parameters are changed.

All three parts of the library depends on numerical methods. Implementations for most numerical approaches can be typically found as Fortran subroutines. Thus, the whole library can be considered as a thermodynamic shell to numerical routines. The problem is not to develop new numerical methods but rather to employ the latest achievements in numerical science.

Below is a list of criteria that, in my view, should be considered as must for the library.

- A user must be able to add new user-defined phase models and new thermodynamic algorithms to the library.
- When the user is programming the new phase model, nothing (not a line) must be changed in the thermodynamic and optimization routines.
- Optimization and thermodynamic algorithms must be generic and not depend on a particular phase model.

Currently none of the available thermodynamic software satisfies with these criteria. The current paper presents the library that approaches these goals.

2. Library Design

The library, as shown in Fig. 1, comprises three parts. In the implementation presented they are as follows: PHASE – phase models, TD_ALGO – thermodynamic algorithms, VARCOMP – the inverse problem. Let us start with a look what mathematical tasks are solved by each part of the library.

2.1. Background for Phase Models Library (PHASE)

The thermodynamic system consists from phases. In any case, at least a single phase must be considered. A system that does not contain phases is outside of the thermodynamic scope. In the present work, the terms *phase* and *solution* will be employed interchangeably because in the general case the phase is a N-component solution. Unfortunately, the term *solution* has two unrelated meanings, chemical and mathematical, and in the current paper I need both of them. Hope that this will not bring much confusion.

A phase (solution) comprises several chemical substances (often they are referred to as species). Each of them has a molecular formula, that can be expressed as a formula vector, $\mathbf{a}_i$ (a set of subscripts to the elements in the molecular formula). A set of all the formula vectors leads to the formula matrix, $\mathbf{A} = \{\mathbf{a}_1, \ldots , \mathbf{a}_N\}$ (see, for example [7]), where N is the number of species. The important characteristic of the phase is the number of components in the solution, which is determined as a rank of the formula matrix

$$C = \mathrm{rank}(\mathbf{A}) \tag{2.1}$$

If $C < N$ then the solution is usually referred to as an association solution, and a set of components can be obtained as a subset of linearly independent species.

The task of the first part of the library, PHASE, is to create a framework for modeling the molar Gibbs energy of a phase as a function of external variables: temperature, pressure and mole fractions of the components

$$\mathrm{G_m}(T, p, x_1, \ldots, x_C) \tag{2.2}$$

Note that by definition the sum of the mole fractions is always equal to one, $\Sigma_i x_i = 1$, and thus, the number of independent variables here is equal to $C + 1$, where C is the number of phase components. If we speak of an associated solution, than the mole fractions in Eq. (2.2) mean the gross-composition. They should be computed as if the solution contains only the independent components without any other species.

Quite often, for example in an associated solution, in the original Gibbs energy there are more variables than the required number of independent external variables. We can express this in the general form as follows

$$\mathrm{G_m}(T, p, x_1, \ldots, x_C; y_1, \ldots, y_N) \tag{2.3}$$

where y_i is the extra-variable. In the example of an associated solution, y_i would mean the "true" mole fraction of the i-th species. The extra-variables y_i are sometimes referred to as internal variables. Conventionally, all the internal variables are determined by minimizing the Gibbs energy over unknown internal variables, $y_1, \ldots, y_N$, at constant external variables, $T, p, x_1, \ldots, x_C$. Let us denote the values found in this procedure by the subscript *eq*. Then the values obtained are substituted into the original Gibbs energy and, as a result, we still have the Gibbs energy as function of the external variables only

$$\mathrm{G_m}\{T, p, x_1, \ldots, x_C; y_{1,eq}(T, p, x_1, \ldots, x_C), \ldots, y_{N,eq}(T, p, x_1, \ldots, x_C)\} \tag{2.4}$$

Note that as shown in Eq (2.4), $y_{i,eq}$, by itself is a function of T, p, $x_1, \ldots, x_N$. The major practical consequence of introducing internal variables is that even if the original Gibbs energy is given in the closed form, the final Gibbs energy can be usually obtained by numerical methods only (it is highly unlikely that the results of the minimization might be found in the closed form). The principal difference between Eq. (2.3) and Eq. (2.4) is that while the latter concerns only the equilibrium states of the solution the former describes the non-equilibrium states as well. The TDLIB user is supposed to be interested in Eq. (2.4) only because the number of applications where it is necessary to deal with Eq. (2.3) directly is very low.

The Gibbs energy is usually partitioned as follows

$$\mathrm{G_m} = \mathrm{G_{ref}} + \mathrm{G_{mix}} \tag{2.5}$$

where the first term relates to the Gibbs energies of pure components before mixing

$$\mathrm{G_{ref}} = \Sigma_i\, x_i\, \mathrm{G}^*_{\mathrm{m},i}(T, p) \tag{2.6}$$

where a superscript * reminds us that this quantity is for a pure component. The second term is called as the Gibbs energy of mixing and corresponds to the process of mixing of pure components at constant temperature and pressure. In turn, the Gibbs energy of mixing is sometimes partitioned to the ideal Gibbs energy and the excess Gibbs energy

$$\mathrm{G_{mix}} = \mathrm{G_{ideal}} + \mathrm{G_{excess}} \tag{2.7}$$

For the simplest models without internal parameters, the ideal Gibbs energy is estimated as follows

$$\mathrm{G_{ideal}} = \Sigma_i\, x_i\, \mathrm{R}T \ln x_i \tag{2.8}$$

The excess Gibbs energy is usually modeled as some sum of products of a function in temperature and pressure and a function in mole fractions. Worthy of noting that both ideal and excess Gibbs energies usually should be equal to zero when a composition vector corresponds to the pure component.

The molar Gibbs energy of the phase as a function of external parameters (temperature, pressure, and mole fractions, Eqs 2.2 and 2.4) can be considered as a master function, because all other thermodynamic properties of the phase can be inferred from its molar Gibbs energy by means of differentiating. The integral thermodynamic properties are obtained as follows.

$$\begin{aligned}
\mathrm{S_m} &= -(\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x}} \\
\mathrm{H_m} &= -T^2\{(\partial \mathrm{G_m}/T)/\partial T)_{p,\mathbf{x}} \\
\mathrm{C}_{p,\mathrm{m}} &= -T(\partial^2 \mathrm{G_m}/\partial T^2)_{p,\mathbf{x}} \\
\mathrm{V_m} &= (\partial \mathrm{G_m}/\partial p)_{T,\mathbf{x}} \\
\alpha_V &= (\partial^2 \mathrm{G_m}/\partial p \partial T)_{\mathbf{x}}/(\partial \mathrm{G_m}/\partial p)_{T,\mathbf{x}} \\
\kappa_T &= -(\partial^2 \mathrm{G_m}/\partial p^2)_{T,\mathbf{x}}/(\partial \mathrm{G_m}/\partial p)_{T,\mathbf{x}}
\end{aligned} \tag{2.9}$$

In these equations, one must employ the Gibbs energy in the form of Eq. (2.2) or (2.4). If for some phase model with internal parameters we do not have Eq. (2.4) in the closed form, it is always possible to take the derivatives numerically. However even in this case, we can find at least the first derivatives of the Gibbs energy analytically if we employ the mathematical rules for the derivatives of the compound function.

We have

$$\mathrm{G_m}(T, p, \boldsymbol{x}) = \mathrm{G_m}\{T, p, \boldsymbol{x}, \boldsymbol{y}_{eq}(T, p, \boldsymbol{x})\} \tag{2.10}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are the vectors of mole fraction of components and internal variables accordingly. Then

$$(\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x}} = (\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x},\mathbf{y}} + \Sigma_i\, (\partial \mathrm{G_m}/\partial y_i)_{T,p,\mathbf{x},y(\mathrm{j}\neq\mathrm{i})}\, (\partial y_i/\partial T)_{p,\mathbf{x}} \tag{2.11}$$

Similar equations can be written for other first derivatives of the Gibbs energy. If the internal parameters are found, as was mentioned above, by minimizing the Eq. (2.3) over internal variables, $y_1, \ldots, y_N$, then Eq. (2.11) can be simplified because in this case the second term from the right side happens to be zero, and we finally have

$$(\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x}} = (\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x},\mathbf{y}} \tag{2.12}$$

In some cases we need to apply Eq. (2.11) directly. To this end, it is necessary to compute the derivatives $(\partial y_i/\partial T)_{p,\mathbf{x}}$. Some problem here is that, as a rule, internal variables are not known as functions of T, p, $\boldsymbol{x}$ in the closed form, but rather they are computed as a solution of a system of N non-linear equations with N unknowns.

$$\begin{gathered} f_1(y_1, \ldots, y_N; T, p, x_1, \ldots, x_C) = 0 \\ \ldots \\ f_N(y_1, \ldots, y_N; T, p, x_1, \ldots, x_C) = 0 \end{gathered} \tag{2.13}$$

The notation above emphasizes that the values of internal variables that should be found by solving Eq. (2.13) depends on temperature, pressure, and component mole fractions. In this case, the rule for implicit derivatives allows us to find $(\partial y_i/\partial T)_{p,\mathbf{x}}$ required. What is necessary is to solve a system of N linear equations as follows

$$\begin{gathered} \Sigma_i\, (\partial f_1/\partial y_i)_{T,p,\mathbf{x},y(\mathrm{j}\neq\mathrm{i})}\, (\partial y_i/\partial T)_{p,\mathbf{x}} = -\, (\partial f_1/\partial T)_{p,\mathbf{x},y} \\ \ldots \\ \Sigma_i\, (\partial f_N/\partial y_i)_{T,p,\mathbf{x},y(\mathrm{j}\neq\mathrm{i})}\, (\partial y_i/\partial T)_{p,\mathbf{x}} = -\, (\partial f_N/\partial T)_{p,\mathbf{x},y} \end{gathered} \tag{2.14}$$

The system of equation (2.14) can be easily modified in order to find $(\partial y_i/\partial p)_{T,\mathbf{x}}$ and $(\partial y_i/\partial x_k)_{T,p,x(\mathrm{l}\neq\mathrm{k})}$ if necessary.

Partial molar properties can be also computed from the molar Gibbs energy. Let us take the chemical potential as an example. By definition,

$$\mu_\mathrm{i} = (\partial \mathrm{G}/\partial n_i)_{T,p,n(\mathrm{j}\neq\mathrm{i})} \tag{2.15}$$

where $\mathrm{G} = n\, \mathrm{G_m}$, n is total number of moles. After some work with derivatives, one can obtain

$$\mu_\mathrm{i} = \mathrm{G_m} - \Sigma_k\, x_k\, (\partial \mathrm{G_m}/\partial x_k)_{T,p,x(\mathrm{j}\neq\mathrm{k})} + (\partial \mathrm{G_m}/\partial x_i)_{T,p,x(\mathrm{j}\neq\mathrm{i})} \tag{2.16}$$

Similar formulas can be obtained for other partial properties. Note that after partitioning the Gibbs energy, the partial properties will be also partitioned. In the case of the phase model with internal parameters, the approach expressed above by Eqs (2.10) to (2.14) can be used to find $(\partial \mathrm{G_m}/\partial x_i)_{T,p,x(\mathrm{j}\neq\mathrm{i})}$.

Pure or stoichiometric substances are considered to be a special, degenerated case of a solution model, because formally they can be called as one-component solutions ($N = 1$). In thermodynamic slang they are referred to as point phases. Here, the molar Gibbs energy depends on temperature and pressure only, $\mathrm{G}^*_{\mathrm{m},i}(T, p)$ (the mole fraction is always equal to one). As a result, all the partial properties are equal to the integral molar properties, for example

$$\mu^* = \mathrm{G}^*_\mathrm{m} \tag{2.17}$$

For pure substances, the Gibbs energy of mixing as defined above must be equal to zero. However, there is some sense still to partition the Gibbs energy of a pure substance

$$\mathrm{G}^*_\mathrm{m} = \mathrm{G_{ref}} + \mathrm{G_{mix}} \tag{2.18}$$

where two terms have the same notation as in Eq. (2.5) for solutions but have a completely different thermodynamic meaning.

Thermodynamics laws permit us to determine experimentally the change in the Gibbs energy only. If we see something like G = 5000 J·mol^{-1}, this means that the author have chosen some reference state and the Gibbs energy actually corresponds to some difference between the current and reference states. For solutions in Eq. (2.5) the second term, the Gibbs energy of mixing is the difference by definition and its numerical value is defined unambiguously. On the other hand, the numerical value of the first term depends on the chosen reference state for the pure components (see Eq. 2.6).

As a result, it is good to make a similar partition for pure substances (Eq. 2.18), when G_{ref} means some reference state and G_{mix} is the change in the Gibbs energy between the reference state and the given pure substance. This can be achieved by introducing a reaction of formation of the pure substance from chosen key substances, which will represent the reference state

$$\Sigma_j \nu_j B_j = C \qquad (2.19)$$

where C is the given pure substance and B_j are the key substances. Thus, for pure substances

$$G_{mix} = \Delta_r G(19) \qquad (2.20)$$

$$G_{ref} = \Sigma_j \nu_j G_{m,j}(B_j) \qquad (2.21)$$

The notation introduced by Eqs (2.20) and (2.21) might look strange. Yet, this allows us to maintain a similar formal structure for pure substances and solutions, and as a result, this permits us to develop a similar interface. After all, the point phase is a special case of a solution. Along this way, a system can be considered as a vector of phases, some of them being solutions and some point phases. Otherwise, this would require us to develop two different objects for point phases and solutions. In my view, this would lead to unnecessary complexity for thermodynamics algorithms because then it would be necessary to describe a system as two separate vectors. Finally, to make a smoother transition between solutions and point phases, it is assumed that for the point phases $G_{ideal} = 0$ and $G_{excess} = G_{mix}$.

2.2. Background for Thermodynamic Algorithms Library (TD_ALGO)

Generally speaking, an algorithm represents a transformation of the input vector **x** to the output vector **y** according to the given rules. This concept was adopted in order to develop the general interface for the algorithm in the TDLIB.

In the current release of the library, just one thermodynamic algorithm is implemented– the computation of the phase equilibrium conditions. Let us consider the appropriate mathematics for this case.

Let us assume that in the system at equilibrium there are *P* phases. In the general case, a number of components for different phases might vary. Suppose, that the *k*-th phase has C_k components, and the total number of species in the system is given as

$$K = \Sigma_k C_k \qquad (2.22)$$

In Eq. (2.22) the phase Gibbs energies are considered to have the form of Eqs (2.2) or (2.4). In other words, the task of computing the internal variables, if they exist, is assumed to be already solved. Formula vectors of *K* species form a formula matrix of the system (see, for example [7]) and its rank defines the number of independent components in the whole system

$$C = \mathrm{rank}(\mathbf{A}) \qquad (2.23)$$

This equation is analogous to Eq. (2.1) with difference that here the whole system, comprising several phases, is described.

The number of linear-independent chemical reactions that can take place in the system is given as

$$R = K - C = \Sigma_k C_k - C \qquad (2.24)$$

The stoichiometric coefficients in these reactions form a stoichiometric matrix

$$\mathbf{v} = \{ v_{ij} \}, i = 1, \dots, K; j = 1, \dots, R \qquad (2.25)$$

Then, according to the equilibrium criterion, R equations should be held between the chemical potentials of the phase components

$$\Sigma_i \; v_{ij} \; \mu_i(T, p, \mathbf{x}) = 0 \qquad (2.26)$$

Chemical potentials of phase components in the system of equations (2.26) depend on temperature, pressure and mole fractions of components. Hence, the total number of variables to take into account in (2.26) is equal to

$$L = 2 + \Sigma_k (C_k - 1) = 2 + \Sigma_k C_k - P \qquad (2.27)$$

because for the k-th phase there are only $C_k - 1$ independent mole fractions. The difference between the number of variables and the number of equations is defined as a number of degrees of freedom

$$F = L - R = 2 + \Sigma_k C_k - P - (\Sigma_k C_k - C) = 2 + C - P \qquad (2.28)$$

Eq. (2.28) expresses the well-known Gibbs phase rule.

For a given set of P phases, the task is to compose the system of equations (2.26), to choose F from L variables and set their values, and finally to solve the system (2.26) for the other R variables that will be considered as unknowns. Such a task is very common in the inverse problem, when there are experimental results on phase equilibrium, and it is necessary to include them into the simultaneous assessment of thermodynamic properties.

The next function will be used extensively in the TD_ALGO library

$$\mathrm{F}_{\mathrm{min}} = \mathrm{sqrt}\{\Sigma_j \; [\Sigma_i \; v_{ij} \; \mu_i(T, p, \mathbf{x})]^2\} \qquad (2.29)$$

If it is zero (in numerical science, this means close to zero enough), then the system (2.26) is solved and we have an equilibrium state, if it is not, then the value of $\mathrm{F}_{\mathrm{min}}$ is proportional to the proximity of the current state to the equilibrium one.

In a general sense, in the algorithm PhaseEquilibrium implemented in TDLIB, the input vector **x** is formed from the F variables set before solving the system (2.26), and the output vector **y** is defined by the R variables found while solving the system (2.26).

Another important thermodynamic task – computation of the equilibrium composition in the closed system at given temperature, pressure and mole numbers of elements [29] is the first on my list of further extensions to the library.

2.3. Background for Inverse Problem Library (VARCOMP)

Let us assume that we have M experiments, and the i-th experiment comprises N_i experimental points. In the general case, each experimental point in the i-th experiment can be expressed in the next form

$$\{y_{ij}, x_{ij}, \mathbf{z}_i\}; i = 1, \dots, M; j = 1, \dots, N_i \qquad (2.30)$$

where y_{ij} is what has been measured, x_{ij} is what has been changed and the vector $\mathbf{z}_i$ contains other values that have been fixed during the i-th experiment. In other words, Eq. (2.30) describes the conventional experiment in physical chemistry, when only one variable is changed during the experiment, even though the output variable is a function of several variables. As experimental physical chemistry switches to multidimensional experimental design (modern analytical chemistry appears to be doing so), the approach described below may need to be somewhat modified.

As was mentioned in Sec. 2.1 the phase Gibbs energies can contain unknowns to be determined from the available experimental values (Eq. 2.30). Let us denote all the unknowns in the Gibbs energies by the vector Θ. Now we can use thermodynamic laws to express the measured quantity, y_{ij}, as function of controlled variables in the i-th experiment and unknowns in the Gibbs energies

$$y_{ij} = f_i(x_{ij}, \mathbf{z}_i; \mathbf{\Theta}) + \varepsilon_{ij} \quad (2.31)$$

It is always possible for all thermodynamic experiments. As a result, the task of simultaneous assessment is to obtain a set of the unknown parameters in the system of equations (2.31) that gives the best description of the original experimental values in Eq. (2.30). The good description is usually tied with small residuals

$$\varepsilon_{ij} = y_{ij} - f_i(x_{ij}, \mathbf{z}_i; \mathbf{\Theta}) \quad (2.32)$$

so the task is to find such a vector Θ, which gives the smallest ε_{ij}.

There is a common problem tied with the application of Eq. (2.32). In many important cases, the function $f_i(x_{ij}, \mathbf{z}_i; \Theta)$ should be computed numerically, for example, as a result of solving numerically a system of equations (2.26). In practice, this means that in order to compute $f_i(x_{ij}, \mathbf{z}_i; \Theta)$ some numerical subroutine should be called. However, the numerical subroutine well might not find the solution required. A numerical subroutine could just return some code stating that convergence has not occurred or that there were some other problems, which prevented the subroutine to find the answer required. In thermodynamic tasks this is a common case when the initial guess for the vector Θ is bad, and also when the equilibrium in question cannot be computed in principle. Commercial packages available do almost nothing to help thermodynamicist in this case. The ThermoCalc, as I was told, in this case returns an arbitrary value for the $f_i(x_{ij}, \mathbf{z}_i; \Theta)$ with the hope that with the next values of Θ this function will be computed normally. Typically, this problem requires an assessor to find manually the better initial guess for the vector Θ. In thermodynamics circles such a situation is known as "manual" optimization (see, for example [4]).

In order to cope with this problem in our library it is suggested that if $f_i(x_{ij}, z_i; \Theta)$ was not computed successfully, then instead of Eq. (2.32) the residual is computed as follows

$$\varepsilon_{ij} = \text{penalty(proximity to successful computing of } f_i) \quad (2.33)$$

This means that some penalty is assigned to the residual. This should help to keep the whole optimization going with some hope that the optimizer will finally find such a value of the vector Θ when all the residuals could be computed successfully. The trick, to be fortunate along this way, is to make the returned penalty depend on something that characterizes the closeness to successful computing of f_i. This would push the optimizer to change the vector Θ to the right direction.

In the current release of TDLIB, the problem described above appears in the PhaseEquilibrium algorithm within the TD_ALGO library. The good penalty to return in the case of the failure of the numerical subroutine here is F_{min} (Eq. 2.29). This will be further discussed in Sec. 3.3.1.

The conventional approach to define the "best" solution is tied with the minimal value of the sum of squares

$$\mathrm{SS} = \boldsymbol{\varepsilon}' \, \mathbf{D}(\boldsymbol{\varepsilon})^{-1} \, \boldsymbol{\varepsilon} \quad (2.34)$$

where ε is the vector of all residuals ε_{ij} computed by Eq. (2.32) or (2.33) from all experiments (its length is $\Sigma_i N_i$) and $\mathbf{D}(\varepsilon)$ is its dispersion matrix. In the weighted least squares, the dispersion matrix is modeled by the diagonal matrix, containing the inverse variances of experimental points, $\mathbf{D}(\varepsilon) = diag\{\mathrm{D}(\varepsilon_{r,ij})^{-1}\}$.

There are two related problems, which limit the application of the weighted least squares. First, all the variances should be chosen by an expert *a priori* before the assessment. Second, the diagonal structure of the dispersion matrix does not allow us to treat systematic errors. To this end, the linear error model was introduced in Ref. [8]

$$\varepsilon_{ij} = \varepsilon_{r,ij} + \varepsilon_{a,i} + \varepsilon_{b,i}(x_{ij} - x_i) \quad (2.35)$$

Here it is assumed that the total experimental error ε_{ij} consists not only of the reproducibility error $\varepsilon_{r,ij}$, but also of two systematic errors, $\varepsilon_{a,i}$ and $\varepsilon_{b,i}$. Both systematic errors are constant within the i-th experiment, but they are assumed to change randomly among different experiments. The first systematic error, $\varepsilon_{a,i}$, accounts for the shift systematic error and second, $\varepsilon_{b,i}$, for the tilt laboratory factor (tilt systematic error).

The linear error model results in the dispersion matrix of experimental errors, $\mathbf{D}(\varepsilon)$, taking the block-diagonal form (see details in [8]). Each block corresponds to a single experiment and its elements are functions of three variance components

$$\mathrm{D}(\varepsilon_{r,ij}) = \sigma^2_{r,i},\ \mathrm{D}(\varepsilon_{a,i}) = \sigma^2_{a,i},\ \mathrm{D}(\varepsilon_{b,i}) = \sigma^2_{b,i} \qquad (2.36)$$

The considerations above allow us to set up a more general task than the least squares as follows. For the given experimental points (Eq. 2.29), it is necessary to determine the vector Θ with unknown parameters in the thermodynamic model and unknown variance components contained in the dispersion matrix simultaneously. The maximum likelihood method provides a framework to achieve this goal, that is, to maximize

$$\mathrm{L} = -\ln\{\det[\mathbf{D}(\boldsymbol{\varepsilon})]\} - \boldsymbol{\varepsilon}'\,\mathbf{D}(\boldsymbol{\varepsilon})^{-1}\,\boldsymbol{\varepsilon} \qquad (2.37)$$

It also provides the criterion for the best solution for the system (2.31). The algorithm for maximizing Eq. (2.37) under the linear error model given by Eq. (2.35) is described in Ref. [8]. The applications of this approach to materials problem are described in Ref. [9 – 11].

It should be especially mentioned, that the weighted least squares is a special simplified case of the new general task that can be reached by equating the variances of systematic errors (and hence the systematic errors by themselves) to zero and supplying the ratio between variances of the reproducibility error *a priori*. This means that the VARCOMP library can be used to solve the conventional least squares tasks. If you are in doubts about Eq. (2.35), you can employ the traditional treatment without any problem.

2.4. Some Thoughts on Possible Implementations (a single muster-function, interpreting, object-oriented approach)

The key to the successful development of the whole library is to solve the requirements described in Sec. 1.2. The main problem here is tied with the first part of the library. Let us discuss it.

The goal of the first part of the library may look rather simple: to make a framework for modeling the phase molar Gibbs energy (Eq. 2.2) as a function of temperature, pressure and mole fractions. The problem is that we do not know its form; actually it is known that in real applications it could take rather diverse forms. However, from the viewpoint of other parts of the library the phase Gibbs energy should look consistently, that is, it should take the temperature, pressure, and mole fractions as input, and produce the Gibbs energy and/or its derivatives (see Eq. 2.9) as output. The final goal is to represent a system as a uniform vector of phases, even though if the Gibbs energies of different phases might be computed quite differently.

In order to discuss this problem, I will employ C++ although the problem by itself is language independent. In C++ a class with a rather simple interface may express the requirement above.

```
class phase
{
private:
  ...
public:
  double G(double T, double p, double *x);
  double H(double T, double p, double *x);
  double S(double T, double p, double *x);
  double Cp(double T, double p, double *x);
  double V(double T, double p, double *x);
  double alpha(double T, double p, double *x);
  double kappa(double T, double p, double *x);
}
```

The notation above states that there are some internal data structures (they should be described after the keyword private) that are unimportant for other parts of the library. What is of real importance is the external interface after the keyword public, that is, the functions to compute thermodynamics properties of the phase.

The simplest approach to try to achieve our goal is to model Eq. (2.2) by some single master function, that is, to stick to some analytical expression for Eq. (2.2). For example, for some binary solutions the regular solution model can represent the Gibbs energy

$$\mathrm{G_m}(T, p, x_2) = (1 - x_2)\mathrm{G}^*_{\mathrm{m},1} + x_2\mathrm{G}^*_{\mathrm{m},2} + \mathrm{R}T\,(1 - x_2)\ln(1 - x_2) + \mathrm{R}T\,x_2 \ln x_2$$
$$+ (1 - x_2)\,x_2\,\Sigma_i\,(\mathrm{A}_i + \mathrm{B}_i T + \mathrm{C}_i T \ln T)(1 - 2x_2)^i$$

The difference among solutions is supposed to be in number of terms in the sum and in different values of interaction parameters, A_i, B_i, and C_i. If this equation would be enough for all the binary solutions, then the task of its software implementation has had been quite evident and easy.

The simplicity of implementation in the case of a single master function comes from the fact that the internal data structures will be the same for all the phases. As a result, the thermodynamics properties can be executed by means of a call to the same functions.

This can be done relatively easy in any language including procedural ones, for example in Fortran. Some problem with procedural languages along this way is that the syntax for a function/subroutine call will be a bit messy because it is impossible to separate internal data structures with external interface. For example, in C++ the class phase is employed as follows

```
phase foo; //variable foo represents a particular phase
cin >> foo; //foo is initialized from standard input
foo.G(T, p, x); //computing the Gibbs energy
```

In Fortran, this might be expressed as follows (assuming that the internal data structures are within array COEF

```
C declaring internal data structures
      DOUBLE PRECISION COEF(100)
C initializing internal data structures from stream IN
      INIT_COEF(COEF, IN)
C computing the Gibbs energy
      G(COEF, T, P, X)
```

The main difference is that the internal data structures are hidden from the user in the case of C++ and they are exposed to everybody in the case of Fortran. Guess, what happens if for any reason it is necessary to change the internal data structures. In C++ the user should change nothing in his/her program, and in Fortran all the functions calls are to be rewritten. However this is another problem outside of the scope of the current paper, and I am not going to discuss it. After all, tastes differ.

From what I have seen so far, the most advanced master function is developed in ThermoCalc [12] where it allows a user to employ regular, regular association and lattice model along the uniform way. Still, it is quite clear that a single master function approach imposes rather strict limits on the user. A single master function defines just subset of all the possible functions, and one can just hope that this subset is enough for thermodynamic modeling. If some new phase models are developed in the future that cannot fit in the Procrustean bedstead of the master function implemented in the particular software, then a user of this software is in trouble. Actually, even in ThermoCalc the number of master functions is more than one.

The extreme case of the single master function approach is interpretation of the expression during the run-time. Note that interpreting removes almost all the limits imposed on the user, at least principally. Now he/she can enter any functional form for Eq. (2.2).

When I have started working on this project, I thought that writing an interpreter is a nice opportunity to rely upon (see [13, 14]). After several tries, I gave this idea up for three reasons:

1) **Performance.** If for simple cases the Gibbs energy can be expressed rather compactly, for multicomponent solutions it can take a page or even more. In order to interpret a very long expression effectively it is necessary to do something special.
2) **Derivatives.** Besides the Gibbs energy by itself, the user needs its first and second derivatives (see Eqs 2.9, 2.15 and 2.16). The approach with interpreting ordinarily means that derivatives should be taken numerically. This hits precision dramatically. In principle, the derivatives can be taken algebraically in the symbol form of by means of automatic differentiating but in the case of long expressions this seems not to be very practical.
3) **Models with internal variables** (Eq. 2.3 and 2.4). It is necessary to implement some internal solvers to deal with them. I think that it is the most problematic task to solve.

Thus, working with several master functions at once seems to be inevitable because at the present time there is no an uniform thermodynamic model for all the phases. This means that it is necessary to deal with several sets of internal data structures, and computing the thermodynamics properties requires the use of several sets of functions. A typical approach within procedural languages is to introduce some variable describing the model employed, for example

```
int code;
```

Then, while implementing the functions for thermodynamic properties, one has to utilize a sequence of `if` and `then` statements

```
double G(double T, double p, double *x)
{
  if (code == 1) then
  {
  ..return G1(T, p, x);
  }
  else if (code == 2) then
  {
  ..return G2(T, p, x)
  }
  else if (code == 3) then
  {
  ..return G3(T, p, x)
  }
  ...
}
```

Along this way, the main problem is connected with the maintenance of the code in the future when new functions should be added to the implemented ones. If a user of the library decides that it is time to add a new model within the approach above, it would be necessary to insert changes in almost all the library. I would certainly hate this. I prefer not to touch the working pieces of the code because otherwise you never know what the consequences might follow.

Object-oriented programming gives a better framework to handle this situation (see more complete discussion in [15]). In C++, a programmer can employ virtual functions. Now the class phase have in the private area just a pointer to the abstract class `RefPhase` and actually is a simple shell to the actual implementations (in other words, a smart pointer).

```
class phase
{
private:
  RefPhase *bar;
public:
  double G(double T, double p, double *x)
  {
  return bar->G(T, p, x);
  }
...
}
```

The class RefPhase

```
class RefPhase
{
public:
  virtual double G(double T, double p, double *x) = 0;
    //=0 shows that this functions is not yet implemented
...
}
```

is declared abstract because it mere describes the interface required to implement and makes nothing by itself. As a result a user cannot declare a variable of this type.

The main idea is that a user can derive new classes from RefPhase and to implement whatever internal data structures and functions for computing thermodynamic properties. For example,

```
class solution1 : public RefPhase
{
private:
  ... //its own internal data structures
public:
  virtual double G(double T, double p, double *x)
  {
  ... //its own implementation for the Gibbs energy
  }
...
}

class solution2 : public RefPhase
{
private:
  ... //its own internal data structures
public:
  virtual double G(double T, double p, double *x)
  {
  ... //its own implementation for the Gibbs energy
  }
...
}
```

The most striking feature of this approach is that the example presented in the beginning of this section

```
phase foo;
cin >> foo;
foo.G(T, p, x);
```

does not change at all. For a library user everything is the same. The difference is tied with the question - which function will be called in the third line, `foo.G(T, p, x)`: `solution1::G(T, p, x)` or `solution2::G(T, p, x)`. Because the function is declared as virtual, this question will be solved at the run-time. If during input the pointer `bar` (private part of the class phase) is initialized as a pointer to the variable of the class `solution1`, then the function `solution1::G(T, p, x)` will be executed. If `bar` points to the variable of the class `solution2`, then the function `solution2::G(T, p, x)` is called.

Now if a user writes the new solution model

```
class solution3 : public RefPhase
{
private:
  ... //the new internal data structures
public:
  virtual double G(double T, double p, double *x)
  {
  ... //the new implementation for the Gibbs energy
  }
...
}
```

nothing should be changed in the previously written code.

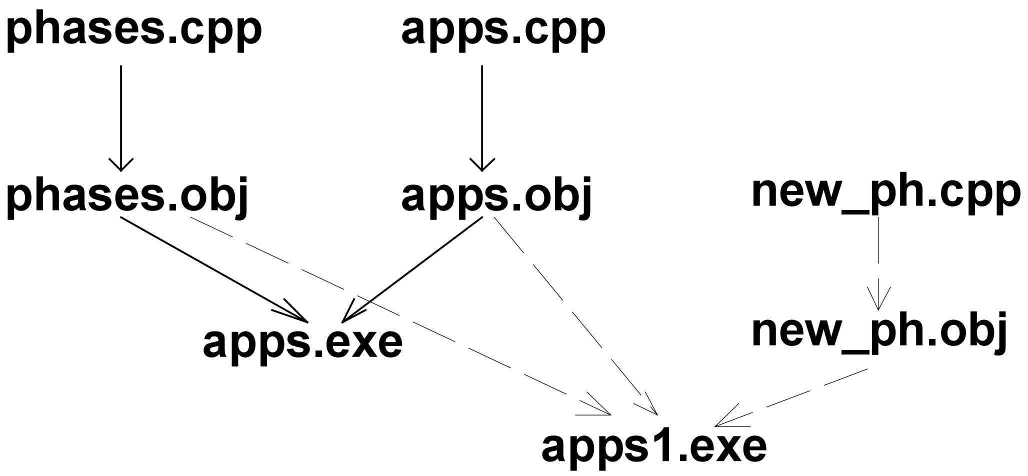


Fig. 2. Adding new model of the multicomponent solution under the object-oriented approach.

This is illustrated in Fig. 2. Let us imagine that first we developed some solution models and programmed them in the file `phases.cpp`. Then we wrote some thermodynamic algorithms to work with in the file `apps.cpp`. After that, we compiled both files to produce object files `phases.obj` and `apps.obj` and then finally linked object files to obtain the executable `apps.exe`. Now we decide to add a new solution model and write it in the file `new_ph.cpp`. If everything was done under the object-oriented approach described above, then during addition of the new model nothing must be changed in the files `phases.cpp` and `apps.cpp`. The only requirement is that the new class should be derived from `RefPhase`. Actually, we do not have even recompile old files. All what is necessary is to compile the file `new_ph.cpp` and to link it with previously created object modules `phases.obj` and `apps.obj`.

To conclude, I believe that the object-oriented approach allow us to build a library that will resemble a construction set and to make a job of thermodynamicist closer to playing LEGO (see Fig. 3).

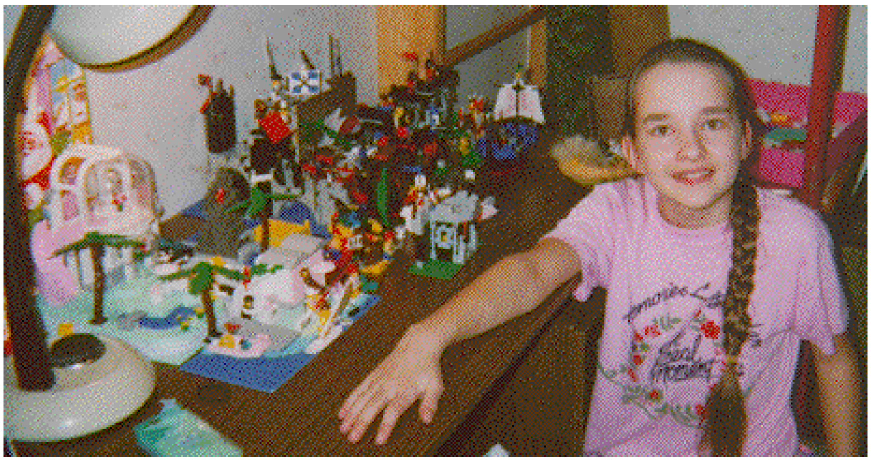


Fig. 3. I wish I had the same joy while doing thermodynamic modeling as my daughter playing LEGO.

2.5. External Representation of the library objects: XML (SGML)

Another important thing is the external representation of the library objects. The first priority here is to develop a portable format that would allow thermodynamicists to exchange their local databases among each other. The format should not be tied with a particular software but rather follow the general needs in thermodynamic modeling (see Sec. 2.1 to 2.3). Also, as was already mentioned, we should expect new models for multicomponent solutions to appear in the future. Therefore, the format should provide necessary means to accommodate this. Finally, it would be good if the format is intuitive and lucid for thermodynamicists.

In my view, Extended Markup Language (XML) that is a subset of Standard Generalized Markup Language (SGML) gives a good framework to achieve these goals. XML allows us to create specific formal rules for a document markup. After that, the documents obeying these rules can be handled rather easily by computers. Moreover, these rules by themselves can be written formally in what is called by Document Type Definition. As a result, the documents written in XML can be considered to be highly portable.

Probably, everyone already heard about Hypertext Markup Language (HTML) - the official language of documents on World-Wide-Web. However, not everyone might be aware that HTML is a subset of SGML. Actually, HTML is just a Document Type Definition in SGML.

The general task of any computer format is to divide the content to the elements, and an element may usually comprise other elements. A computer program reading input should be able to find these elements and take appropriate actions. In order to make it possible, elements are separated by delimiters. As such, the role of any format is to introduce delimiters for all the elements.

Within XML this task is solved by a single rule as follows. Each element is surrounded by two tags, for example

```
<element_name> the element body </element_name>
```

where each tag starts with the symbol "<", ends by the symbol ">" and contains the element name within. The difference between tags is in the symbol "/" preceding the element name in the ending tag. At first glance, this may look a bit awkward, but for a format designer this single rule gives complete freedom to formally describe any sophisticated relationships among elements. And after all, it is very simple. Another important property of SGML is that the starting tag can contain the element attributes, for example

```
<element_name attribute1=value1 attribute2=value2> the element body
</element_name>
```

This is very handy, because quite often there are some element properties that should become known to the software but for some reasons they can not be included into the element body.

XML by itself do not almost impose limits on what elements should be introduced, which elements may contain what elements, what attributes the elements can possess. This is a task of the format designer who should develop the Document Type Definition. However, this is another question, and I will not go into further details because the two rules written above are enough for working with my library. Much more information on SGML are available on-line [16], and the complete description of SGML is in the book [17].

Let me explain this by means of a small example. The Gibbs energy of mixing of the Ba-Cu-Y ternary liquid melts can be described as follows

$$G_{mix} = G_{ideal} + x_1x_2(-7191+3.363T) + x_1x_3 85960$$
$$+ x_2x_3\{v_2(-120940+21.347T) + v_3(-43980+8.133T)\}$$

where x_1, x_2, and x_3 are mole fractions of Ba, Cu, and Y accordingly and v_2 and v_3 are some functions of the mole fractions. If the functional dependence is hard-programmed as a generalized polynomial within the software, the data file can contain something like as follows

```
Liquid
Ba Cu Y
-7191 3.363
85960
-120940 21.347 -43980 8.133
```

The general problem with data files as above is that later on it is quite difficult to remember which number stands for what coefficient in the model. It would be good if the data file were self-descriptive (I think that from this perspective the ThermoCalc format for the Gibbs energy is the best so far). The best way to describe the place of the numbers is to keep the whole expression in the data file, for example

```
Liquid
Ba Cu Y
Gid + x1*x2*(-7191 + 3.363*T) + x1*x3*85960 + x2*x3*(v1*(-120940 +
21.347*T) + v2*(-43980 + 8.133*T))
```

Now there is almost no problem for a human being to read this file but unfortunately this cannot be said about computers. Remember that the goal is not to interpret this expression. The functional dependence is assumed to be hard-programmed, and the software still requires just numbers for input. The expression by itself is redundant and should be seen as text labels, which explain assignment of numbers to a person. Then, to find a middle ground, it is necessary to help the software. As a result, in my format the Gibbs energy of Ba-Cu-Y melts by means of SGML is written as follows

```
<SimpleSolution class=phase id=Liquid>
  <components> Ba Cu Y </components>
  <IdealMixing class=FuncTpx>
    +R*T*x(Ba)*log(x(Ba))
    +R*T*x(Cu)*log(x(Cu))
    +R*T*x(Y)*log(x(Y))
  </IdealMixing>
  <Polynomial class=FuncTpx formalism=Muggianu>
    +x(Ba)*x(Cu)*(
      <func_x> + </func_x>
      <Cp_zero class=func_Tp>
        (
        <coef> -7191 </coef> +
        <coef> 3.363 </coef> *T)
      </Cp_zero>
    )
  </Polynomial>
  <Polynomial class=FuncTpx formalism=Muggianu>
    +x(Ba)*x(Y)*(
      <func_x> + </func_x>
      <Cp_zero class=func_Tp>
        (
        <coef> 85960 </coef> +
        <coef> 0 </coef> *T)
      </Cp_zero>
    )
  </Polynomial>
  <Polynomial class=FuncTpx formalism=Muggianu>
    +x(Cu)*x(Y)*(
      <func_x> +v(Cu)* </func_x>
      <Cp_zero class=func_Tp>
        (
        <coef> -120940 </coef> +
        <coef> 21.347 </coef> *T)
      </Cp_zero>
      <func_x> +v(Y)* </func_x>
      <Cp_zero class=func_Tp>
        (
        <coef> -43980 </coef> +
        <coef> 8.133 </coef> *T)
      </Cp_zero>
    )
  </Polynomial>
</SimpleSolution>
```

If the entire markup is removed we receive the data file as in the previous example, but now, because of the markup, it is relatively easy to write the code to handle this. All the elements will be explained in the next sections.

3. Thermodynamicist Guide

The TDLIB make should build the binary `assess`, which can handle a variety of thermodynamic tasks. This section presents the TDLIB library from the perspective of the `assess` user. The current `assess` functionality is limited. It contains just what I needed for my projects so far. So, if you need to solve a sophisticated thermodynamic problem, you might need to add functionality. However, as was mentioned in the Introduction, the TDLIB presents a framework or infrastructure when the new thermodynamic models and algorithms can be added with absolutely no changes in the previously written code.

The binary assess is old-fashioned. It reads input text files and writes output text files. The responsibility of the user is to prepare input files and to process output files. In the current release, the TDLIB could produce the text files that you can just view or import into the plotting software of your choice and draw a graph. Also the TDLIB could create files, which are directly loadable to the Gnuplot (http://www.cs.dartmouth.edu/gnuplot_info.html, it is free software). As such, creating plots with TDLIB and Gnuplot is almost automatic.

All the files for `assess` are plain ASCII files, and in order to create them you need a text editor, for example NOTEPAD. In my own work, I employ VIM (http://www.vim.org, it is free software). Besides other advantages, it has nice syntax highlighting feature (including SGML) that makes working with the files for the TDLIB a bit easier.

The input files are pretty sophisticated. As was discussed, they are based on XML, and you are supposed to write the straight XML-like elements. It means that you need to have some programming experience. The main idea behind using XML is that this gives a solid background for the future development. In the future it would be rather an easy task to create the Graphic User Interface that deals with these files while providing the user the menu bar and different dialog boxes to work with the objects. Let us take an HTML as an example. Nowadays, almost nobody writes the straight HTML but rather people use GUI's, even though the files created by the different GUI programs can be transferred all over the world due to the background HTML standard.

Still, in the current release of the TDLIB there is no GUI, and you should be ready to work with the plain ASCII files. There are two main reasons. First, the current XML-like elements, developed by me, are not in the stable state. There are a lot of changes as compared with the TDLIB'99, and I expect a lot of changes in the future. It takes quite a time, in order to develop the language. Second, I do not like creating GUI. Personally, I am quite comfortable with console applications, even though I do not mind using GUI's created by others.

The library implements objects, shown in Table 3.1, that, in my view, are necessary for applications of chemical thermodynamics. In the present section, they will be described from the `assess` user perspective, that is, this means how the objects can be written in the file and what the user can do with them.

Note that some objects are referred to as polymorphous. This means that they belong to the some category. Objects in the same category model the same concept, their behavior is similar and they can be used interchangeably. For example, all the phase objects model the Gibbs energy of the phase, Eq. (2.2) and you, for example, can freely change SimpleSolution to AssociatedSolution. The user can extend polymorphous category, if he/she is proficient in programming. As was discussed in Sec. 2.4, the addition of a new object type to the polymorphous category changes nothing the previously written code.

Table 3.1. Objects available in the TDLIB'00

Category	Objects	Polymorphism	Part of the library
Unknown in the vector Θ	coef	no	PHASE
Chemical element	elem	no	PHASE
Chemical formula	formula	no	PHASE
Function in temperature and pressure (func_Tp)	null_Tp, Cp_zero, Cp_const, Cp_BB2, Cp_BB2_ref, Cp_BB4, IVT_Tp, SGTE_Tp, ideal_gas, V_const, alpha_const, alpha_kappa_const, alpha_kappa_const2, compound_Tp, complex_Tp, calc_Tp	yes	PHASE
Species	species	no	PHASE
Function in temperature, pressure and mole fractions (FuncTpx)	NullFuncTpx, IdealMixing, Reference, RedlichKister, HochArpshofen, Polynomial	yes	PHASE
N-component solution (phase)	NullPhase, PointPhase, NumericalDerivatives, SimpleSolution, CuOx_plane, CuOx_ordered_plane, ApBq [a], AssociatedSolution, associated_solution	yes	PHASE
Algorithms	PassThrough, PhaseProperty, reaction, PhaseEquilibrium	yes	TD_ALGO
Transformation $y = f(x_1, x_2, \ldots, x_n)$	convert	no	TD_ALGO
Intermediary between algorithms and other objects	compute	no	TD_ALGO
Objects, describing the computation of the array of numbers	NullOutput, ComputeOutput, SeriesOutput, ResidualOutput, SpinodalOutput	yes	TD_ALGO, VARCOMP
Creating output file	OutputFile	no	TD_ALGO
Residual (Eq. 2.32)	residual	no	VARCOMP
Statistical hypotheses (Eq. 2.35 and 2.36)	series	no	VARCOMP
Experimental series (Eq. 2.30)	series	no	VARCOMP
Global options	globals	no	VARCOMP

[a] This object is not currently available because at the last moment I have found some problem with its implementation.

3.1. Getting started

There are several file types, which `assess` can read for input. Any project must contain one or more model files (.mod) where there should be description of the problem to solve and of the output to compute. Into the model files, one can place objects representing phases, algorithms to solve (phase equilibria) and output to make. Phase objects define the models for the phase Gibbs energies. Phase equilibrium objects define what phase equilibria should be taken into account and their properties in respect to Eq. (2.26). The output objects define what information should be computed to write to the file or to prepare for plotting with Gnuplot. The model files are the most difficult to prepare, because it is necessary to learn format for almost all the objects listed in Table 3.1.

The direct problem, that is, computing equilibrium properties for the given system, can be done by means of `assess` as follows

```
assess model_file1[.mod] model_file2[.mod] … -o output_file
```

where the user should specify which model files to process, and what base name should be given to the output files. Output objects, defined in the model files, should specify the extension that will be added to the base name *`output_file`* from the command line. As a result, the command will produce a number of files with the same name *`output_file`* but different extensions. If no output file is specified, all the output is done to the standard output.

In order to solve the inverse problem the user should make the additional preparation. First, within the model files (typically within the Gibbs energies) it is necessary to declare unknowns in the form of

```
<coef name=B unknown=1> 6.5 </coef>
```

This specifies that the optimizer should take the initial value for the unknown under name B equal to 6.5 and then it should find the better value by maximizing the likelihood function (Eq. 2.37). In order to make it happen the user should add to the model files the description of the residuals (Eq. 2.32) for each experiment and to prepare the data file (.dat) with the experimental values by themselves in the form of Eq. (2.30). Also, the user should take an expert decisions about the quality of the available experiments (statistical hypotheses) and to express them in the set file (.set). Roughly speaking, the set file defines the weighted matrix.

Then, in order to run the inverse problem, one should type

```
assess model_files -d data_files -s set_files -o output_file
```

Here the program read the model files, the data files and the set files, then it starts maximizing Eq. (2.37). The vector of residuals ε is computed for the values in the data file in accordance with Eq. (2.32). $f_i(x_{ij}, z_i; \Theta)$ should be defined in the model file by means of the objects, representing residuals. The set files will be used to specify unknowns in the dispersion matrix $\mathbf{D}(\varepsilon)^{-1}$. After the optimization procedure stops, the program will compute the results, required by the output objects for the final solution obtained.

Now, before going to boring details, let us consider two simple examples.

3.1.1. Simple data fitting – a linear regression (tdlib/ex/line - Ref. [8])

Let us start with the simplest case, not related to thermodynamics. Suppose, it is necessary to determine the parameters a and b from the several experimental according to

$$y_{ij} = \mathrm{a} + \mathrm{b}\, x_{ij} + \varepsilon_{ij}$$

This is a pretty primitive task, but this would allow you to understand the meaning of Eq. (2.35). This case is discussed in Ref. [8] where the special attention is paid to the comparison of the linear error model with the conventional least squares.

In the directory tdlib/ex/line there are next files (see Ref. [8] for the discussion of pseudo-experimental values and the results).

File	Description
ini.mod	The model file
line1.dat	Six pseudo-experimental series, when the systematic errors are absent
line2.dat, line3.dat, line4.dat	Six pseudo-experimental series with systematic errors. The systematic errors increase from line2.dat to line4.dat
ml.set	The systematic errors are taken into account. The hypothesis $\sigma^2_{r,i} = \sigma^2_r$, $\gamma_{a,I} = \gamma_a$, $\gamma_{b,i} = \gamma_b$
wls.set	The ordinary least squares. The systematic errors are ignored. The hypothesis $\sigma^2_{r,i} = \sigma^2_r$, $\gamma_{a,i} = 0$, $\gamma_{b,i} = 0$

It is supposed there are results of six experiments $\{y_{ij}, x_{ij}\}$ from which we would like to determine the parameters `a` and `b`. The pseudo-experimental series are in the `.dat` files in the form as follows,

```
ser_i,
line,
y x,
100 10,
...
140 20;
```

While the pseudo-experimental values have been generated in the files `line2.dat` to `line4.dat` the shift and tilt systematic errors have been added to each series.

The file `ini.mod` defines what `assess` should do with the experimental results. First, there is the `residual` object which computes y^{calc} = a + b x_{ij} and forms a residual y_{ij} - y^{calc}. Then goes the `OutputFile` object which directs to prepare a file with the extension `xy` to plot with Gnuplot.

The command

```
assess ini -d line1 -s ml-o l1_ml
```

process the values from the `line1.dat` under the hypothesis defined in `ml.set` and produces several output files `l1_ml.*`. `l1_ml.lst` is the main listing, `l1_ml.par` is the file, containing the values of unknown parameters found, `l1_ml.xy` contains output of the `OutputFile` object. You can draw a Fig. 2a from Ref [8] by using the command

```
gnuplot l1_ml.xy -
```

These files and also the results of the next commands

```
assess ini -d line1 -s wls -o l1_wls
assess ini -d line2 -s ml  -o l2_ml
assess ini -d line2 -s wls -o l2_wls
assess ini -d line3 -s ml  -o l3_ml
assess ini -d line3 -s wls -o l3_wls
assess ini -d line4 -s ml  -o l4_ml
assess ini -d line4 -s wls -o l4_wls
```

are in the subdirectory out. They process all the pseudo-experimental sets under the two different hypotheses, and it is possible to see the difference between the conventional least squares method and the new approach.

Note that the change to the more complicated models is very simple - you have just to change the function presented within the `convert` object. Because it does real interpreting it is possible to write down any function in the closed form within it.

3.1.2. The Ba-Cu binary system (tdlib/ex/bacu - Ref. [9])

There are two intermetallic compounds in the Ba-Cu system, BaCu and $BaCu_{13}$. There are experiments on the enthalpy of formation of BaCu, the partial enthalpy of the melt, the liquidus temperatures as a function of the melt mole fraction, and the non-variant temperatures.

In the directory tdlib/ex/bacu there are next files (see Ref. [9] for more details on the task in question).

File	Description
sys.mod	The description of the Gibbs energies
alg.mod	The description of the phase equilibria
res.mod	The description of the residuals to treat the experimental values
h.out.mod	The output to plot the partial enthalpies
pd.out.mod	The output to plot the phase diagram
non.out.mod	The output to print the non-variant points and the estimates of the melt spinodal
expr.dat	The experimental values, the data file (see Table 1 in [9])
ml.set	The hypotheses for the recommended solution (solution ML in [9])
ml1.set	The hypotheses for the solution ML1 in [9] (systematic errors are equated to zero, but the reproducibility errors are left unknown). This is the intermediary case between wls.set and ml.set.
wls.set	The weighted least squares assumptions.

The sys.mod file defines four point phases, Ba_s, Cu_s, BaCu and BaCu13 and the binary melt, L. Note that the point phases are related to the one mole of atoms. The Redlich-Kister polynomial is employed to describe the melt (phase L). Then in the alg.mod file there are several objects, defining four mono-variant equilibria, Ba-L, BaCu-L, $BaCu_{13}$-L, Cu-L, and three non-variant equilibria, Ba-L-BaCu, BaCu-L-$BaCu_{13}$, $BaCu_{13}$-L-Cu. After that, in the res.mod file there are residuals for the enthalpy of formation of BaCu, the partial enthalpies of Ba and Cu in the melt, the liquidus temperatures and non-variant temperatures.

There are three files with the output objects. First, h.out.mod prepares the plot in the Gnuplot format for the partial enthalpies, the second, pd.out.mod does the phase diagram. The third file prints in the text format the state of the non-variant points and the estimate of the melt spinodal respectively.

The next commands

```
assess sys alg res h.out pd.out non.out -d expr -s ml -o ml
assess sys alg res h.out pd.out non.out -d expr -s ml1 -o ml1
assess sys alg res h.out pd.out non.out -d expr -s wls -o wls
```

will compute the three solutions for the Ba-Cu system according to the three different set of hypothesis. Each run will produce (see out subdirectory) the .lst file with the main listing, .par file with the new values of unknown parameters, and four files defined by the output objects (.pd, .h, .non and .mis). It is possible to plot the partial enthalpies and the phase diagram with the Gnuplot as follows

```
gnuplot ml.h -
gnuplot ml.pd -
```

The main problem of the assessment, that is, how to choose the initial guess for the unknown parameters, is left without the discussion in this example, because the initial guess in the sys.mod corresponds to the solution recommended in [9].

You can try what happens when the initial guess in not that good. In the ini.par file all the unknowns are equated to zero, what means that the ideal solution and $\Delta G = 0$ for the stoichiometric phases are employed as the initial guess. This is rather a rough initial guess. There is no a straightforward answer how to come to the good initial guess. I would recommend to start the phase diagram optimization with the weighted least squared because at the beginning it is necessary to find some solution with the required topology of the phase diagram. And when the initial guess is rough if in addition to the unknowns in the Gibbs energy one puts unknowns to the variance components, then it will be no good.

It is possible to supply the initial guesses from the .par file to the `assess` with the option –v. Because the Ba-Cu phase diagram is rather simple, the next command will give the desired solution without any additional tricks from the rough initial guess taken from the ini.par file

```
assess sys alg res –v ini –d expr –s wls –o r1
```

The solution `r1` should be equivalent to the `wls` solution, obtained above. Now we can use this solution in order to try the more complicated statistical hypotheses

```
assess sys alg res –v r1 –d expr –s ml1 –o r2
assess sys alg res –v r1 –d expr –s ml –o r3
```

The other strategies to generate the successful initial guess will be discussed in the Case Studies section.

3.2. Describing phases. I. Simple objects in the PHASE library (tdlib/ex/phase)

The main object in the PHASE library is `phase`. Actually, it is a category of the polymorphous objects, which represent the phase Gibbs energy, and each object within this category deals with a particular phase (solution) model. There are underlying objects (`coef`, `elem`, `formula`, `func_Tp`, `species` and `FuncTpx`) that are used to build the phase objects, and in order to understand the `phase` object we have to begin with the underlying objects. Unfortunately, it is impossible to show the examples of the underlying objects with the `assess` program directly – it works just with the phases. As a result, we will start with `coef`, `elem`, and `formula` objects. They are rather simple and they can be presented without the live examples. Then we will go to the `PointPhase` object and discuss with its help the `func_Tp` and `species` objects. After than, we will consider `SimpleSolution` object that is just a container for the `FuncTpx` objects. And finally, we will discuss other objects in the phase category. `CuOx_plane` is the lattice model for the two-component solid solutions of high temperature superconductors similar to $YBa_2Cu_3O_{6+z}$, `CuOx_ordered_plane` is the lattice model for the two-component solid solutions of high temperature superconductors similar to $Y_2Ba_4Cu_7O_{14+w}$, `ApBq` is the lattice model to describe non-stoichiometric compounds $A_pB_{q\pm x}$, and `AssociatedSolution` is a generalized association solution model. Once more, if you need to work with other solution models, not included in the library, you can do it if you can master programming in C++ (see the code and the Programming Guide if I will finally write it).

The format for all phase models can be expressed generally in the next form

```
<phase_model class=phase id=phase_id>
...
</phase_model>
```

where the element with the name *`phase_model`* has two attributes. The attribute `class` has the same value (`phase`) for all the phase models, and it shows that the object *`phase_model`* belongs to the phase category. The attribute `id` is the phase identifier, and it is supposed to be unique within the phase namespace. Within the element, there goes the information specific for each phase model.

The phase can be referenced from other object as follows

```
<phase class=phase IDREF=phase_id></phase>
```

where the *`phase_id`* is supposed to be defined previously.

There are several numerical parameters that affects all the TDLIB library. They can be changed from the default values by means of the `globals` object. Below there are few of them with the default values related to the PHASE library. The others will be introduced later on.

```
<globals
  DefaultT=1000
  Defaultp=1
  R=8.31441
  FirstDerivativeStep=1.49012e-07
  SecondDerivativeStep=6.05545e-05
  GetNuTolerence=2.22045e-13>
</globals>
```

When a user does not specify the temperature and pressure, they will default to `DefaultT` and `Defaultp`. The universal gas constant is equal to `R`. When the derivatives are taken numerically the library employs `FirstDerivativeStep` and `SecondDerivativeStep` as a relative step values for the first and second derivatives accordingly.

When it is necessary to determine the coefficients for the chemical reaction, TDLIB employs the algorithm, described in Ref. [7]. To this end, the linear system of equations based on the formula matrix is solved. If in the resultant vector some element by the absolute value will be less than `GetNuTolerence`, then it is considered to be zero.

3.2.1. Object coef

The object `coef` is a bridge between the PHASE and VARCOMP libraries. It represents a value that may need to be found during the inverse task (members of the vector Θ in Eq. 2.31). For the PHASE and TD_ALGO libraries by themselves (direct problems) it is enough just an anonymous form of `coef`

```
<coef> numerical expression </coef>
```

where within the element one can put any numerical expression, for example

```
<coef> 44.5*3.5 + 5.67 </coef>
```

which will be evaluated on input. Here one can use all the arithmetic operations (+, -, *, /, ^), functions (acos, abs, asin, atan, cos, exp, log10, log, sqrt, tan) and the constant R (by default R = 8.31441).

If we would like to optimize the value defined by `coef` from the experimental results, then it is necessary to add two attributes - the coefficient name, and flag, whether this variable should be optimized. Then the XML representation has the next format

```
<coef id=variable_name unknown=flag_value> numerical expression
</coef>
```

The attribute `id` can take any string, and it shows the coefficient name that will be used by the optimizer. The *`variable_name`* should be unique within the `coef` namespace. The attribute `unknown` can be equal either 0 (false) or 1 (true). All the coefficients with the attribute `unknown=1` are included in the vector Θ, and if `unknown=0`, then the coefficient is fixed and it is not changed during the solution of the inverse problem. The value of the attribute `unknown` as well as the value of the coefficient can be changed in the parameter file (.par).

Examples:

```
<coef id=La1 unknown=1> 3.5 </coef>
<coef id=a unknown=0> 1.5347 </coef>
<coef> 4.68 </coef>
```

In the first example, the coefficient name is equal to `La1`, the variable should be optimized in the inverse task and its initial value is 3.5. In the second example, the coefficient name is equal to `a`, and the coefficient is fixed. The difference between the second and the third line is that the `coef a` will be listed in the `.par` file and the attribute `unknown` can be change from there. The third `coef` is not accessible by the optimizer at all.

There are more features in the `coef` object, but they are unimportant for the PHASE library, and they will be introduced later on - in the VARCOMP library.

3.2.2. Objects elem and formula

The objects `elem` and `formula` do not have a special SGML format because they can unambiguously be represented by a single token, that is, a string with no spaces. For example, CH3CH2OH and Al2(SO4)3 are considered to be valid chemical formulas. Note that it is possible to show a molecular structure in some extent by means of parentheses and repeated structural units. In addition to element names, `formula` can contain plus or minus to show the electron (its name is assumed to be "e"), "+" means e-1, "-" means e1, for example, C60+, F-. In my implementation it is allowed to add some comment to the chemical formula after the underscore symbol, for example, Al2(SO4)3_alpha. This allows us to describe several modifications of the substance, for example, C_diamond and C_graphite.

The element index is represented by a double-precision number. This allows for such notation as Ba0.5Cu0.5 and Ba0.0714Cu0.9286, if this is desired.

The formal description of the `formula` syntax can be expressed as follows (actually, there should be no spaces between the formula parts).

```
formula:
  primary
  primary formula
  formula +
  formula -

primary:
  elem
  elem double
  (formula)double
```

The chemical formula consists from chemical elements. Note that there is new ambiguity in terminology in the present paper: there is a chemical element and there is an XML element. The chemical element name is assumed to be one or two characters. The chemical element name is case sensitive. By default, one can use all the chemical elements from the Periodic Table. They defined in the file elem.cpp (see LIST_ELEMENTS).

A user can define his/her own set of elements. Just put the SGML element `elements` at the beginning of the first .mod file. For example,

```
<elements>
A  100
B  150
C  50
</elements>
```

This defines that in all the formulas one can use three chemical elements only, A, B, and C. The numbers are equal to the chemical element masses.

The order of chemical elements in the SGML element `elements` determines their sorting order. By default, the sorting order of chemical elements is equal to that accepted in Ref. [18]. For the program `assess`, the sorting order of the chemical elements almost does not matter.

3.2.3. Object PointPhase (tdlib/ex/phase/pp)

The object `PointPhase` models one-component solutions, that is, stoichiometric substances. As such, its Gibbs energy is a function in temperature and pressure only, G(*T*, *p*). In order to define it, we need a chemical formula and a function in temperature and pressure. In TDLIB the point phase is considered as a wrapper to the `species` object. For example,

```
<PointPhase class=phase id=test>
  <species> Y2O3
  </species>
</PointPhase>
```

In the directory tdlib/ex/phase/pp/ this is in the file `ex1.mod`. The file `out.mod` defines the output of thermodynamic properties of the phase test. If we give a command

```
assess ex1 out
```

we should see that in our case all the properties are equal to zero, because by default the function in temperature and pressure is equal to zero.

A species as well as the point phase is an entity that can be characterized by a chemical formula and the Gibbs energy in temperature and pressure. However, there is a subtle difference between a species and a point phase. A point phase certainly contains one species. Yet, if we consider equilibria in solutions we might need some species, which do not have the corresponding point phases. As a result, TDLIB contains species, which could be used in any solution model, and point phases, which represent one-component solutions.

Now let us see, how to define the function in temperature and pressure within the `species` object and then what other features the `species` object possesses.

3.2.4. Objects in category func_Tp (tdlib/ex/phase/pp)

The objects in category `func_Tp` shown in Table 3.2 represent the Gibbs energy as a function in temperature and pressure only, $f(T, p)$ (for example, the Gibbs energy of a species). The relationship of $f(T, p)$ with the Gibbs energy is that the derivatives of $f(T, p)$ are considered to comply with Eq. (2.9), otherwise it can be an arbitrary function in temperature and pressure.

Table 3.2. The description of the objects in the category `func_Tp` (pressure is assumed to be in atmospheres, $p^o = 1$ atm).

Object	Description
`null_Tp`	$G(T, p) \equiv 0$
`Cp_zero`	$G(T, p) = a + bT$ ($C_p = 0$)
`Cp_const`	$G(T, p) = a + bT + cT\ln T$ ($C_p = \text{const}$)
`Cp_BB2`	$G(T, p) = a + bT + cT\ln T + dT^{0.5}$ ($C_p = -c + 0.25dT^{-0.5}$ [19])
`Cp_BB2_ref`	$G(T, p) = H_o - S_oT + a(T - T_o - T\ln(T/T_o)) + 4b(T^{0.5} - T_o^{0.5})$ ($C_p = a + bT^{-0.5}$ [19])
`Cp_BB4`	$G(T, p) = a + bT + cT\ln T + dT^{0.5} + eT^{-1} + fT^{-2}$ ($C_p = -c + 0.25dT^{-0.5} - 2eT^{-2} - 6fT^{-3}$ [19])
`IVT_Tp`	$G(T, p) = a + bT + cT\ln T + dT^2 + eT^3 + fT^4 + gT^{-1}$ ($C_p = -c - 2dT - 6eT^2 - 12fT^3 - 2gT^{-2}$) [18]
`SGTE_Tp`	$G(T, p) = a + bT + cT\ln T + dT^2 + eT^3 + fT^7 + gT^{-1} + hT^{-9}$ ($C_p = -c - 2dT - 6eT^2 - 42fT^6 - 2gT^{-2} - 90hT^{-10}$) [20]
`ideal_gas`	$G(T, p) = RT\ln(p)$ ($V = RT/p$)
`V_const`	$G(T, p) = vp/p^o$ ($V = v/p^o$)
`alpha_const`	$G(T, p) = ve^{\alpha T}(p/p^o - 1)$ ($V = ve^{\alpha T}/p^o$)
`alpha_kappa_const`	$G(T, p) = ue^{\alpha T}(1 - e^{\kappa(1 - p/po)})$ ($V = u\kappa e^{\alpha T}e^{\kappa(1 - p/po)}/p^o$)
`alpha_kappa_const2`	$G(T, p) = (v/\kappa)e^{\alpha T}(1 - e^{\kappa(1 - p/po)})$ ($V = ve^{\alpha T}e^{\kappa(1 - p/po)}/p^o$)
`calc_Tp`	A simple interpreter for any $G(T, p)$ in the closed form.
`complex_Tp`	A simple shell to combine two objects $G(T, p) = G_1(T, p) + G_2(T, p)$.
`compound_Tp`	Contain a vector of `func_Tp` (classes within the vector must be in the hierarchy of `simple_Tp`) to work with compound functions in temperature and pressure.

Probably, there are too many objects in this category, and it would be possible to leave here less objects. The reason is that this is the first polymorphous category of objects I have created and it has been used extensively as a “shooting-range” to check how it works.

In the directory tdlib/ex/phase/pp you will find files with the name after the `func_Tp` object which show you their syntax. The point phase is used as a shell to demonstrate how `func_Tp` objects works. The command as follows

```
assess cp_const out
```

will show you the results of the computation of the thermodynamics properties from the given `func_Tp` object.

All objects `func_Tp` have the attribute `class` which has the same value, `func_Tp`, and it shows that this object belongs to the `func_Tp` category.

The simplest object is `null_Tp` which represents the case when G(T, p) = 0. It is usually used as a stub when you have to place a `func_Tp` object but for some reason you want it to be equal to zero. Actually, `ex1.mod` is equivalent to `null_tp.mod` because in the first case the `null_Tp` object was added implicitly.

The classes from `Cp_zero` to `SGTE_Tp` represents some special cases for G(T, p) when the mole volume is assumed to be zero (V = 0, see Table 3.3). The objects from `ideal_gas` to `alpha_kappa_const2` represent special cases for some equations of state (C_p = 0, see Table 3.3). The functions and their derivatives for these objects are hard coded. As a result, you can change nothing in the format of these elements but to change the values of `coef` objects within them. The main idea here is to make the file format self-describing. However, the text outside `coef` elements is considered by the software just as text labels, and there is absolutely no interpreting when the thermodynamic functions, associated with these objects, are computed.

Two objects of different `func_Tp` objects might be equivalent from mathematical point of view, for example as in the case below

```
<Cp_zero class=func_Tp> (
  <coef> 10531 </coef> +
  <coef> 348.5 </coef> *T)
</Cp_zero>
<Cp_const class=func_Tp> (
  <coef> 10531 </coef> +
  <coef> 348.5 </coef> *T+
  <coef> 0 </coef> *T*log(T))
</Cp_const>
```

However there is some difference between them if we compare the computer resources required. The second object takes more RAM because it needs to reserve space for additional information. Also, computing in the second case can take a bit more time, because the computer might well take log(T) before multiplying it by zero. I think this depends on a compiler.

The difference between `Cp_BB2` and `Cp_BB2_ref` is in different meaning of the `coef` objects. The functions are equivalent between each other, as they can be transformed to each other with no loss of information. However, if we consider the inverse problem, then the unknowns which could be introduced will be different, and it might influence the numerical properties of the inverse problem. The same concerns the `alpha_kappa_const` and `alpha_kappa_const2` objects.

The object `SGTE_Tp` can deal with magnetic contribution as described in Ref. [20]. To this end, one can add three attributes Bo, Tc and pm to the object. For example,

```
<SGTE_Tp class=func_Tp Bo=0.22 Tc=95 pm=0.28>
...
</SGTE_Tp>
```

The object `complex_Tp` combines any two `func_Tp` objects. This allows us to combine a function, describing the temperature dependence of the heat capacity, with another function, describing the equation of state. Run

```
assess complex_tp out
```

and compare the output with

```
assess cp_bb2 out
assess ideal_gas out
```

The object `calc_Tp` is a simple interpreter in temperature and pressure. Its body could contain any function in temperature and pressure. Within the `calc_Tp` object one can use all the arithmetic operations (+, -, *, /, ^), functions (acos, abs, asin, atan, cos, exp, log10, log, sqrt, tan) and the constant R = 8.31441. Thus, `calc_Tp` can be used when your needs cannot be handled by special cases included in the library (see Table 3.3). Compare

```
assess complex_tp out
assess calc_tp out
```

The output is the same, however the `calc_Tp` object does real interpreting. Note though, that because of interpreting the `calc_Tp` object takes more RAM, its performance is slower by several times and the first and second derivatives, listed in Eq. (2.9), are taken numerically.

The last object `compound_Tp` is to deal with compound functions. It contains a vector of `func_Tp` objects. The T,p-plane is supposed to be covered by a grid of rectangles, described by a set of T,p-values as follows

T_1,p_1	T_1,p_2	…	T_1,p_n
T_2,p_1	T_2,p_2	…	T_2,p_n
…	…	…	…
T_m,p_1	T_m,p_2	…	T_m,p_n

and G(T, p) within each rectangle is assumed to be represented by a simple `func_Tp` object. The continuity of the whole compound function and its derivatives is up to the developer of the compound G(T, p) function.

Run

```
assess compound_tp out
```

and compare the output with

```
assess complex_tp out
```

Finally, it should be mentioned that any `func_Tp` object could have an `id` attribute. Its value should be unique for the `func_Tp` namespace. Then this object can be referenced from other object as follows

```
<func_Tp class=func_Tp IDREF=func_Tp_id></func_Tp>
```

where the *`func_Tp_id`* is supposed to be defined previously. See file `ex2.mod` as an example.

3.2.5. Object species (tdlib/ex/phase/pp/)

As was mentioned above, basically a species should be a combination of two objects: `formula` and `func_Tp`. However, it is helpful to allow the use of Eq. (2.18) when the total Gibbs energy of the species is partitioned to the two parts, the Gibbs energy of some reaction (Eqs 2.19 and 2.20) and the sum of the Gibbs energies (Eq. 2.21). Let me explain this with the next example, related to the point phases. Note that the same concerns the species in the solution.

There are five stoichiometric compounds in the Cu-Y system, Cu_6Y, Cu_4Y, Cu_2Y_7, Cu_2Y, and CuY. Hence, on order to describe the equilibria, one has to employ seven species for the seven point phases, five intermetallic compounds and two pure components, and accordingly seven Gibbs energies G(T, p)$_i$ (i lists all seven compounds). A typical thermodynamic ambiguity is that we cannot determine the absolute Gibbs energies, although this has no implications for computing the equilibrium composition. As a result, two of seven Gibbs energies can be equated to arbitrary functions (their choices depend on the accepted reference states in the project), and this does not change equilibria computations.

Eqs (2.18) to (2.21) allows us to introduce five reactions

$$p\,Cu + q\,Y = Cu_pY_q$$

(the formation of intermetallic compounds from pure components). The five Gibbs energies of reactions are defined unambiguously because they correspond to change in the Gibbs energy, and, actually, all equilibria computations depend just on these reaction Gibbs energies. Still, it is useful to think about seven Gibbs energies (one species - one Gibbs energy). This simplifies many formulas in chemical thermodynamics. Then we can express all seven Gibbs energies by means of two arbitrary functions (the Gibbs energies of pure components, see G_{ref} in Eqs 2.18 to 2.21) and five unambiguous reaction Gibbs energies (G_{mix} in Eqs 2.18 to 2.21). This would simplify significantly support of the database in the future. If we need to change the current reference state for any reason, then we have to change just the two Gibbs energies of pure components. The Gibbs energies of intermetallic compounds will then change instantly because of Eqs (2.18) to (2.21).

Another reason for employing Eq. (2.18) is tied with inverse tasks. Here we need to put unknowns in the Gibbs energies of intermetallic compounds. Much better choice is declaring unknowns within the reactions Gibbs energies (Eq. 2.19 and 2.20) because this usually leads to a more stable numerical task.

On the other hand, Eq. (2.18) is general. A user can always equate G_{ref} to zero and, in the example above, define the seven Gibbs energies independently. Certainly, he/she must not forget about the thermodynamics laws, but this is another question.

In order to describe the Eq. (2.18), one can put into the `species` object the SGML element `ref_plane`. The `func_Tp` object before the `ref_plane` element will mean G_{mix}, and the `ref_plane` by itself - G_{ref}. By default, when the `ref_plane` element is absent, it is assumed that $G_{ref} = 0$.

The file species.mod in the directory tdlib/ex/phase/pp demonstrates the syntax. The file out2.mod shows how to retrieve the full property and the parts, `ref` and `mix`. The command

```
assess species out2
```

will show you how it works.

The element `ref_plane`, if present, contains a list of pairs of numbers and `species`. Note that this is a recursive definition, because the `species` within the `ref_plane` can have their own reference planes. The recursion is allowed and supported by the library.

If on input all the numbers before species are equal to zero or they are absent, then during the input the formation reaction of the current species from the components in `ref_plane` will be equated automatically. Otherwise, the correctness of the reaction is not checked. The example is in the file k2so4.mod. In order to see it, it is necessary to put the –m flag to the `assess`. For example,

```
assess -m k2so4
```

With this flag the `assess` rewrite the model file and you can see how it has interpreted the input. Note, that first while making output the `assess` writes the `globals` object. It will be discussed later on.

The species could have an `id` attribute (see species.mod). In this case it can be referenced from the other objects, for example

```
<species IDREF=Y2O3></species>
```

This will work if and only if the species with the `id` equal to Y2O3 was already defined earlier.

The namespaces for the species and phases are different. However, because the point phase is, after all, a species, the TDLIB features the automatic promotion of species to the point phase. This means, that if there is a reference to the phase

```
<phase IDREF=ph1></phase>
```

and the `ph1` is not found among the defined phases, then TDLIB will search the `ph1` in the species namespace, and if it is found there, the point phase will be formed automatically from the species with `id` equal to the given one.

3.2.6. Object SimpleSolution (tdlib/ex/phase/simple/)

In general, the Gibbs energy for the simple solution models (for example, non-ordering solutions) can be represented as a sum over some functions in temperature, pressure and mole fractions

$$\mathrm{G_m}(T, p, x_1, ..., x_C) = \Sigma_k f_k(T, p, x_1, ..., x_C) \qquad (3.1)$$

Typically, the functions on the right side of Eq. (3.1) are classified as $\mathrm{G_{ref}}$ (Eq. 2.6), $\mathrm{G_{ideal}}$ (Eq. 2.8), and $\mathrm{G_{excess}}$ and Eq. (3.1) usually looks like as follows

$$\mathrm{G_m}(T, p, x_1, ..., x_C) = \Sigma_i\, x_i\, \mathrm{G}^*_{\mathrm{m},i}(T, p) + \Sigma_i\, x_i\, \mathrm{R}T \ln x_i + \Sigma_k\, \mathrm{g}_{\mathrm{int},k} \qquad (3.2)$$

where $i = 1, ..., C$, $(C \geq 2)$ and three terms correspond to $\mathrm{G_{ref}}$, $\mathrm{G_{ideal}}$, and $\mathrm{G_{excess}}$ accordingly.

In the current release, Eq. (3.1) and not Eq. (3.2) was chosen as the ground to model the Gibbs energy without the internal parameters, because it gives more freedom and flexibility. As a result, the object `SimpleSolution` is considered as a container for polymorphous objects in category `FuncTpx` which are used to model f_k in Eq. (3.1). Thus, if you would like to create any new phase models without internal parameters, all you need in TDLIB is to program a new object in the category `FuncTpx`. In my understanding, Eq. (3.1) should handle all the cases here.

The file `ex1.mod` in the directory tdlib/ex/phase/simple/ shows the empty `SimpleSolution` object for the fictitious binary A-B system, and the file `out.mod` describe the output of various properties. The command

```
assess ex1 out
```

will show that as expected the empty `SimpleSolution` object produces zeros for all the properties.

Now let us see, how to define the function in temperature, pressure and mole fractions within the `SimpleSolution` object. Note that even if the example will deal with the two-component solution, there is almost no software limits on a number of components and a number of `FuncTpx` objects you could put within `SimpleSolution` object. You are just limited by the size of available RAM and the processor speed.

3.2.7. Objects in category FuncTpx (tdlib/ex/phase/simple)

In order to represent $\mathrm{G_{ref}}$ TDLIB contains the object `Reference`. It allows us to work with some more general functions than Eq. (2.6). Let us discuss this with an example of the two-binary solution. Typically $\mathrm{G_{ref}}$ here would be represented as

$$\mathrm{G_{ref}} = x_A\, \mathrm{G}^*_{\mathrm{m},A}(T, p) + x_B\, \mathrm{G}^*_{\mathrm{m},B}(T, p) \qquad (3.3)$$

However, we could express $x_A = 1 - x_B$ and receive

$$\mathrm{G_{ref}} = \mathrm{G}^*_{\mathrm{m},A}(T, p) + x_B\, \Delta\mathrm{G}^*(T, p) \qquad (3.4)$$

or alternatively $x_B = 1 - x_A$

$$\mathrm{G_{ref}} = \mathrm{G}^*_{\mathrm{m},B}(T, p) + x_A\, \Delta\mathrm{G}^{*'}(T, p) \qquad (3.5)$$

While Eqs (3.3) to (3.5) are equivalent in the mathematical sense, in the inverse problems when functions in temperature contain unknowns there might be a difference in numerical stability of the optimization problem. Here the transformation from Eq. (3.3) to (3.4) and (3.5) allows us to reparameterize the goal function, and if this leads to the less correlated unknowns, this would improve the convergence. Also, in my view, the Eq. (3.4) or (3.5) more suitable to describe the solid solutions kind of AB_{P+Z} where the end members of the binary solution are AB_P and AB_{P+1}.

As a result, the `Reference` object represents a general function as follows

$$G_{ref} = G_o(T, p) + \Sigma_i x_i G_i(T, p) \tag{3.6}$$

where the sum is over an arbitrary set of the mole fractions of components. The files `ref1.mod` to `ref3.mod` represent Eqs (3.3) to (3.5) accordingly, and the next commands will show that the results are identical

```
assess ref1 out
assess ref2 out
assess ref3 out
```

It is possible to put several `Reference` objects. It is useful, for example, when it is necessary to define the ideal gas phase. The file `ref4.mod` shows how it could be done.

Formally speaking, the `Reference` object contains pairs of `func_x` and `species` elements. The use of species allows us to model the Gibbs energy of pure component of the solution with the use of Eqs (2.18) to (2.21) (see also Section 3.2.5).

The `IdealMixing` object models Eq. (2.8). If this object is left empty on input (see file `ideal1.mod`) it represents exactly Eq. (2.8). Again you can obtain the numerical properties with the command

```
assess ideal1 out
```

If you unsure how TDLIB will understand the input, it is always possible to use `-m` flag at the command line, for example

```
assess -m ideal1
```

and all the objects will be shown in a form which could be called a “canonical”.

The `IdealMixing` object also allows us to work with some more general function than Eq. (2.8), namely with

$$G_{ideal} = \Sigma_i b_i x_i RT \ln x_i \tag{3.7}$$

where b_i are arbitrary numbers and the sum can be taken over arbitrary subset of the components. The file `ideal2.mod` demonstrates this possibility.

There is a numerical problem associated with Eqs (2.8) and (3.7) concerning chemical potentials and partial entropies. The part in chemical potential associated with Eq. (3.7) is equal to

$$\mu_{ideal,i} = b_i RT \ln x_i \tag{3.8}$$

and when the mole fraction is zero we have a negative infinity. What happens when zero or negative values are substituted for x_i in Eq. (3.8) depends on a compiler. Borland C++ up to 5.02 did not like it. gcc in such a situation behaves in a more reasonable manner, for it uses “infinity” and “not-a-number” values. Still, it is necessary to put in special efforts to handle this case in the library. The current decision is to use an idea which I have found in Ref. [27], that is, to change $\ln(x)$ at values of x less than eps to the next function

$$f(x) = \ln(eps) - 1.5 + (x/eps)(2 - 0.5\, x/eps) \tag{3.9}$$

The function above and its first derivative is continuous with the $\ln(x)$ and, at the same time, leads to the reasonable values when x is zero or less than zero. The TDLIB behavior is controlled by the two global parameters which are given below with their default values

```
<globals
  EpsForNegLog=1e-07
  NegativeLog=1>
</globals>
```

When `NegativeLog` is equal to one the use of Eq. (3.9) is allowed and then `EpsForNegLog` sets the value of eps in Eq. (3.9). This simplifies the development of the thermodynamic algorithms because now it possible to freely use the zero and even negative values of mole fractions. However, this decision has its backside, that is, if you need to compute the chemical potential of some component when its mole fraction is very low by default the values obtained might not be accurate.

In order to see this effect, run the next commands in the directory `tdlib/ex/phase/simple`

```
assess ideal1 outsmall
assess neglog ideal1 outsmall
```

In my own practice, I have encountered this backside few times when in order to proceed further I needed to change the default values of `EpsForNegLog`. In the future, this has to be done in some more intelligent way that would not require the human intervention.

G_{excess} is expressed as the sum over interaction terms, $g_{\text{int},k}$, including binary, ternary and so on interactions. The number of binary interaction terms is equal to the number of all the combinations between two mole fractions, x_{i1} and x_{i2}, the number of ternary interaction terms is equal to the number of all the combinations among three mole fractions, x_{i1}, x_{i2}, and x_{i3}, and so on.

The binary interaction term in the binary solution can be generally expressed as

$$g_{AB} = \Sigma_r\, a_r(T, p)\, f_r(x_A, x_B) \qquad (3.10)$$

where $a_r(T, p)$ is some function in temperature and pressure, sometimes referred to as the interaction parameter, and $f_r(x_A, x_B)$ is some function in mole fractions, such that it is equal to zero for pure components. In the binary solution $x_A + x_B = 1$, and there are many equivalent forms of Eq. (3.10) for the same function $f_r(x_A, x_B)$. However, if we use Eq. (3.10) for the multicomponent solution directly, these equivalent form become different because here $x_A + x_B \neq 1$. Several projection formalism have been introduced (see, for example, Ref [21]). The most famous are Kohler and Muggianu formalisms, described as

$$v_i = x_i/(x_A + x_B) \qquad (3.11)$$

$$v_i = x_i + (1 - x_A - x_B)/2 \qquad (3.12)$$

accordingly, where i means A or B. In these equations one takes mole fractions x_A and x_B from the multicomponent system, and obtains the mole fractions v_A and v_B that should be used in the next equation

$$g_{AB} = (x_A x_B)/(v_A v_B) \cdot \Sigma_r\, a_r(T, p)\, f_r(v_A, v_B) \qquad (3.13)$$

Thus, Eq. (3.13) represents the binary interaction term in the general case for a multicomponent solution. In the case of the binary solution Eq. (3.13) is equivalent to Eq. (3.10). There are several choices for $f_r(x_A, x_B)$ (see, for example, Ref. [22]). The current release of TDLIB contains three objects, `RedlichKister`, `HochArpshofen`, and `Polynomial` to describe the most popular interactions. As for other polymorphous categories, a user, proficient in programming, can add new objects. The general polynomial can be modeled with the `Polynomial` object as

$$g_{AB} = x_A x_B\, \Sigma_r\, a_r(T, p)\, v_A^{rA}\, v_B^{rB} \qquad (3.14)$$

where the user can employ arbitrary number of terms with arbitrary powers. Note that this includes a Borelius polynomial

$$g_{AB} = x_A x_B\, \Sigma_r\, a_r(T, p)\, v_A^{R-r}\, v_B^{r} \qquad (3.15)$$

where $r = 0, \ldots, R$, as a special case.

Redlich-Kister polynomial is expressed as

$$g_{AB} = x_A x_B\, \Sigma_r\, a_r(T, p)\, (v_A - v_B)^r \qquad (3.16)$$

where $r = 0, \ldots, R$, and Hoch-Arpshofen [23] polynomial as

$$g_{AB} = (x_A x_B)/(v_A v_B) \cdot \{\Sigma_r\, a_r(T, p)\, (v_A - v_A^r) + \Sigma_q\, a_q(T, p)\, (v_B - v_B^q)\} \qquad (3.17)$$

where $r \subset (2, \ldots, R)$, and $q \subset (2, \ldots, R)$. Hoch and Arpshofen employed just two members In Eq. (3.17), so latter can be viewed as the generalization. Actually, one should not include all the members in both sums over r and q in Eq. (3.17), because in this case some members may become linear dependent.

For the direct tasks, computing the equilibrium composition, Eqs (3.14) to (3.17) are equivalent, as one can transform them equivalently in each other. In the inverse task, this is not so. Because the basis functions are different (and so is the set of unknowns), their behavior also differs. The basis functions in the Redlich-Kister polynomial are close to the orthogonal ones, and this makes the optimization task somewhat easier. On the other hand, the Hoch-Arpshofen polynomial allows us to support the necessary topological behavior of the Gibbs energy more easily. The basis functions here have a nice feature: each function $x_A - x_A^r$ is concave in the closed interval [0, 1]. This means that only the sign of $a_r(T, p)$ determines whether $a_r(T, p)(x_A - x_A^r)$ is convex or concave. If you add to this feature a statement, that the sum of convex functions is also convex, then it gives some additional power into the hand of the assessor.

The M-ary interaction term of C-component solution is defined as the interaction between M from C components with their mole fractions expressed as x_j, where the index j lists the mole fractions in the given set of M components. However, only the general polynomial (`Polynomial` object) allows us to easily generalize Eq. (3.10) to M-ary interaction term from the binary one. In this case, Eq. (3.14) takes the next form

$$g_{\text{M-}ary} = (\Pi_j\, x_j)\ \Sigma_r\ a_r(T, p)\ (\Pi_j\ v_j^{nrj}) \qquad (3.18)$$

The sum in Eq. (3.18) is over any possible products of v_j. The projections formalisms for the M-ary interaction term (Eqs 3.11 and 3.12) become

$$v_i = x_i/(\Sigma_j\, x_j) \qquad (3.19)$$

$$v_i = x_i + (1 - \Sigma_j\, x_j)/M \qquad (3.20)$$

accordingly.

All objects to represent interactions have a common attribute, `formalism`. The value of `formalism` set the projection formalism to `NoChange`, `Muggianu`, `Kohler`. The `NoChange` means $v_i = x_i$, as in the case when the order of interaction is equal to the number of components, $M = C$. If the order of interaction is equal to the number of components of the solution than the formalism on input is reset to `NoChange` because it takes less computational time. For `RedlichKister` objects the values `NoChange` and `Muggianu` are equivalent. If you unsure, how TDLIB understood your input, as was mentioned above, use `-m` flag on the command line and `assess` will produce the output of the model.

The internal format of interaction objects mimics the mathematical expression, Eqs. (3.14) to (3.17). Each item is represented by a pair of SGML elements, The first, `func_x`, serves as a text label to show a power of the basis function, the second, which should be of the `func_Tp` category, describes $a_r(T, p)$. The user should indicate the names of the components, the projection formalism, and the items required. On input the order of the items can be in any order.

As in the case of `func_Tp` objects, the representation of the mathematical expression in the format is done just to make the format self-descriptive, and this does not mean interpreting at run-time. Eqs. (3.14) to (3.17) are hard-coded within the objects, hence, they have a good performance and all the derivatives (Eq. 2.16) are programmed in the closed form.

The files `rk1.mod`, `ha1.mod`, `ha1_.mod`, and `pol1.mod` in the directory `tdlib/ex/phase/simple` contains the different interaction objects which describe the same mathematical function. However, because the type of the polynomial is different the numerical values of the coefficients are different. The file `outfrml.mod` produces the Gibbs energy for these objects under the different formalisms. If the sum of the two mole fractions, taking part in the interaction, is equal to one, than there is no difference between the objects under any formalism. If the sum of these two mole fractions is not equal to one, than there is a difference. You can see it, if you type in the next commands

```
assess rk1 outfrml
assess ha1 outfrml
assess ha1_ outfrml
assess pol1 outfrml
```

The files `rk2.mod`, `ha2.mod`, and `pol2.mod` demonstrate the same with a bit more complicated polynomial. Run the next commands

```
assess rk2 outfrml
assess ha2 outfrml
assess pol2 outfrml
```

to view it.

The `FuncTpx` objects could have an `id` attribute. In this case it can be referenced from the other objects, for example

```
<FuncTpx class=FuncTpx IDREF=id_value></FuncTpx>
```

Finally, the library supports the `NullFuncTpx` object, which models the interaction term identically equal to zero. It is not very useful in thermodynamic applications, but it is helpful for internal tasks of the TDLIB library.

3.2.8. Object CuOx_ordered_plane (tdlib/ex/phase/y247/)

The `CuOx_ordered_plane` object was designed to work with the $Y_2Ba_4Cu_7O_{14+w}$ (Y247) superconductor. Its structure is pretty similar to that of Y123 (see below) but the Y247 phase experimentally was not found in the tetragonal form. This allowed us to speculate that, in this case, the order parameter x is always equal to $z/2$, what corresponds to the case when the basal plane is in the completely ordered state. As a result, the Gibbs energy of the Y247 does not contain an internal parameter, and it can be expressed as

$$\Delta_{ox}G(T, w) = g_1(T) + w\, g_2(T) + w(1 - w)\Sigma_i a_i(T)(1 - w)^i + \\ 2\, RT[w \ln w + (1 - w) \ln (1 - w)] \qquad (3.21)$$

In the previous version of TDLIB this can not be handled by the polynomial model, and in order to work with Eq. (3.21), the `CuOx_ordered_plane` object was programmed. In TDLIB'00 Eq. (3.21) can be expressed by means of `SimpleSolution`, and thus the `CuOx_ordered_plane` object can be considered as obsolete. It is left in the library just because it was already programmed earlier.

The files `plane.mod` and `simple.mod` in the directory `tdlib/ex/phase/y247/` displays how to represent Eq. (3.21) by means of `CuOx_ordered_plane` and `SimpleSolution` objects accordingly. The next commands

```
assess plane outtd -o p
assess simple outtd -o s
```

will compute the various thermodynamic properties for the two objects to the two files, `p.td` and `s.td`. They should be absolutely identical, what can be checked by means of

```
diff p.td s.td
```

3.2.9. Object NumericalDerivatives (tdlib/ex/phase/y247/)

While the solution model is programmed there are a lot of possibilities for mistakes. In order to simplify the debugging, the `NumericalDerivatives` object computes all the derivatives of the phase model numerically. The file `num.mod` in the directory `tdlib/ex/phase/y247/` shows the very simple syntax, and the file `outnum.mod` compares the hard-programmed and numerical derivatives. Run

```
assess plane num outnum
assess simple num outnum
```

to see that the derivatives for the `CuOx_ordered_plane` and `SimpleSolution` objects was programmed correctly.

3.3. Describing phases. II. Solution models with the internal parameters in the PHASE library (tdlib/ex/phase)

For a solution model with internal parameters, TDLIB allows us to partition Eq. (2.10) as follows

$$G_m(T, p, \boldsymbol{x}) = G_{ref}\{T, p, \boldsymbol{x}\} + G_{mix}\{T, p, \boldsymbol{x}, \boldsymbol{y}(T, p, \boldsymbol{x})\} \qquad (3.22)$$

where the meaining of G_{ref} and G_{mix} is changed a bit as compared with Eq. (2.5) for the simple solution models. Here G_{ref} denotes the part of the Gibbs energy, which does not depend on internal parameters, and G_{mix} describes the part with internal parameters. Such a division permits us to optimize computations for the whole model. Also in Eq. (3.22) and below the subscript *eq* at internal parameters is omitted to simplify the equations.

Then there are the two cases, with one independent internal parameter and many independent internal parameters accordingly. Let us start with the former. We have

$$G_m(T, p, \boldsymbol{x}) = G_{ref}\{T, p, \boldsymbol{x}\} + G_{mix}\{T, p, \boldsymbol{x}, y(T, p, \boldsymbol{x})\} \qquad (3.23)$$

where y is determined by solving the non-linear equation

$$F(y;\ T, p, \boldsymbol{x}) = 0 \qquad (3.24)$$

The first derivatives of the Gibbs energy in the form of Eq. (3.23) can be found as

$$\partial G_m/\partial z = \partial G_{ref}/\partial z + \partial G_{mix}/\partial z + (\partial G_{mix}/\partial y)(\partial y/\partial z) \qquad (3.25)$$

where z is T, p, or x_i. The derivative for the internal parameter can be determined by means of the rule for the implicit derivatives

$$\partial y/\partial z = -\ (\partial F/\partial z)/(\partial F/\partial y) \qquad (3.26)$$

Thus, according to Eqs (2.9) and (2.16), Eqs (3.25) and (3.26) allows us to determine H, S, V, and chemical potentials for the solution model with one internal parameter given by Eq. (3.23). In order to determine other properties we need second derivatives of the Gibbs energy. However, because we already have the first derivatives, we could use them in order to numerically determine the second derivatives. That is, we have for the heat capacity

$$C_p = (\partial H_m/\partial T)_{num} \qquad (3.27)$$

where the subscript *num* shows that this derivative is taken numerically. As a result, we have to find numerically the first derivative of the property, computed from the Gibbs energy according to (3.25) and (3.26). This improves the precision if as compared with the numerical determination of the second derivative of the Gibbs energy directly. The step here is controlled by the global option

```
<globals
  IntVarStep=6.05545e-05>
</globals>
```

The similar equations can be written for thermal coefficients

$$\partial^2 G_m/(\partial T \partial p) = (\partial V_m/\partial T)_{num} \qquad (3.28)$$

$$\partial^2 G_m/\partial p^2 = (\partial V_m/\partial p)_{num} \qquad (3.29)$$

and for derivatives which are necessary to compute the partial enthalpies, entropies, and volumes, for example

$$\partial^2 G_m/(\partial T \partial x_i) = -(\partial S_m/\partial x_i)_{num} \tag{3.30}$$

The partial heat capacities and partial thermal coefficients for the solution models with internal parameters are not supported in the current release of TDLIB, because these are the third derivatives of the Gibbs energy and, at the same time, the number of cases when they are needed is very low.

Conventionally, Eq. (3.24) to find the internal parameter is obtained as follows

$$\partial G_{mix}/\partial y = F(y;\, T, p, \boldsymbol{x}) = 0 \tag{3.31}$$

Then the equations above can be simplified. The first derivatives of the Gibbs energy (Eq. 3.25) becomes

$$\partial G_m/\partial z = \partial G_{ref}/\partial z + \partial G_{mix}/\partial z \tag{3.32}$$

where z is T, p, or x_i. This simplifies the computation of the H, S, V, and chemical potentials and hence allows us to change Eqs. (3.27) to (3.30) for the second derivatives of the Gibbs energy to the next ones

$$C_p = \partial H_{ref}/\partial T + \partial H_{mix}/\partial T + (\partial H_{mix}/\partial y)(\partial y/\partial T) \tag{3.33}$$

$$\partial V_m/\partial T = \partial V_{ref}/\partial T + \partial V_{mix}/\partial T + (\partial V_{mix}/\partial y)(\partial y/\partial T) \tag{3.34}$$

$$\partial V_m/\partial p = \partial V_{ref}/\partial p + \partial V_{mix}/\partial p + (\partial V_{mix}/\partial y)(\partial y/\partial p) \tag{3.35}$$

$$\partial S_m/\partial x_i = \partial S_{ref}/\partial x_i + \partial S_{mix}/\partial x_i + (\partial S_{mix}/\partial y)(\partial y/\partial x_i) \tag{3.36}$$

When we have several independent internal parameters (Eq. 3.22), we have set of equation to determine them

$$\mathbf{F}(\boldsymbol{y};\, T, p, \boldsymbol{x}) = 0 \tag{3.37}$$

where $\mathbf{F}$ is a vector of equations and $\boldsymbol{y}$ is a vector of unknown parameters. Then Eq (3.25) is to be changed to

$$\partial G_m/\partial z = \partial G_{ref}/\partial z + \partial G_{mix}/\partial z + \Sigma_i\, (\partial G_{mix}/\partial y_i)(\partial y_i/\partial z) \tag{3.38}$$

Here, in order to determine the set of the derivatives $\partial y_i/\partial z$ one has to solve a linear system of equations analogous to Eq. (2.14).

If the equations to determine unknown parameters are obtained by equating the derivatives of G_{mix} over y_i to zero

$$\partial G_{mix}/\partial y_i = F_i(\boldsymbol{y};\, T, p, \boldsymbol{x}) = 0 \tag{3.39}$$

then, as in the case with one internal parameter, Eq. (3.38) simplifies to Eq. (3.32), and Eqs (3.27) to (3.30) to determine the second derivatives of the Gibbs energy will become

$$C_p = \partial H_{ref}/\partial T + \partial H_{mix}/\partial T + \Sigma_i\, (\partial H_{mix}/\partial y_i)(\partial y_i/\partial T) \tag{3.40}$$

$$\partial V_m/\partial T = \partial V_{ref}/\partial T + \partial V_{mix}/\partial T + \Sigma_i\, (\partial V_{mix}/\partial y_i)(\partial y_i/\partial T) \tag{3.41}$$

$$\partial V_m/\partial p = \partial V_{ref}/\partial p + \partial V_{mix}/\partial p + \Sigma_i\, (\partial V_{mix}/\partial y_i)(\partial y_i/\partial p) \tag{3.42}$$

$$\partial S_m/\partial x_j = \partial S_{ref}/\partial x_i + \partial S_{mix}/\partial x_i + \Sigma_i\, (\partial S_{mix}/\partial y_i)(\partial y_i/\partial x_j) \tag{3.43}$$

TDLIB supports both cases to determine internal parameters, the conventional (Eqs 3.31 and 3.39), and general, when equations to determine unknown parameters (Eqs 3.24 and 3.27) are considered to be arbitrary and not to be equal to $\partial G_{mix}/\partial y_i$. The latter seems to be unusual, but in my view, it still might be usefull.

Phases with the internal parameters might produce the debugging information, because in order to compute the Gibbs energy and the derivatives one first has to solve Eqs (3.24) and (3.27) numericaly. This feature can be turned on for a particular phase by means of the attribute `debug` in the XML element for this phase. However, if you would like to turn it on for all the phases at once, set the global option `DebugPhase` to 1, as follows

```
<globals
  DebugPhase=1>
</globals>
```

3.3.1. Object CuOx_plane (tdlib/ex/phase/y123)

The `CuOx_plane` object represents a special case of the lattice model developed by Degterov and Voronin to describe the Gibbs energy of the $YBa_2Cu_3O_{6+z}$ (Y123) phase [11]. It can also be employed for other superconductors similar to Y123, that is, containing the basal plane CuO. According to the model, the Gibbs energy of the reaction

$$0.5Y_2O_3 + 2BaO + 3CuO + (0.5z\text{-}0.25)O_2 = YBa_2Cu_3O_{6+z} \tag{3.44}$$

can be expressed as follows

$$\begin{aligned}\Delta_{ox}G(T, z, x) &= g_1(T, p) + g_2(T, p)z + z(1 - z)\Sigma_i a_i(T, p)(1 - z)^{i-1} \\ &\quad + (c^2 - x^2)\Sigma_i b_i(T, p)(1 - z)^{i-1} \\ &\quad + RT[(c + x)\ln(c + x) + (c - x)\ln(c - x) + (1 - c + x)\ln(1 - c + x) \\ &\quad + (1 - c - x)\ln(1 - c - x) + z\ln z + (1 - z)\ln(1 - z)]\end{aligned} \tag{3.45}$$

where z is the stoichiometric index ($0 \le z \le 1$), c is short for $z/2$, x is the order parameter ($0 \le x \le z/2$), $g_i(T, p)$, $a_i(T, p)$ and $b_i(T, p)$ are some functions in temperature and pressure. The notation in Eq. (3.44) is the same as in Ref. [11], and it differs a little bit from the accepted in the present document. Hope that this will not bring forth a lot of confusion.

The order parameter, x, is the internal parameter of the model. That is, for all the thermodynamic functions at equilibrium, we have just three independent variables, the temperature T, the pressure p, and the index z, because this phase is a binary solution. The value of x at any given T, p and z can be determined by minimizing Eq. (3.45) over x. This means that according to the notation above this model falls to the conventional solution models with one internal parameter (Eqs 3.31 to 3.36). Then substituting the equilibrium value of x back to Eq. (3.45) we receive the equilibrium Gibbs energy

$$\Delta_{ox}G(T, p, z) = \Delta_{ox}G\{T, p, z, x_{eq}(T, p, z)\} \tag{3.46}$$

This means, that even though we have Eq. (3.12) in the closed form, the computation of the equilibrium Gibbs energy and other thermodynamic properties is possible by numerical methods only. More details, including the computation of derivatives (Eqs. 2.9 to 2.16), are given in Ref. [11].

The XML representation of the `CuOx_plane` object, shown in the file `y123.mod` in the directory `tdlib/ex/phase/y123`, imitates Eqs (3.22) and (3.45). G_{ref} in Eq. (3.22) is presented by the `Reference` object and then goes G_{mix} modeled by Eq. (3.45). As in the case of interaction objects, the imitation here is achieved by modeling each term in Eq. (3.45) by means of pairs of XML elements, `func_x`, a text label to describe the basis function in Eq. (3.45), and `func_Tp` to represent the function in temperature and pressure. The user can put in any number of terms with different types of `func_Tp` objects. On input, terms can follow in any order.

File `t.out.mod` allows us to print various properties of the `CuOx_plane` object, for example the command

```
assess y123 alg t.out -o prop
```

will write them to the file `prop.t`. Files `x1.out.mod` to `x3.out.mod` print smaller number of values that could be fitted in the screen. The next commands will compare the derivatives of the Gibbs energy computed by means of Eqs (3.32) to (3.36) with those computed numerically by means of `NumericalDerivatives` object.

```
assess y123 alg x1.out
assess y123 algnum x1.out
assess y123 alg x2.out
assess y123 algnum x2.out
assess y123 alg x2.out
assess y123 algnum x2.out
```

There should be no difference except for the partial entropies and Gibbs energies when mole fraction is equal to zero and one. The explanation for this difference is tied with Eq. (3.9) employed by TDLIB by default.

3.3.2. Objects AssociatedSolution and associated_solution (tdlib/ex/phase/ass)

The objects `AssociatedSolution` and `associated_solution` represent the generalized polynomial association solution model. There are some problems with both of them. The former is the new implementation, and the latter is left from the TDLIB'99. The `AssociatedSolution` object is more general but however it is not 100% numerically stable if the number of associates is more than one. The `associated_solution` object employs the VCS subroutine, which code is published in [7], as the computational engine. It is much more numerically stable but it is limited in functionality and it is distributed in the binary form of `access` only because Prof. W.R. Smith does not want me to distribute the code of the VCS subroutine.

I will start with some underlying philosophy, and then will give the examples and the more detailed description of the objects.

The Gibbs energy of the associated solution is conventionally considered to have a form of Eq. (3.2). The difference is that the rank of the formula matrix (Eq. 2.1), C, is smaller then number of species, N, some of which in this case are referred to as associates. This means that the chemical reactions are taking place among species. The number of linear independent chemical reactions is given by the next equation [7]

$$R = N - C \tag{3.47}$$

Let us denote the mole fractions of species as y_i, $i = 1, ..., N$. Then the original molar Gibbs energy of the association solution can be written as

$$\mathrm{G_m}(T, p, y_1, \dots, y_N) \tag{3.48}$$

At equilibrium, we can describe the molar Gibbs energy of the solution by means of C independent components. Let us denote the mole fractions of components as x_k, $k = 1, ..., C$. Hence, we can consider the solution either as containing C independent components, or as containing N species. Because this is the same solution, we can equate these two Gibbs energies

$$\mathrm{G_m}(T, p, x_1, \dots, x_C) = n\, \mathrm{G_m}(T, p, y_{1,eq}, \dots, y_{N,eq}) \tag{3.49}$$

However, in the general case, one mole of C components is different from one mole of N species, so we should multiply Eq. (3.48) by the total number of moles of the species, $n = \Sigma_i\, n_i$, which are situated within the one mole of the solution formed by C components. Sometimes, it is more convenient to switch from the mole fractions of species, y_i, to the numbers of moles, $n_i = n\, y_i$, and as a result we can write that

$$\mathrm{G_m}(T, p, x_1, \dots, x_C) = \mathrm{G}(T, p, n_{1,eq}, \dots, n_{N,eq}) \tag{3.50}$$

The equilibrium numbers of moles in Eq. (3.50) can be found from the equilibrium criterion (see, for example, Ref. [7]), which lead us to the system of N non-linear equations as follows

$$x_k = \Sigma_i\, a_{ki}\, n_i \qquad (3.51)$$

$$\Sigma_i\, \nu_{ij}\, \mu_i = 0 \qquad (3.52)$$

The first C equations (Eq. 3.51, $k = 1, ..., C$) show the material balance between the independent components and the species. a_{ki} is the element of the formula matrix in which the species are composed from the independent components. The last R equations (Eq. 3.52, $j = 1, ..., R$) state that at equilibrium the change in chemical potentials should be zero for each chemical reactions. ν_{ij} is the element of the stoichiometric matrix, and it shows the stoichiometric number of the i-th species in the j-th chemical reaction.

Again, Eqs (3.50) - (3.52) should be considered as algorithm to compute the Gibbs energy of the association solution. First, for given T, p, x_1, …, x_C one has to solve the system of equations (3.51) - (3.52) for $n_{i,eq}$, taking into account that $n_i \geq 0$, and then it is necessary to substitute the values found into Eq. (3.50).

We have the special case of the ideal association solution when the excess Gibbs energy is equal to zero

$$\mathrm{G_{excess}} = 0 \qquad (3.53)$$

in other words, when the chemical potentials of the species can be expressed as

$$\mu_i = \mu_i^o + \mathrm{R}T \ln y_i \qquad (3.54)$$

This case has very good mathematical properties because here the Gibbs energies (3.48) and (3.50) are convex, and one can prove that there is one and only one solution of the system of equations (3.51) - (3.52) within the admissible range $n_i \geq 0$. In addition, there are good algorithms available which can solve the system for all the possible cases with the automatic generation of the initial guess. As for me, for this case I really like the VCS subroutine from the Ref. [7].

In the general case of the polynomial association solution the situation is much worse. It is always possible to solve the system of equations (3.51) - (3.52) for a particular case, however I do not know the general algorithm, which could solve this system in any case with automatic generation of the initial guess with 100% guarantee of the success. One of the main problems here is that in the general case the system of equations (3.51) - (3.52) might have several solutions, each of them corresponding the local minimum within the admissible range $n_i \geq 0$, and, as a result, in this case it is necessary to switch to the problem of finding the global minimum.

At this point, let us ask themselves why it is necessary to solve the system (3.51) - (3.52). The typical answer is that we need the equilibrium composition of associates. Then, let us ask the next question, whether the associates have physical meaning, that is, what are the proves, that they exist in the solution. Here, the answers vary. Some researches are confident that the associates do exist and they even try to determine their chemical composition by processing the experimental values under the either ideal of regular association model. Others just say that the association model takes into account ordering, and they are not that sure that the particular set of associates chosen to fit the experimental values has real prototypes in the solution.

Personally, I share rather the extreme view that the introduced associates are just mathematical tricks to develop more suitable model for multicomponent solution. Well, associates do have some relationship with the real interactions within the solution, but I would say that the associates just some shadows of real interactions.

From the viewpoint of mathematical description, it is roughly possible to compare the associated solution model with the splines. The idea is about the same - to divide the area to the small pieces in order to decrease the power of the polynomial, and along this way, the associates play the role of the spline nodes. If we accept this point of view, then we do not have to compute the equilibrium composition of associates at all. If I say that associates do not have the physical meaning, I do not have to interpret $y_{i,eq}$ as equilibrium mole fractions, even though this was the initial way of thinking.

As a result, the idea behind the new generalization of the polynomial associated solution model is to modify the original system of equations (3.51) - (3.52) in order to simplify the determination of its roots, that is, to change Eq. (3.52) to the next one

$$\Sigma_i \; \nu_{ij} \; \mu_i^{id} = 0 \qquad (3.55)$$

where the chemical potential is assumed to have a form of Eq. (3.54). Hence, the values of $y_{i,eq}$ are found as if the solution would be the ideal associated one. As was mentioned above, this task can be considered as good and one can expect no numerical problems here.

Once again, in this case the associates are not considered as meaningful physical entities but rather a mathematical notation, and their mole fractions, $y_{i,eq}$, found as the solution of the system of equation (3.51) and (3.55), are not considered as equilibrium ones. Now this is just a mathematical trick to develop a spline-like model. The background idea here is to change Alan Oates phrase “Let us put more physics to the CALPHAD models” to the “Let us put more mathematics to the CALPHAD models”.

In order to make it possible to elaborate this idea, TDLIB should support the two implementations of the association model, the conventional one based on the system of equations (3.51) and (3.52) and the new one, based on the system of equations (3.51) and (3.55).

As was mentioned above, the `associated_solution` object was taken from TDLIB’99. It relies just on the VCS subroutine, and as such, it can handle only the new model. Let us discuss determining the derivatives in this case. It is possible to derive the expressions for the first derivatives of the molar Gibbs energy (3.50) as follows (see Eqs 2.11 and 3.38).

$$(\partial \mathrm{G_m}/\partial T)_{p,\mathbf{x}} = (\partial \mathrm{G}/\partial T)_{p,\mathbf{x},\mathbf{n}} + \Sigma_i \; \mu_i \; (\partial n_i/\partial T)_{p,\mathbf{x}} \qquad (3.56)$$

$$(\partial \mathrm{G_m}/\partial p)_{T,\mathbf{x}} = (\partial \mathrm{G}/\partial p)_{T,\mathbf{x},\mathbf{n}} + \Sigma_i \; \mu_i \; (\partial n_i/\partial p)_{T,\mathbf{x}} \qquad (3.57)$$

$$(\partial \mathrm{G_m}/\partial x_k)_{T,p,x(l\neq k)} = (\partial \mathrm{G}/x_k)_{T,p,x(l\neq k),\mathbf{n}} + \Sigma_i \; \mu_i \; (\partial n_i/\partial x_k)_{T,p,x(l\neq k)} \qquad (3.58)$$

where $(\partial n_i/\partial T)_{p,\mathbf{x}}$, $(\partial n_i/\partial p)_{T,\mathbf{x}}$, and $(\partial n_i/\partial x_k)_{T,p,x(l\neq k)}$ can be found as a solution of the linear systems of equations (3.59) - (3.60), (3.61) - (3.62) and (3.63) - (3.65) accordingly (see Eq. 2.14)

$$\Sigma_i \; a_{ki} \; (\partial n_i/\partial T) = 0 \qquad (3.59)$$

$$\Sigma_i \; (\nu_{ij}/n_i - \Delta\nu_j/n)(\partial n_i/\partial T) = \{\Sigma_i \; \nu_{ij}(S_i^o - \mathrm{R}T \ln y_i)\}/(\mathrm{R}T) \qquad (3.60)$$

$$\Sigma_i \; a_{ki} \; (\partial n_i/\partial p) = 0 \qquad (3.61)$$

$$\Sigma_i \; (\nu_{ij}/n_i - \Delta\nu_j/n)(\partial n_i/\partial p) = -(\Sigma_i \; \nu_{ij} \; V_i^o)/(\mathrm{R}T) \qquad (3.62)$$

$$\Sigma_i \; a_{ki} \; (\partial n_i/\partial x_l) = 1 \quad (k = l) \qquad (3.63)$$

$$\Sigma_i \; a_{ki} \; (\partial n_i/\partial x_l) = 0 \quad (k \neq l) \qquad (3.64)$$

$$\Sigma_i \; (\nu_{ij}/n_i - \Delta\nu_j/n)(\partial n_i/\partial x_k) = 0 \qquad (3.65)$$

The second derivatives of the Gibbs energy can be found numerically (see Eqs. 3.27 – 3.30).

It is useful to extract from the chemical potential of the species the part tied with the independent components as follows

$$\mu_i = \Sigma_k \; a_{ki} \; \mathrm{G}^*_{\mathrm{m},k}(T, p) + \Delta\mu_i \qquad (3.66)$$

As a result the Gibbs energy of the associated solution can be written as

$$\mathrm{G}(T, p, n_1, \ldots, n_N) = \Sigma_i \; \mu_i \; n_i = \Sigma_k \; x_k \; \mathrm{G}^*_{\mathrm{m},k}(T, p) + \Sigma_i \; \Delta\mu_i \; n_i \qquad (3.67)$$

The first part corresponds to the G_{ref} in Eq. (3.22) and the second to G_{mix}. The format for the `associated_solution` object follows Eq. (3.67) and (3.22) when the `Reference` object models the first part and the `SimpleSolution` object surrounded by the `internal_solution` tags – the second one. The examples are in the `tdlib/ex/phase/ass` directory.

The `AssociatedSolution` object is a more general replacement for the `associated_solution` object. First, with the difference with all the previous TDLIB objects the XML element `AssociatedSolution` is tied with several background C++ classes. The idea is that at the beginning the object is handled by the loader that determines the most appropriate numerical engine to use and gives the control to it. The two circumstances are taken into account in the present release: the number of independent reactions and the value of the `equilibrated` attribute. The latter controls the use of either the conventional model (Eqs. 3.51 and 3.52, if equal to one) or the new model (Eqs. 3.51 and 3.55, if equal to zero). For the ideal association solution the results should not depend on the value of this attribute. The number of independent reactions is determined by analyzing the formula matrix. If it is equal to one then the employed solver works very well. The only problem here might be the case when `equilibrated=1` and there are several local minima. Then which minimum will be found depends on chance. The solver for the case with several independent reactions has not been worked out to the final stage yet. You have to expect to encounter the case when the convergence will not be achieved.

The second generalization is the opportunity to use the `SimpleSolution` object both for the G_{ref} and G_{mix} in Eq. (3.22) and (3.67). This gives the possibility to simultaneously employ interaction terms based on the internal mole fractions, y_i, for the species in the internal solution and on the mole fractions of the independent components, x_i. It means a great deal of flexibility within the same implementation.

The third difference is tied with using the chemical variables as the internal ones. In this case the Eq. (3.51) is solved explicitly as follows (see Ref. [7] for details)

$$n_i = n_{i,o} + \Sigma_j \ \nu_{ij} \ \xi_j \qquad (3.68)$$

where we have R independent chemical variables ξ_j as a replacement for the N linear dependent number of moles of the species in the internal solution. It simplifies both the determining the solution of Eq. (3.52) and the derivatives similar to Eqs. (3.56) to (3.58).

The behavior of all the `AssociatedSolution` objects is controlled by a few global options, listed below.

```
<globals
  AssFactr=1000
  AssM=10
  AssPgtol=1.49012e-08
  AssTolerance=2.22045e-14>
</globals>
```

The first three of them set the parameters for the subroutine LBFGSB employed as the numerical engine within this object in the case when the number of independent reaction is more than one. There description can be found within the file `tdlib/lib/toms/lbfgsb.f`. The `AssTolerance` option controls the minimum length of the admissible range for the chemical variable. If its length is less than `AssTolerance`, than this chemical variable is considered to be fixed.

Finally, it should be mentioned that there is some sense to use associates A_pB_q in the form $A_{p/(p+q)}B_{q/(p+q)}$ even this is not required by TDLIB. Some arguments can be found elsewhere in Ref. [25, 26].

In the directory `tdlib/ex/phase/ass` there are files demonstrating `AssociatedSolution` and `associated_solution` objects. The `ideal.mod` describes the simple ideal association solution with one associated complex, the interactions are added in the `reg.mod`. The `reg2.mod` defines the regular association solution with two associates. The more sophisticated case is in the file `bise.mod`. It describes the ideal associated gas in the Bi-Se vapors. The files with the suffix `old` implement the analogous cases by means of the `associated_solution` object. The files from `x.out.mod` to `x3.out.mod` define the output of different properties.

The commands as follows

```
assess reg alg x1.out
assess reg algnum x1.out
```

will produce the output of the properties computed by the object itself and by numerical differentiation accordingly. You could see that the `associated_solution` object is working pretty good (it does not support partial enthalpies and entropies though) as well as the `AssociatedSolution` object in the case of a single independent chemical reaction. When the number of chemical reactions is more than one (files `reg2.mod` and `bise.mod`), than the `AssociatedSolution` object gives less reasonable results. First, there are some problems with convergence, second, there is a little bug in derivatives to compute the chemical potentials. It is necessary to put in it some more efforts.

In the directory `tdlib/ex/phase/ass/compare` there are files to compare the conventional and new association models. In the file `ass.mod` there is a case with one associated complex when on the right side there is a positive interaction. This is sometimes useful in order in order to deal with the systems when there is a strong negative interaction in the liquid and at the same time there is a miscibility gap on either side of the associated complex. The next commands

```
assess ass out -v ini -o t
gnuplot t.g -
gnuplot t.int -
```

will plot the molar Gibbs energy with the chemical potentials and the internal mole fractions for both models. Then it is possible to increase the value of the parameter `mis` in the file `ini.par` and to see what happens. Note that the internal mole fractions for the new model do not depend on the value of the interaction parameter even though the Gibbs energy and chemical potentials do. When `mis` is greater than 100000 than there should be irregular behavior in both graphs for the conventional model. The reason is that here there are two solutions within the admissible range and the `AssociatedSolution` object takes either from these two possible solutions by chance. There is no such a problem for the new model because for it there is only one solution in the admissible range for all the possible values of the model parameters.

3.4. Describing phase equilibria. Objects in the TD_ALGO library

The main goal of the TD_ALGO library is to support the framework in order to develop thermodynamic algorithms. As such, there is a category `algorithm` in which a user can add new objects to implement new algorithms. The objects from the `algorithm` category communicate with the upper objects by means of the `compute` object, which is considered to be an intermediary layer with the goal to adjust the flow of the information between the upper object and the `algorithm`.

The `algorithm` could have its own state, and in the inverse problem when the optimizer changes the values of unknown parameters, it is necessary to disclose this to all the algorithms. In order to make it possible, in the current release of TDLIB all the algorithms are required to have the attribute `id` - anonymous algorithms are forbidden. All the algorithms could also have the attribute `debug`. If it is equal to 1, then the algorithm might produce some information about its work. In order to turn on debugging at once for all the algorithms, set the value of `DebugAlgorithm` in the `globals` object to 1.

Another polymorphous category introduced in the TD_ALGO library is `output`. It is designed to describe the output in the abstract form of the two-dimensional array of double values when each column of this array is supposed to have a textual name.

One of the purposes of algorithms is to produce output. The overall relationship between the objects in this case is as follows. The `OutputFile` object contains a vector of `ComputeOutput` objects from the `output` category. Each `ComputeOutput` object contains a vector of `compute` objects, which in turn contain the objects from the `algorithm` category.

Let us see first, how it works with the example of the dummy `PassThrough` algorithm. Then the others, more meaningful algorithms, `PhaseProperty`, `PhaseEquilibrium` and `reaction` will be presented.

3.4.1. Object PassThrough (tdlib/ex/algo)

As was mentioned earlier, the algorithm is considered to be a black box that takes a set of input values, **x** and after the processing produces a set of output values, **y**. Each value in both input and output sets has a textual name by which it can be accessed by the upper object.

The `PassThrough` is a dummy algorithm that just passes the input values to the output without any treatment. It can take any name on input and then it forms the value with the same name in the output set.

The file `pass1.mod` in the directory `tdlib/ex/algo` demonstrate the simplest use of the `PassThrough` object. The `compute` object sets two input values within the `PassThrough` object and then just retrieves them. The file `pass2.mod` shows how to organize a loop within the `ComputeOutput` object, and finally the file `pass3.mod` demonstrate the conversion of the output values obtained by the algorithm by means of the convert object. Run

```
assess pass1
assess pass2
assess pass3
```

to see the results.

3.4.2. Object OutputFile

The `OutputFile` object is a simple container for the objects from the `output` category. It takes two-dimensional arrays of values from each `output` object and prints them to the output file with an extension specified by the attribute `ext`. There is an attribute `format` that controls the way, how the values are printed. By default, `format = file` that means that `OutputFile` prints first the names of the columns for the first `output` object followed by its two-dimensional array, then an empty line, and after that repeats this for all the `output` objects that it contains. If `format = axum` then all the columns with their names from all the `output` objects are printed side by side. If the length of some column is less than the others the non-existent values will be substituted by the text `miss`. Such a file can be easily imported to Axum for plotting. I would expect that other plotting packages ate this format either.

The third option, `format = gnuplot` leads to the output file, which can be read directly by Gnuplot to plot a two-dimensional graph. Here the `OutputFile` object implies some additional information implicitly. The rules are as follows. All the `output` objects within the `OutputFile` object are plotted in the same graph but each of them is processed independently. The first column of the `output` object is treated as abscissa, all others as ordinates. The attribute `ChangeXY` in the `output` object changes abscissa and ordinate between each other for this `output` object.

3.4.3. Object convert

Sometimes it is necessary to transform the set of values. To this end, the XML element `convert` has been developed. It implements an arbitrary transformation function in several variables, f(x_1, ..., x_N), evaluated on the fly by means of interpreting. The element by itself is as follows

```
<convert name=name> expression </convert>
```

where within the expression one can use all the arithmetic operations (+, -, *, /, ^), functions (acos, abs, asin, atan, cos, exp, log10, log, sqrt, tan) and the names of values available in the current scope. The scope depends on the place where the `convert` object resides. The attribute `name` sets the name of the output value; by default it is empty. If a name of some x_i contains parentheses and/or comma, then it is necessary to surround it by the `<str></str>` tags.

3.4.4. Object ComputeOutput (tdlib/ex/algo)

This is the only object from the `output` category in the TD_ALGO library. Some others from this category will appear in the VARCOMP library.

In its simplest form it is just a container for `compute` objects (see file `pass1.mod`). Here the `compute` object on its own supplies all the information to the algorithm.

The alternative is to make a loop (see file `pass2.mod`). Here the `start` element sets a set of values as well as their names. All these values will be available on input for the `compute` objects. The `finish` element defines the stopping criteria. The allowable comparison operations are `GE` (greater or equal), `EQ` (equal), and `LE` (less or equal). The `step` element defines the actions to increment the values. Here the `convert` element can take all the names defined in the `start` element and assign the result to any value.

In order to define the starting values and the stopping criteria, the `start` and `finish` elements in the `ComputeOutput` object might also have a `compute` object. It allows us to dynamically change the range in which values will be changed within the loop. It is important for drawing phase diagrams when it is necessary first to compute non-variant points and then to connect them between each other by mono-variant lines. The examples of these approaches can be found in the files `tdlib/ex/bacu/pd.out.mod` and `tdlib/ex/bise/pd.out.mod`.

3.4.5. Object compute

The `compute` object helps communicate upper objects (in the current release, these are `ComputeOutput` and `residual`) with the objects from the `algorithm` category. It takes on input a set of values from the upper object and sets the required values within the `algorithm` object. Along this way the `compute` object can translate names and convert the values to those required by the `algorithm` object. Also, the `compute` object prepares the set of output values to return to the upper object where it also could change the names of the values and make the conversions required. For example, the next object

```
<compute>
  <algorithm class=algorithm IDREF=s11_L></algorithm>
  <input name=T> T </input>
  <input name=x2> x(L,Se) </input>
  <output> T </output>
</compute>
```

defines the communication with the `s11_L` algorithm. It says that on input the values of `T` and `x2` of the upper object should be set as the values of `T` and `x(L,Se)` of the algorithm accordingly. On output it is necessary to obtain the resultant value of `T` with no changes.

Formally, the body of the `compute` object should contain the object from the `algorithm` category and then a list of XML elements `input` and `output`.

Each XML element `input` contains the value in the `algorithm` to be set. The XML element `input` can have either `name` or `value` attribute. The attribute `name` defines the name of the value from the upper object to be taken. The attribute `value` allows the user to set the value of the algorithm to the given number. If the `input` element does not have neither attribute `name` nor `value`, it expects the `convert` object in the body, for example,

```
<input><convert>t+273.15</convert> T </input>
```

This means the value of `T` within the `algorithm` object should be equal to the given expression where the upper object should supply the value of `t`.

The XML element `input` also can have attribute `once`, for example

```
<input value=500 once> T </input>
```

This means that this element will be applied to just once during the interaction of the upper object and the `algorithm`. For example, if this is encountered in the `compute` object within the loop of the `ComputeOutput` object, it will be applied just before the start of the loop.

The SGML element `output` contains the values to retrieve and can have the attribute `name`, to translate the name, if necessary. By default, the `name` will be equal to the `name` of value from the `algorithm`. The `algorithm` might produce several values for a single `output` element but this is algorithm dependent.

It is possible to put `convert` objects after the `output` elements. For example,

```
<compute>
  <algorithm class=algorithm IDREF=L></algorithm>
  <input name=T> T </input>
  <input name=x2> x(Te) </input>
  <output> H_mix </output>
  <output name=x1> x(Bi) </output>
  <output name=x2> x(Te) </output>
  <convert> H_mix/x1/x2 </convert>
</compute>
```

Here all `output` objects form a set of values that is considered as input for the `covert` elements. It is possible to place several `covert` elements in order to form a new set of output values. However, the rule is that if any `convert` object is present within the `compute` object, then its set of output values is formed by `convert` elements only. This means that in the example above the output of the compute object consists from only single entity.

3.4.6. Object PhaseProperty (tdlib/ex/phase)

The purpose of the `PhaseProperty` object is to provide access to the properties of the `phase` object. The examples of the `PhaseProperty` objects can be found in the directory `tdlib/ex/phase` where they were used to print the phase properties. Its body contains just the `phase` object to work with. It has the attribute `DependentMoleFraction` to specify what mole fraction should be computed from the $\Sigma_i x_i = 1$ constraint. This attribute could be equal to `first` (default), `last`, `no` or the name of the component.

On input from the `compute` object, the `PhaseProperty` object allows us to specify `T`, `p`, and `x(`*`component_formula`*`)`. On output it support a variety of values, shown in the next table.

Description	Text string
Temperature	`T`
Pressure	`p`
Mole fraction	`x(`*`component_formula`*`)`
All state variables (`T`, `p`, and mole fractions)	`state(all)`
Thermodynamic molar property (below any property is specified as *`Z`*)	`G, H, S, V, Cp, dVdT, dVdp`
Property modifier (in the form *`Z_modifier`*)	`full` (default), `ref`, `mix`, `ideal`, `excess`
Partial property	`Z(`*`component_formula`*`)`
Partial properties for all components	`Z(all)`
Value of internal variable	`internal(`*`variable_name`*`)`
Values of all internal variables	`internal(all)`
The stability of all the internal phases	`stable(all)`
The stability of the internal phases	`stable(`*`phase_name`*`)`

What properties does a particular solution model support and what is the meaning of the property modifiers depends on the implementation.

3.4.7. Object PhaseEquilibrium (tdlib/ex/bise)

In the current release of TDLIB, the `PhaseEquilibrium` object is the most advanced algorithm designed to compute the phase equilibrium. It contains the list of phases, which should exist at the equilibrium, and the division of the external variables to the groups according to the Gibbs phase rule, Eq. (2.28).

Let us consider an example of the Bi-Se system (directory tdlib/ex/bise). There are six phases defined in the file `sys.mod`: `L` (a binary Bi-Se melt), `s01` (solid Se), `s10` (solid Bi), `s11` (a compound Bi0.5Se0.5 with a wide homogeneity range), `s23` (solid Bi0.4Se0.6), and `s32` (solid Bi0.6Se0.4). The liquid `L` has a miscibility gap on the right from the `s23` phase. This example is taken from Ref [30].

The file `alg.mod` defines the phase equilibria that are necessary to compute the phase diagram and they are employed in the file `pd.out.mod` to plot the phase diagram. Run the next commands to obtain the plot

```
assess sys alg pd.out -o t
gnuplot p.td -
```

The file `pd.out.mod` contains the information, which could be called as the phase diagram topology. In principle in the direct problems, the files analogous to `alg.mod` and `pd.out.mod` could be generated automatically, because the molar Gibbs energies of the phases (file `sys.mod`) contain all the information to build the phase diagram. Unfortunately, in the current release of TDLIB, this functionality is not available.

I use TDLIB primarily for the inverse problems, when it is necessary to find the unknown parameters within the Gibbs energy by means of fitting the experimental values. Here the only way to create the files kind of `alg.mod` and `pd.out.mod` is by hand because if you take the file `sys.mod` with an arbitrary values in the Gibbs energies than it would produce the phase diagram with quite a different topology than required.

Let us start with two-phase equilibria. In this system, they describe the mono-variant lines. The simplest are the liquidus lines between point phases and the melt. For example, in order to describe the liquidus between s23 and L, one can define

```
<PhaseEquilibrium class=algorithm id=s23_L>
  <phases>
```

```
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=s23></phase>
  </phases>
  <state status=dependent> x(L,Bi) </state>
  <state status=constraint> x(L,Se) </state>
  <state status=unknown> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

The object above contains the element `phases` with the list of phases and a list of the `state` elements. The attribute `status` within the `state` element describes the status of external variables. Within the `PhaseEquilibrium` object it is possible to use temperature (`T`), pressure (`p`), and the mole fractions in the form `x(`*`phase_id`*`,`*`component_formula`*`)`.

The status `dependent` means that this mole fraction should be computed according to the constraint $\Sigma_i x_i = 1$. The status `unknown` declares that this variable is unknown in the system of equations (2.26). The values of variables declared as `constraint` or `HardConstraint` are assumed to be set before the beginning of the numerical solution of Eq. (2.26). Thus, the number of `unknown` elements is equal to the number of equation in (2.26), which is given by Eq. (2.24). The total number of `constraint` and `HardConstraint` elements is given by Eq. (2.28). In our case, we have one equation to be solved and thus we should set two other variables in this equilibrium.

The difference between the `constraint` and `HardConstraint` variables is tied with the ability of the other objects to change their values. The value of the `HardConstraint` variable is set in the `PhaseEquilibrium` object and could not be changed by other objects. On the other side, the value of `constraint` is assumed to be set by the other objects, before they would like to obtain the solution of the system (2.26).

The object above reflects the fact, that in the system in question, this equilibrium is considered to be mono-variant, that is, there is a line at the phase diagram, describing the possible points {x(L,Se), T}, which all correspond to the given equilibrium. The object defined above allows the user to set the mole fraction of the melt and then to compute the liquidus temperature.

It is possible to define a different object for the same equilibrium

```
<PhaseEquilibrium class=algorithm id=s23_L_2>
  <phases>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=s23></phase>
  </phases>
  <state status=dependent> x(L,Bi) </state>
  <state status=unknown> x(L,Se) </state>
  <state status=constraint> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

when the user should set the temperature and then the object will compute the mole fraction of the melt.

The miscibility gap is also the two-phase equilibrium. It is possible to define as follows

```
<PhaseEquilibrium class=algorithm id=L1_L2>
  <phases>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=L></phase>
  </phases>
  <state status=dependent> x(L_1,Bi) </state>
  <state status=unknown> x(L_1,Se) </state>
  <state status=dependent> x(L_2,Bi) </state>
  <state status=unknown> x(L_2,Se) </state>
  <state status=constraint> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

Here again the pressure is considered as the constraint that can not be change in the other objects. The user should set the temperature and the object will try to compute two unknown mole fractions because in this case there are two equations in the system (2.26). Note that when a phase is listed several times in the `phases` element, it should be referred to as *`phaseid_phasenumber`*, for example, `L_1` refers to the first liquid, `L_2` to the second.

Three-phase equilibria in the two-component system are considered to be non-variant, that is, in this case there should be no variables to set. For example, the object

```
<PhaseEquilibrium class=algorithm id=s10_L_s32>
  <phases>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=s10></phase>
    <phase class=phase IDREF=s32></phase>
  </phases>
  <state status=dependent> x(L,Bi) </state>
  <state status=unknown> x(L,Se) </state>
  <state status=unknown> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

declares the three-phase equilibrium among `s10`, `s32`, and the melt. At the constant pressure there is just a point in the phase diagram, describing this equilibrium, and the above object allows the user to determine the temperature and the melt mole fraction for this point. Another example is the equilibrium among `s23` and two miscible liquids. Here it is possible to compute the non-variant point with the next object

```
<PhaseEquilibrium class=algorithm id=s23_L1_L2>
  <phases>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=s23></phase>
  </phases>
  <state status=dependent> x(L_1,Bi) </state>
  <state status=unknown> x(L_1,Se) </state>
  <state status=dependent> x(L_2,Bi) </state>
  <state status=unknown> x(L_2,Se) </state>
  <state status=unknown> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

The number of unknowns is more than in the previous example because here there are more equations in the system (2.26).

It is possible to declare the state variable as `functional`. This allows us to specify an expression in order to compute the value of this variable from the values of other state variables in the given `PhaseEquilibrium` object. In this respect, the status `dependent` can be considered as a special hard-programmed case of `functional` and in the example above instead of

```
<state status=dependent> x(L_2,Bi) </state>
```

one could write

```
<state status=functional>
  <convert> 1 - <str>x(L_2,Se)</str> </convert>
  x(L_2,Bi)
</state>
```

With this technique it is possible to compute the melting of the `s11` solid solution as follows

```
<PhaseEquilibrium class=algorithm id=Tms11>
  <phases>
    <phase class=phase IDREF=L></phase>
    <phase class=phase IDREF=s11></phase>
  </phases>
  <state status=dependent> x(L,Bi) </state>
  <state status=unknown> x(L,Se) </state>
  <state status=dependent> x(s11,Bi) </state>
  <state status=functional>
    <convert><str>x(L,Se)</str></convert>
    x(s11,Se)
  </state>
  <state status=unknown> T </state>
  <state status=HardConstraint value=1> p </state>
</PhaseEquilibrium>
```

where the `functional` status gives us a means to define that at melting `x(s11,Se) = x(L,Se)`. Note that in the case of the Bi-Se system this equilibrium is metastable for the `s11` phase has the peritectic decomposition at lower temperatures.

Another option is `status=value` that adds a value of some property of any phase from the `PhaseEquilibrium` object as an additional equation to Eq. (2.26). This allows us to put additional constraints and gives us more flexibility in setting up the system of equation to solve. The file `tdlib/ex/algo/prop.mod` demonstrates how by means of this feature to compute such a temperature when the entropy of the phase is equal to the given value. Run

```
assess prop
```

to see how it works.

Now let us pay attention to the numerical behavior of the `PhaseEquilibrium` object, which is based on the LBFGSB subroutine from the NETLIB/TOMS library. It well might be that the object will not compute the result required. Actually in the two objects considered above there is ambiguity that will give undesired results for sure. This is the main difference of the `PhaseEquilibrium` object with the objects in the PHASE library, when the goal was to deliver numerically reliable objects even in the case of the models with internal parameters. The main reason is that there are many cases when the solution of Eq. (2.26) does not exist in principle for a given set of phases with the current values of unknown parameters.

In order to enhance the chances for success, the user should set the bounds and initial estimates for the unknowns. To this end, one can use attributes `lower` and `upper` to set the bounds and `value` to set the initial estimate, for example

```
<state status=unknown lower=800 upper=950 value=880> T </state>
```

By default, the bounds for the temperature are PETmin and PETmax from the `globals` object and for the mole fraction the default bounds are 0 and 1. This causes ambiguity in the object with `s23_L_2`, defined above, when the melt mole fraction should be determined for the given temperature. The problem here is that there are two possible solutions from both sides of the `s23` phase. Run

```
assess sys s23_l
```

to see that in this case you can receive the solution from both sides in random.

Then, in order to make it clear what the user wants, it is necessary to use bounds, for example

```
<state status=unknown lower=0 upper=0.6> x(L,Se) </state>
```

to show that the solution from the left side is required. This is implemented in the file `s23_l2.mod` and you can compare the results of the command

```
assess sys s23_l2
```

with above.

The object `L1_L2` introduced above will also lead to undesired result. With high probability the user will receive the trivial solution, that is, `x(L_1,Se) = x(L_2,Se)`, which always satisfies the equilibrium equation (2.26) in this case. Then, in order to exclude this case it is necessary to use bounds. The files `mis.mod` and `mis2.out` implement the cases without and with bounds. Run

```
assess sys mis
assess sys mis2
```

and see the difference.

One could expect more intellectual numerical behavior of the `PhaseEquilibrium` object, however in order to achieve it is necessary to develop specialized thermodynamic algorithms because there are many different cases, which require a special treatment.

The object, presented in the current release, is designed mainly for the inverse problems when the topology of the phase diagram and the rough estimates of non-variant point are known, and when it is necessary to find the unknowns in the Gibbs energies, which will give the topology required. In this case, it should be not that difficult to supply the information on bounds and initial estimates to compute the phase equilibrium.

There are three different cases.

1) Initial estimates are set for all the unknowns.
2) Initial estimates are set just for a part of unknowns.
3) Initial estimates are not set at all.

Let us consider them in turn. Note that the bounds are always taken into account.

The first case is the most recommended. Here, the object runs the numerical subroutine LBFGSB to solve the system (2.26) with the initial estimates supplied by the user. If the numerical procedure converges, that's it. If not, then the object computes F_{min} (Eq. 2.29) for the initial user estimate and throws the exception with this value. This allows the upper object to employ F_{min} as a penalty if required (see Eq. 2.33).

Let us especially stress, that in the case of failure, F_{min} is computed for the initial estimate supplied by the user and not for the values returned by the numerical subroutine. This is very important for the inverse problems when the Eq. (2.26) is solved as part of the optimization procedure - determining unknown parameters in the vector Θ in Eq. (2.31). This means that in the case of failure the residual computed by Eq. (2.33) corresponds to the case of so called "indirect minimization", and overall behavior can be formulated as follows. When there is a bad initial guess for the vector Θ and some phase equilibria are not computed (Eq. 2.26 has not been solved for the given vector Θ), the optimization procedure (minimization of Eq. 2.34) will try to change the vector Θ by means of "indirect optimization". The goal of "indirect optimization" is to minimize the values of F_{min} (Eq. 2.33), that in turn means better description of the initial user estimates for equilibria. If all goes well, at some point the equilibria get eventually computed, and after that the optimization procedure will automatically switch to residuals (2.32), which correspond to the "direct minimization".

In the second case, the solution of Eq. (2.26) performed in two steps. First, the object tries to find the best values of variables for which the initial guesses have not been supplied. That is, it tries to minimize F_{min} (Eq. 2.29) when the values of variables for which the initial guesses have been given are fixed. Then the object releases all variables and try to solve Eq. (2.26). Again, if the numerical procedure converges, that's it. If not, then the object throws the exception with the value of F_{min} (Eq. 2.29) obtained in the first step.

In the third case, the object tries to automatically generate the initial guesses which are necessary for the solution of Eq. (2.26). In the case of failure to solve the system (2.26), the object throws the exception with the minimal value of F_{min} obtained during all the tries. I would not recommend this behavior for the inverse problems.

On input from the `compute` object, the `PhaseEquilibrium` object allows us to set the state variables specified as `constraint`, `value` or `unknown`. In the last case, it allows us to set the initial guess. In order to indicate that the initial guess for the unknown variable should be chosen automatically, it is necessary to set it to –1, for example:

```
<input value=-1> T </input>
```

On output the `PhaseEquilibrium` object support a variety of values, similar to those of the `PhaseProperty` object and shown in the next table.

Description	Text string
Temperature	`T`
Pressure	`p`
Mole fraction for the given phase	`x(`*`phase_id`*`,`*`component_formula`*`)`
All state variables (`T`, `p`, and mole fractions) for all the phases	`state(all)`
Thermodynamic molar property (below any property is specified as *`Z`*) for the given phase	`G(`*`phase_id`*`), H(`*`phase_id`*`), S(`*`phase_id`*`), V(`*`phase_id`*`), Cp(`*`phase_id`*`), dVdT(`*`phase_id`*`), dVdp(`*`phase_id`*`)`
Property modifier (in the form *`Z_modifier`*)	`full` (default), `ref`, `mix`, `ideal`, `excess`
Partial property for the given phase	`Z(`*`phase_id`*`,`*`component_formula`*`)`
Partial properties for all components in the given phase	`Z(`*`phase_id`*`,all)`
The value of Eq. (2.29)	`fmin`
All the final values for equations (2.26) (should be close to zero)	`fmin(all)`
The final value for particular equation in (2.26)	`fmin(`*`number`*`)`

The `PhaseEquilibrium` has its own state. The attribute `SaveSolution` controls whether the solution of Eq. (2.26) found should be saved in this state (by default, `SaveSolution = 0` and the current solution is not saved). It is possible to set `SaveSolution` on input from the `compute` object (see file `pd.out.mod`). After that, the solution from the previous computation is taken as the initial guess for the next one. If you set `debug=1`, you could see it in the debug output.

Another attribute of the `PhaseEquilibrium` object, `ThrowException` controls whether the exception discussed above will be thrown. By default, `ThrowException = 1`, and the exception is thrown. If `ThrowException=0`, the object stops throwing exceptions, and in the case of non-convergence just returns the values, which have no physical meaning.

There are global options for all the `PhaseEquilibrium` objects. They are listed below with their default values.

```
<globals
  PETmax=2000
  PETmin=300
  PETolerence=0.01
  PEfactr=100000
  PEiter=100
  PEm=10
  PEpenalty=0.1
  PEpgtol=2.22045e-08
  PEsmall=0.1
  PEstep=1.49012e-06>
</globals>
```

The values of `PETmin` and `PETmax`, as was mentioned above, set the default bounds for the temperature. The value of `PETolerence` is used to determine whether the system (2.26) has been solved. If the final F_{min} (Eq. 2.29) is less than `PETolerence`, the system is considered to be solved. If not, the exception is thrown. The value of F_{min} returned by the `PhaseEquilibrium` object in the case of failure is multiplied by the values of the attribute `PEpenalty`. Other attributes set the numerical behavior of the LBFGSB subroutine (see files `lbfgsbss.cpp` and `lbfgsb.f` in the directory `tdlib/lib/toms/` for details).

3.4.8. Object reaction (tdlib/ex/bise)

Quite often, it is necessary to combine the results obtained in the different `algorithm` objects. To this end, the object `reaction` has been developed. Its primarily use was assumed to compute the property of the chemical reaction, but in the final state it can handle much more general computations.

In the directory `tdlib/ex/bise` there are two example, `reac1.mod` and `reac2.mod` that will introduce this object. The first file shows how to compute the EMF of the reaction

$$2Bi(s) + 3Se(s) = 5Bi_{0.4}Se_{0.6}$$

by means of the equation

$$E(T) = -(1/n\mathrm{F})\{5G_{s23}(T) - 2G_{s10}(T) - 3G_{s01}(T)\}$$

The second file computes the drop enthalpy of $Bi_{0.4}Se_{0.6}$

$$\{H(T) - H(298.15)\}/1000$$

Run

```
assess sys reac1
assess sys reac2
```

to see it in action.

Formally speaking, the `reaction` object is a container for the `compute` objects. It takes on input any values from the upper `compute` object and just passes them to all the `compute` objects it contains. On output it supports just `all` textual string that means the combined set of output values from all the inner `compute` objects.

3.5. Describing residuals. Objects and formats in the VARCOMP library

The VARCOMP is the oldest library in the TDLIB. It was mainly written in 1994 when my programming style was pretty close to FORTRAN. The initial reason for what I have decided to switch to C++ was the need to support an array of pointers to functions in order to write the generous subroutine to compute the sum of squares. The FORTRAN (at least at those days) did not have this feature. Well, "A real programmer can write FORTRAN in any programming language" [28].

In the current release, I left the code mostly as it is. I have just added the changes to make VARCOMP compatible with the PHASE and TD_ALGO libraries, and some other improvements in the interface. As a result the code looks rather nasty. Still, it works.

3.5.1. Object residual (tdlib/ex/bacu and tdlib/ex/line)

The object `residual` is used to implement Eqs (2.32) and (2.33). The object takes the experimental point (2.30) on input and tries to estimate $f_i(x_{ij}, \mathbf{z}_i; \Theta)$. If this was successful the residual given by Eq. (2.32) is returned, otherwise the exception is caught to implement Eq. (2.33).

The `residual` object is based on the `compute` object. The idea is to take a set of experimental points (Eq. 2.30, the values of y_{ij}, x_{ij}, $\mathbf{z}_i$ can go in any order), and to pass them to the compute object as the input values. Then the first output value of the compute object is assumed to represent $f_i(x_{ij}, \mathbf{z}_i; \Theta)$ the residual is computed as Eq. (2.32).

The residual object can have several attributes.

Attribute	Meaning
id	identifier, by which it can be referred to in the data file
yname	the name of y_{ij} in the series
xname	the name of x_{ij} in the series (it affects only the use of Eq. 2.35)
ScaleOfX	this is necessary to implement the approach of the like compromise, described in Ref. [11] (by default, it is equal to 1)

If for the series using this residual the variance $\sigma^2_{b,i}$ is assumed to be equal to zero, then the values of the attributes xname and ScaleOfX do not matter during the optimization.

Let us consider several examples.

```
<residual id=Cp23
   yname=Cp
   xname=T>
  <compute>
    <algorithm class=algorithm id=s23></algorithm>
    <input name=T> T </input>
    <output> Cp </output>
  </compute>
</residual>
```

The object above describes the residual

$$\varepsilon_{ij} = C_{p,ij} - C_p^{\mathrm{calc}}(T_{ij})$$

for the phase s23, that should be declared earlier. The experimental point is assumed to have columns Cp and T.

```
<residual id=HL
   yname=y
   xname=x2>
  <compute>
    <algorithm class=algorithm id=L></algorithm>
    <input name=T> T </input>
    <input name=x2> x(Se) </input>
    <output> H_mix </output>
    <output name=x1> x(Bi) </output>
    <output name=x2> x(Se) </output>
    <convert> H_mix/x1/x2/1000 </convert>
  </compute>
</residual>
```

This object describes the residual

$$\varepsilon_{ij} = y_{ij} - \{\mathrm{H}_{\mathrm{mix}}(T_{ij}, x_2)/(x_1 x_2)\}/1000$$

The experimental point is assumed to have columns y, T and x2.

```
<residual id=s23_L
   yname=T
   xname=x2>
  <compute>
    <algorithm class=algorithm id=s23_L></algorithm>
    <input name=T> T </input>
    <input name=x2> x(L,Se) </input>
    <output> T </output>
  </compute>
</residual>
```

The object describes the residual for the liquidus temperature

$$\varepsilon_{ij} = T_{ij} - T_{ij}^{\text{calc}}$$

The experimental point is assumed to have columns `T` and `x2`. The object `s23_L` corresponds to the `PhaseEquilibrium` object, which is necessary to compute the liquidus. Note that in this case the experimental temperature is set as the initial guess to solve the Eq. (2.26) for the given mole fraction of the liquid. This means, that if the liquidus temperature could not been computed because the values of the vector Θ are too bad, the residual will be equated to F_{min}, estimated at the experimental point, that, in turn, corresponds to the “indirect minimization”.

```
<residual id=s11_L_s23
   yname=T>
  <compute>
    <algorithm class=algorithm id= s11_L_s23></algorithm>
    <output> T </output>
  </compute>
</residual>
```

The object describes the residual for the non-variant temperature

$$\varepsilon_{ij} = T_{ij} - T_{ij}^{\text{calc}}$$

The experimental point is assumed to have a column `T`. In this case it is not necessary to set the experimental point as the initial guess to solve the Eq. (2.26). The initial values for both temperature and the mole fraction of the liquid should be set right in the object `s11_L_s23` because they should be the same for all experimental points.

3.5.2. Objects in the category output

The VARCOMP library adds three simple objects to the `output` category, `SeriesOutput`, `ResidualOutput`, and `SpinodalOutput`.

The `SeriesOutput` object allows us to add experimental series to the plot (see, for example, the file `tdlib/ex/bacu/pd.out.mod`). It contains the names of the series and may have two attributes, `Residuals` and `AllPoints`. The first specifies whether the original values should be plotted (`Residuals=0`), or the residuals (`Residuals=1`). The second – whether it is necessary to plot the experimental points marked as “outliers”.

The `ResidualOutput` object simplifies plotting the recommended solution along with the experimental point (see `tdlib/ex/line/ini.mod`). In principle, all the recommended solutions can be done with `ComputeOutput`, but it might take more efforts than in the case of `ResidualOutput`.

The `SpinodalOutput` object checks the binary solution for the miscibility gaps (see, `tdlib/ex/bacu/non.out.mod`). It has rather primitive capabilities, and it should be redone in the future.

3.5.3. Describing experimental values (tdlib/ex/bacu/ and tdlib/ex/line/)

A data file as well as a hypothesis file has a simple, not XML-based format. I developed it in 1994 when I had no idea what the SGML is.

A data file is written in the free format. White space is recognized as the word delimiter, so the identifies cannot contain spaces. The file consists from the experimental series separated by semicolon. Each series comprises a few fields separated by commas

```
series_name,
equation_name,
column_names,
point1,
point2,
...,
pointN;
```

`series_name` is any identifier, by which this experiment will be referred to in the hypothesis file and in the listing. `equation_name` is a residual name which should be defined in the model files. `column_names` means one or a few identifiers separated by space. Their number determines a total number of experimental values in each experimental point. They also will be used in the `residual` object. `point_i` means one of a few double values separated by space. The number of values to read is equal to the number of column names.

If the number of words in the fields `series_name` or `equation_name` is more than one, the only first is taken and others are ignored. If there are more values in the field `point_i` than the number of the column names, the extra values are ignored. If the number of values is less than that, the absent ones are initialized by zeros. Such rules permit you to write comments in the fields `series_name`, `equation_name`, and `point_i`.

You can exclude either a series or a particular experimental point from processing. To this end, the symbol * can be placed in the data file. In order to exclude a series, put the symbol * before the series name, in order to exclude a point, put the symbol * in the beginning of the field `point_i` to exclude. It is also possible to exclude a series in the hypothesis file.

Although points and series marked with * don't take place in the calculations, they can be presented in the listing and in the plots.

In the current release, there are two optional additions to the format described above. First, it is possible to put a few elements

```
<var name=name value=value></var>
```

after the `equation_name` and before the comma, in order to describe the values, which were constant in this experiment.

Second, it is possible to convert the experimental values on input (dimension conversion and so on). To this end, a few `convert` elements could be put before the `column_names`. Each element will be interpreted as a new column with a name given by the `name` attribute. Names of the old columns can be used freely within the `convert` object.

3.5.4. Describing variance components (tdlib/ex/bacu/ and tdlib/ex/line/)

In principle, the hypothesis file is optional, I am not quite sure that the default hypothesis can fit your needs. As a result, if you would like to obtain reasonable results, you have to prepare this file.

The file describes the variance components described in Eq. (2.36). A more accurate statement that the file allows the user to state hypotheses on the reproducibility standard deviation

$$s_{r,i} = \mathtt{sqrt}(\sigma^2_{r,i})$$

and on two ratios

$$\mathtt{sqrt}(\gamma_{a,i}) = \mathtt{sqrt}(\sigma^2_{a,i}/\sigma^2_{r,i}),\ \mathtt{sqrt}(\gamma_{b,i}) = \mathtt{sqrt}(\sigma^2_{b,i}/\sigma^2_{r,i})$$

The default hypothesis is that all the series are assumed to have its own reproducibility variances, $\sigma^2_{r,i}$, but the same quantities

$$\gamma_{a,i} = \gamma_a,\ \gamma_{b,i} = \gamma_b$$

that is, the number of unknown variance components is $N + 2$ (N is the number of experimental series).

The hypothesis file is written in the free format and contains the statements on the experimental series described in the data file. All statements must be finished by semicolon.

The `assess` reads the statement on a particular series, modifies the hypotheses accordingly and continue reading next statements. The program starts the optimization with the only series from `*.set` files taken into account.

The format of a statement on an experimental series is

```
[*] ser_name, hyp_fl sri, hyp_fl sga, hyp_fl sgb;
```

The symbol *, if present, means that the series will be ignored. `ser_name` is a name of the experimental series from the data files. All the words after the first one will be ignored until comma. `hyp_fl` is a hypothesis flag. `sri` is the initial value for standard deviation of reproducibility, `sga` is the initial value of `sqrt`($\gamma_{a,i}$), `sgb` is the initial value of `sqrt`($\gamma_{b,i}$).

There are three choices for the hypothesis flag. The first is the characters # followed by a number from 0 to 29. This means that the variance component belongs to the i-th set with the same variance. The second is the character % which shows that the variance component is unknown but it has no relationships with variance components from the other series. This is for the degenerated cased when the number of unknown variance components in the i-th group is equal to one. Finally, the character * makes the variance component fixed (it will not be changed in the maximization procedure).

For example, the next fragment

```
series_1, #1 1, #3 0, #2 0;
series_2, #1 1, #3 0, #2 0;
series_3, #2 1, #3 0, #1 0;
series_4, #2 1, % 0, #1 0;
series_5, % 1, #3 0, #1 0;
series_6, * 10, * 5, * 5;
```

declares seven unknown variance components, 1) $\sigma^2_{r,1} = \sigma^2_{r,2}$, 2) $\sigma^2_{r,3} = \sigma^2_{r,4}$, 3) $\sigma^2_{r,5}$, 4) $\gamma_{a,1} = \gamma_{a,2} = \gamma_{a,3} = \gamma_{a,5}$, 5) $\gamma_{a,4}$, 6) $\gamma_{b,1} = \gamma_{b,2}$, 7) $\gamma_{b,3} = \gamma_{b,4} = \gamma_{b,5}$. The initial value for the standard deviations is 1, for gamma's – 0. The variance component for the sixth series are fixed.

If you need just weighed least squares, you can write the hypothesis file as follows

```
series_1, * 2.5, * 0, * 0;
series_2, * 3.5, * 0, * 0;
series_3, * 1.5, * 0, * 0;
series_4, * 2.8, * 0, * 0;
series_5, * 3.8, * 0, * 0;
series_6, * 5.5, * 0, * 0;
```

where all the variances of systematic errors are equated to zero, and some fixed values are entered for the standard deviations of reproducibility.

3.5.5. Describing unknowns (tdlib/ex/bacu/ and tdlib/ex/line/)

As was mentioned in the PHASE library, the objects might have a `coef` objects. The simplest form of the `coef` object is as follows

```
<coef> 5 </coef>
```

where the object is considered as anonymous. In this case it is considered to be unreachable from the optimizer.

On the other hand, it is possible to put in an `id` and a flag to show whether this coefficient will be fixed during the optimization.

```
<coef id=a unknown=0> 5 </coef>
<coef id=b unknown=1> 10 </coef>
```

The difference between `<coef> 5 </coef>` and `<coef id=a unknown=0> 5 </coef>` is that the optimizer is already aware of the `coef a` (even though it will be fixed), and the user can change the status of this coefficient in the `par`-files.

We can put lower and upper bounds on the optimized coefficient, if optimizer can handle this information, for example

```
<coef id=c unknown=1 lower=0 upper=10> 3 </coef>
```

In the current release only LBFGSB subroutine could handle the bounds. Other optimizers will just ignore this information.

Another possible attribute is `scale`.

```
<coef id=c unknown=1 scale=10000> 0 </coef>
```

It shows, as it name states, the scale of the unknown variable. The scaling is done the `coef` class itself and, as such, it does not depend on the optimizer.

The next step is to reference the previously declared coefficient, for example

```
<coef IDREF=c></coef>
```

This allows us to declare the same unknown in the several places simultaneously. The namespace of `coef` is global to the whole application.

There are cases when it is necessary to express that there is some constraint between unknowns. This can be handled in some extent with the use of the computed coefficients, for example

```
<coef id=e computed> a + b </coef>
```

where you can put in the body any expression, containing ids of unknown coefficients.

Currently, the next rule is implemented. The namespaces of computed and unknown coefficients are different, and it is not possible to refer to the `id` of the computed coefficient within another computed coefficient. I do not like this rule, and probably it will be modified.

Note that the computed coefficient could be referenced by means of

```
<coef IDREF=e computed> </coef>
```

3.5.6. Describing optimizers

The algorithm for maximizing Eq. (2.37) under the linear error model given by Eq. (2.35) is described in Ref. [8]. It comprises two steps, iterated consecutively. First, the (2.37) is maximized over unknown variances at fixed parameters, and then the (2.37) is maximized over unknown parameters at fixed variances. The last step is the most crucial and it is equivalent, as it is shown in Ref. [8], to the minimization of the least squares. I have had very good results here with the ZXSSQ subroutine from the ISML library (http://www.vni.com/products/imsl/). Their implementation of the finite difference Levenberg-Marquardt algorithm is very robust and reliable. However, it is a commercial product and it is not included in the release.

In TDLIB'00 you have a choice from three optimizers from NETLIB, `tensolve` (default), `lbfgsb`, and `hooke`. This could be set within the `globals` object as follows

```
<globals>
  <optimizer algorithm=tensolve
    NBigIterations=5>
  </optimizer>
</globals>
```

where the attribute `algorithm` sets the optimizer, the `NBigIterations` – the number of iterations over the two steps mentioned above.

Each optimizer has its own options to set. Run

```
assess -m
```

to watch them. Their description can be found in the files `tensolve.f`, `lbfgsbss.cpp` and `lbfgsb.f` in the directory `tdlib/lib/toms/`, and in the file `tdlib/lib/varcomp/hooke.cpp`.

The `hooke` implements the famous Hooke and Jeeves algorithm and it could be used to proceed from the bad initial guess, because the subroutine does not use the objective function derivatives. However it might well take a lot of iterations to complete.

The `tensolve` works reasonably well, provided the initial guess is not very bad. Note that you must scale the unknowns (by means of the attribute `scale`), in order to use `tensolve` successfully. Otherwise, the results are unpredictable.

The `lbfgsb` is the only choice to take into account the bounds on the variable. However, the convergence is slow, especially if the problem tends to be ill-behaved. As `tensolve`, it also requires the scaling.

4. Case studies

4.1. Fictitious A-B binary system from the ThermoCalc Parrot guide (tdlib/ex/tc_ab)

The most famous software for computational thermodynamics is ThermoCalc [12]. I really like it. The most striking feature is that Bo Sundman created it in Fortran. And to achieve that kind of flexibility you have in ThermoCalc by means of Fortran is very, very sophisticated task. Cheers to Bo.

In ThermoCalc there is a module Parrot for the inverse problems, and there is an example there to demonstrate how it works. In the directory `tdlib/ex/tc_ab` there are files to show how this example can be solved by TDLIB. If you have been working with Parrot than you can make the comparison. Below there is just a brief description of the problem. More details can be found in the ThermoCalc guide.

There are primary bcc solid solutions for both solid A and B. They are described by the bcc solid solution with the miscibility gap. The solid B and its primary solid solution at higher temperatures undergoes bcc to fcc transformation. There is no miscibility gap in the liquid phase. There is a stoichiometric compound A2B, which melts congruently but at lower temperatures it decomposes to primary solid solutions. Below there is a phase diagram with the experimental points drawn by TDLIB.

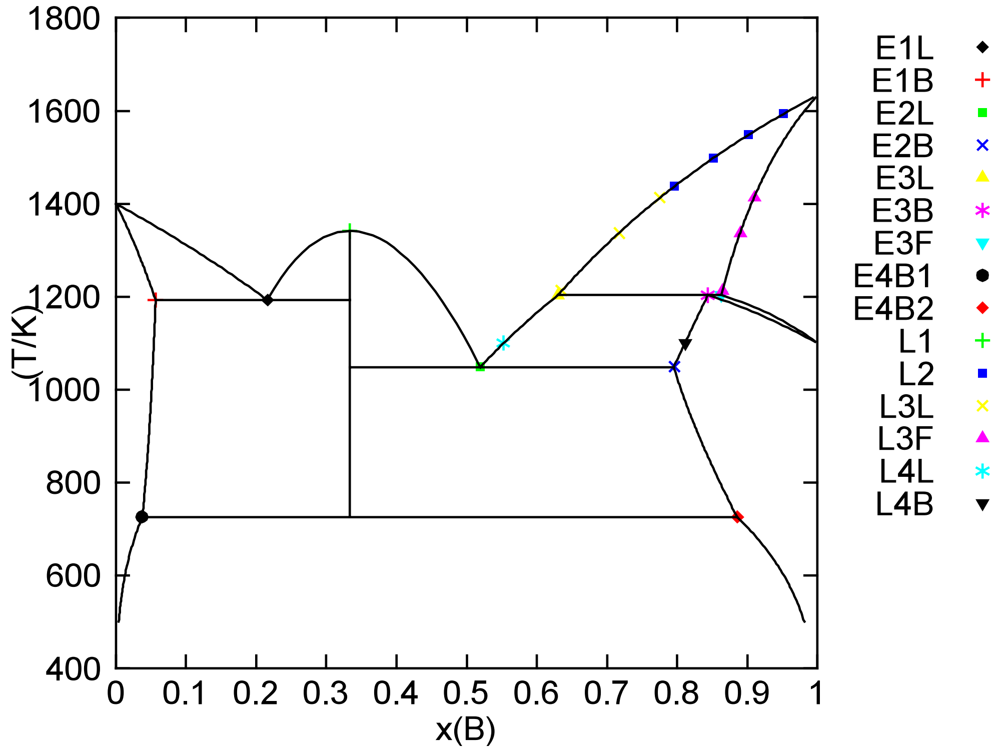


Fig. 4. The phase diagram of the fictitious A-B binary system with the available experimental points.

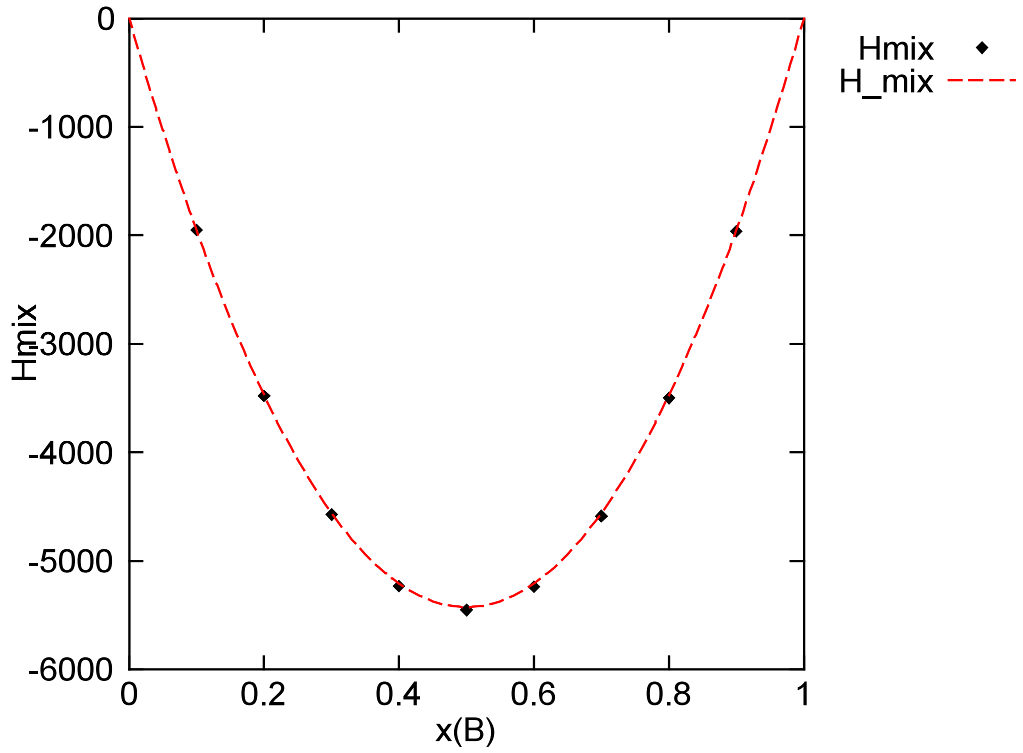


Fig. 5. The enthalpy of mixing of the liquid melts in the fictitious A-B binary system with the available experimental points.

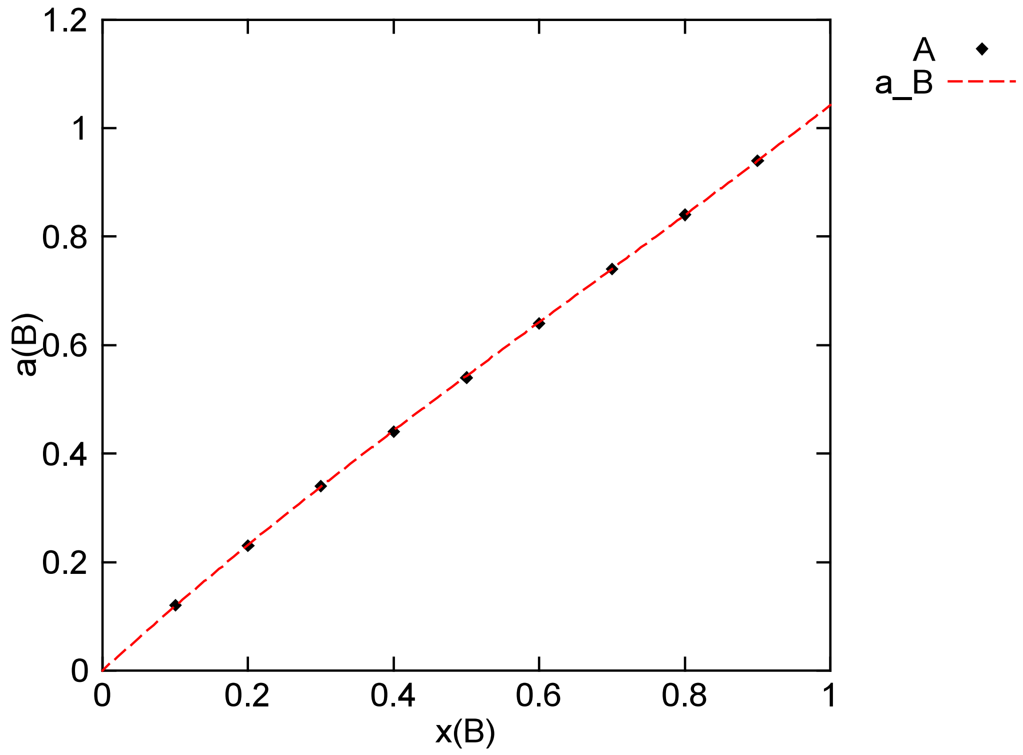


Fig. 6. The activity of B in relation to fcc in the liquid melts in the fictitious A-B binary system with the available experimental points.

For the liquid phase there are experimental enthalpies of mixing and the activity of the second component as shown in Fig. 5 and 6 respectively. The task is to determine unknowns in the Gibbs energies from the available experimental values.

The Gibbs energies of the phases are defined in the file `sys.mod`. All three solution phases are modeled by the Redlich-Kister model (`SimpleSolution` object) as it was suggested in the Parrot guide. The number of unknown parameters is chosen also in accordance with the Parrot guide. Note the scaling of the unknowns. The file `alg.mod` defines the phase equilibria, which are necessary to describe the phase diagram shown above. Once more, in the inverse problem the file analogous to `alg.mod` can be made by hand only, because in the beginning we do not know the values of the unknowns.

The available experimental values are given in the file `expr.dat`. The primary experimental results for the phase equilibria have been given in the weight percents, so in the file they are converted to the mole fractions. In the current release of TDLIB, the values in the non-standard dimensions should be converted to the standard ones. The way of converting is not very elegant, it is relatively low-level, but it is very flexible in nature. Another limitation of the current release is that the experimental point is possible to bind with just a single residual. As a result, when we have a tie-line and it is necessary to form two residuals, the experimental points are duplicated (see series L3L and L3F for example). The file `res.mod` defines the residuals. It shows what calculations are to be done in order to compute the member of the sum of squares related to the given experiment. The file `wls.set` sets the weights. They are assigned the same values as in the Parrot guide.

Finally, the file `out.mod` defines the output to be computed for the found unknown parameters. It describes how to create several files, listed in the next table.

Extension	Description	How to watch the result (`name` is the name of the output given in the command line)
`pd`	The phase diagram with the experimental point (see Fig. 4)	`gnuplot` *`name`*`.pd -`
`hmix`	The enthalpy of mixing with the experimental points (see Fig. 5)	`gnuplot` *`name`*`.hmix -`
`act`	The activity of B with the experimental points (see Fig. 6)	`gnuplot` *`name`*`.act -`
`g600, g1100, g1500`	The molar Gibbs energies of the phase at 600 K, 1100 K, and 1500 K accordingly (see Fig. 7 to Fig. 9)	`gnuplot` *`name`*`.g600 –` `gnuplot` *`name`*`.g1100 –` `gnuplot` *`name`*`.g1500 –`
`non`	Non-variant points.	any text editor
`mis`	The spinodal estimates for the melt, bcc and fcc solutions	any text editor

The command

```
assess sys alg res out -d expr -s wls -o r1
```

will run the inverse problem described above with the initial estimates taken from the file `sys.mod`. I have put there the results pretty close to those obtained in the Parrot guide, so the command above produces the solution of the inverse problem without any problem, and we will consider it as the etalon. The result of this command can be found in the subdirectory out, and the plots are given in Fig. 4 to Fig. 9.

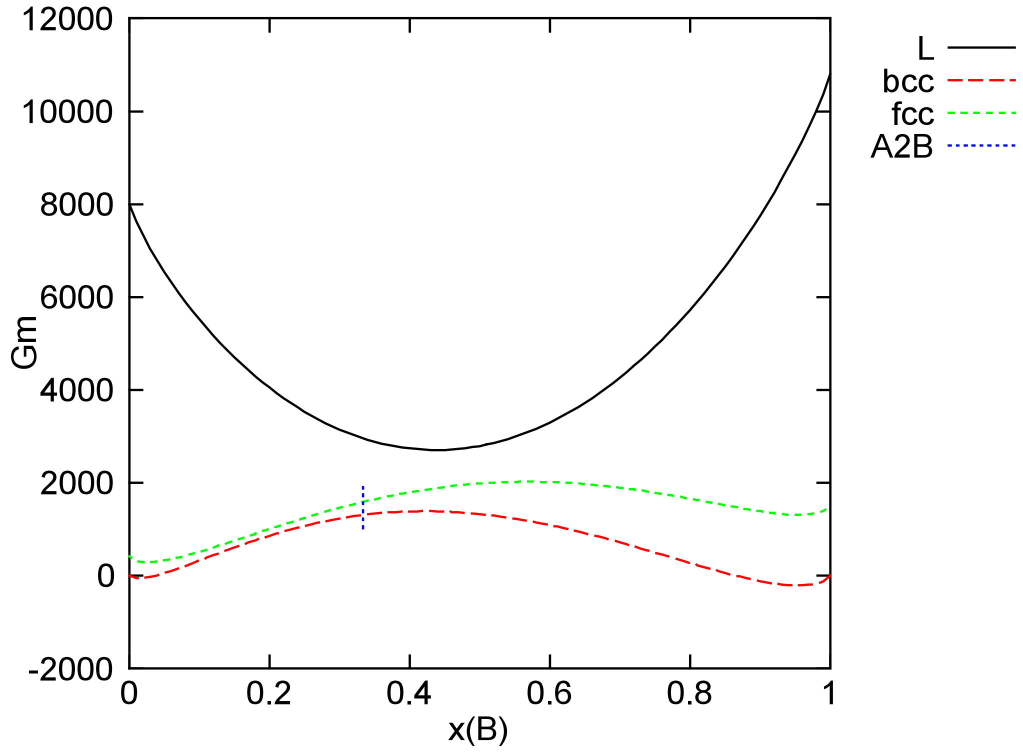


Fig. 7. The molar Gibbs energies of the phases in the fictitious A-B binary system at 600 K.

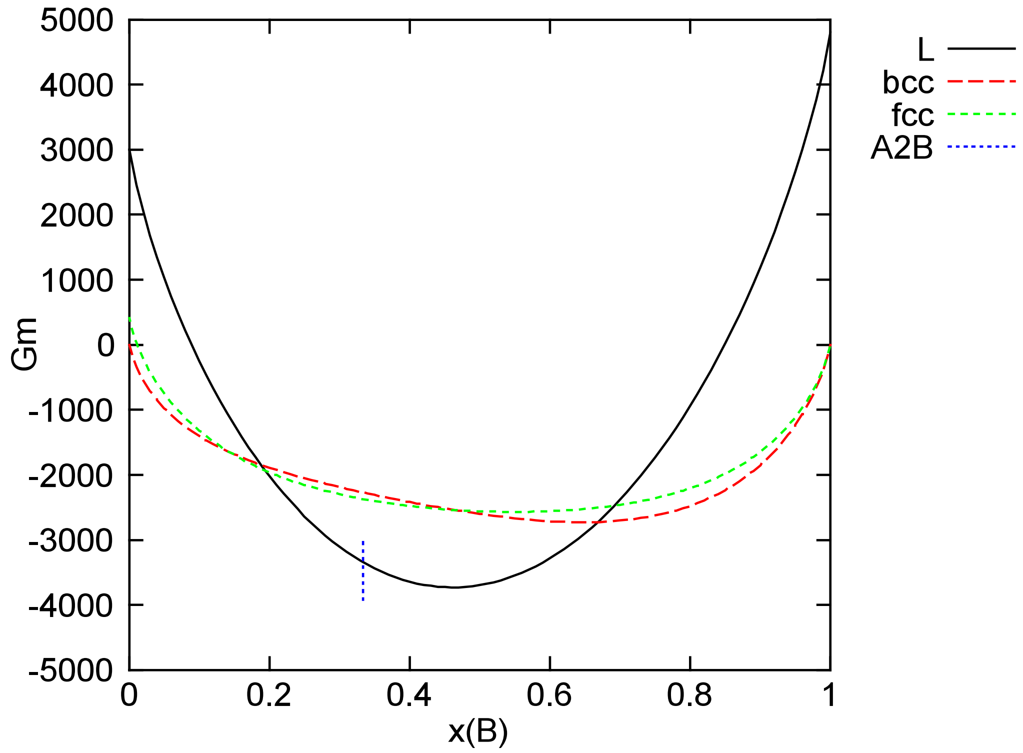


Fig. 8. The molar Gibbs energies of the phases in the fictitious A-B binary system at 1100 K.

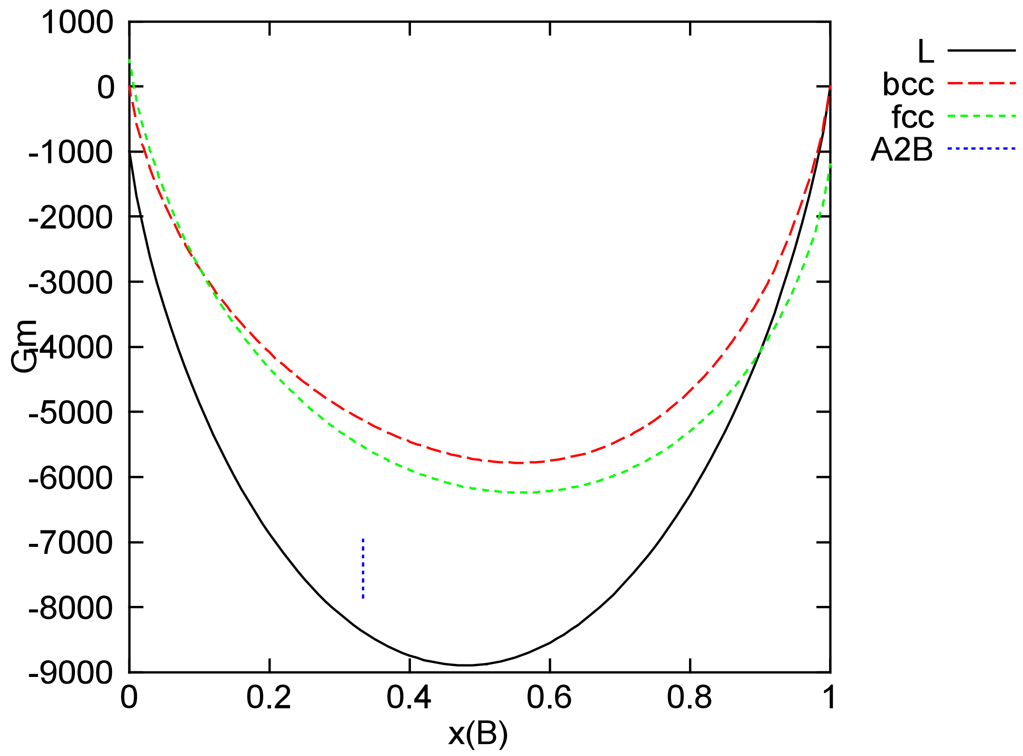


Fig. 9. The molar Gibbs energies of the phases in the fictitious A-B binary system at 1500 K.

Now let us discuss, how it would be possible to obtain the values of the unknowns in the real life, when we do not have initial guess.

In any case we have start with some initial guess. The better it is, the more chances for the success. The file `ini.par` contains the initial guess chosen in the Parrot guide. We could try it with the command as follows

```
assess sys alg res out -d expr -s wls -v ini -o r2
```

It happens that the initial guess chosen in the Parrot guide is considered to be very good for the TDLIB because this command produces the same solution as `r1` and you do not have go through all the steps described in the Parrot guide. Because of the trick to handle the phase equilibria when they are not computed (see Eqs 2.29 and 2.33 and the discussion in the section devoted to the `PhaseEquilibrium` object) the TDLIB finds the solution in this case at once.

Let us try another initial guess that I would have chosen by myself. It is presented in the file `ini2.par`. The reason for this initial guess is as follows. The liquid is considered to be the ideal because the activity and the heat of mixing is not very far from the ideal behavior. The bcc and fcc both are supposed to have a miscibility gap. You should expect it, because there are two primary solutions, and this is pretty typical for the phase diagram shown in Fig. 4. The zeroes for the A2B compound are because I know nothing better to suggest. If you run the optimization starting from this initial guess

```
assess sys alg res out -d expr -s wls -v ini2 -o r3
```

then you should see that for the TDLIB this initial guess is considered to be good either. The result obtained is again the same as `r1`, even though it takes something longer to converge.

Now let us take a really bad initial guess described in the file `ini3.par`. You will find the zeroes only there. It means that all the three solutions phases are considered to be ideal. This initial guess is much close to the real life case when you just do not know were to start. With the default setting, this case breaks TDLIB. If you run

```
assess sys alg res out -d expr -s wls -v ini3 -o p1
```

the optimizer will stop the optimization procedure at the point when the most of the phase equilibria are not computed. Well, the trick implemented in the TDLIB works not in the 100% cases. It is necessary to make much more researches in order to come to truly “automatic optimization”. You must expect that in the real life cases the optimization at once is unrealistic at the current state of the art of the thermodynamic assessment.

What could be done right now? There are several strategies that I employ in my own work. First strategy is described in the Parrot guide – “step by step optimization”. It is necessary to divide the experiments into the groups and to suggest of a sequence of partial optimizations. During them, the assessor comes to a relatively good initial guess, and then he/she performs the full optimization at the last stage. In the case in question, we could separate the experiments on the liquid thermodynamics (file `l.set`), then the experiments on equilibria between the liquid and the A2B phase (file `a2b.set`), the equilibria among the liquid, A2B and bcc (file `bcc.set`), and finally the equilibria including fcc (file `fcc.set`). The typical procedure is to fix all the unknowns but the liquid in the `ini3.par` file and to run

```
assess sys alg res out -d expr -s l -v ini3 -o p1
```

Then, fix the values obtained for the liquid and unfix the values for the A2B phase in the `p1.par` file and run

```
assess sys alg res out -d expr -s l a2b -v p1 -o p2
```

Now, unfix the values for bcc phase in the `p2.par` file and run

```
assess sys alg res out -d expr -s l a2b bcc -v p2 -o p3
```

After that, fix all the values but for fcc phase in the `p3.par` file and run

```
assess sys alg res out -d expr -s l a2b bcc fcc -v p3 -o p4
```

Finally, unfix all the values in the `p4.par` file and obtain the final result

```
assess sys alg res out -d expr -s l a2b bcc fcc -v p4 -o p5
```

The biggest problem with the approach above is that is not quite clear how to separate the problem. I can suggest no formal algorithm here. Try to meditate for a while. It really helps.

Another approach is to play with the option

```
<globals
  PEpenalty=0.1>
</globals>
```

As it was discussed earlier, when the phase equilibria is not computed the `PhaseEquilibrium` object returns F_{min} multiplied by `PEpenalty`. When the initial guess is bad, this product is big and it mostly influences the steps made by the optimizer. During these steps the optimizer comes to such a point from what it can not move further.

There are three files, `p01.mod`, `p001.mod`, and `p0001.mod`, which set this option to 0.01, 0.001 and 0.0001 accordingly. Now try the next commands with the original `ini3.par` file when all the unknowns are unfixed

```
assess p01 sys alg res out -d expr -s wls -v ini3 -o q1
assess p001 sys alg res out -d expr -s wls -v ini3 -o q2
assess p0001 sys alg res out -d expr -s wls -v ini3 -o q3
```

You should see that the third try produces the solution the same as the original `r1`. So, in this case this approach helps to come to the final solution at once even from rather a bad initial guess.

The problem here that it does not work all the time. In many cases the smaller values of the penalty leads to the solution when some equilibria are still not computed but because of the small penalty the optimizer ignores this fact.

Final approach is to try the `hooke` optimizer. Let us consider the solution `q2` obtained above. It gives the reasonable topology of the phase diagram, all the phase equilibria are computed, but the description of the right side of the phase diagram is rather bad. Still, the default `tensolve` optimizer can not make a move from this point because it is gradient-base. If we run `hooke` at this stage it will help us to make a move, and then we can receive the final solution starting `tensolve` again. Run the next commands to see it

```
assess hooke sys alg res out -d expr -s wls -v q2 -o q4
assess sys alg res out -d expr -s wls -v q4 -o q5
```

5. Information for programmers

Just key points for my library are described below. I would suggest you first read carefully this document and run examples in order to understand the whole design. Then have a look at the code. There are a few comments in the files, however knowledgeable people say, "A real programmer does not need comments - a code is obvious" [28].

The library is written in C++. In this release, I have switched from Borland C++ 5.01 to gcc 2.95.2 compiler. I have not tried to port it to other compilers. If your compiler is not gcc, then you have to pay a special attention to the next things.

1) A string class in gcc supports copy-on-write technique. It means that in the next code

```
string a="a very long string"
string b;
b = a;
```

the assignment `b = a` is considered to be a fast operation and it does not require a new memory. If this is not the case in your compiler then the TDLIB will be slower while doing input.

2) There is some Fortran code to compile. You have to compile it and to link it with C++ code. In gcc it could be done with no problem.

3) There is some f2c'd code (converted from Fortran to c with f2c, `http://www.netlib.org/f2c/`). It requires the f2c run-time library. It is included in gcc as g2clib. Also these files require the header g2c.h (in gcc f2c.h from f2c is renamed to g2c.h).

I am working under Cygwin (`http://sourceware.cygnus.com/cygwin/`) under Win32. The filenames here are case-insensitive, so I might miss to check this carefully.

The library is copyrighted by myself and available under the GNU General Public License (version 2 or later). The main idea behind the GNU General Public License is to allow for the widest free distribution of the library (free means freedom, not price). You can find the library at my homepages, `http://www.chem.msu.su/~rudnyi/tdlib/`.

Still, there are some more legal problems because my library relies on other libraries, but they are distributed under their own conditions. Thus, you have to read all the licenses carefully to understand your rights.

Basically, if you are making research you can use all the libraries for free without a problem. If you are going to develop the commercial software, you might be out of luck. Actually, you can do it legally, but first read the GNU General Public License. It imposes certain limits on the distribution of the commercial software. Second, check the licenses for supplementary libraries. You may need to pay to their authors.

All of the libraries are included in the distribution for your convenience. According to their licenses, they can be distributed for free. Whether you can use them or not legally, this is another question.

The table below summarizes all the libraries employed in TDLIB.

Sometimes, I needed to make some changes to the supplementary libraries. It is was necessary either to compile them, or to convert from Fortran to C++ (small editing after f2c), and sometimes to correct the errors. All such cases are documented within the corresponded source code (search for the string "EBR" within the code).

The make from `tdlib/lib` should copy necessary headers to `tdlib/include`, build all the libraries and put the binaries to `tdlib/bin`, and finally to build `assess`, which will also be placed to `tdlib/bin`.

Library	Source, comment	License
phase	mine	GNU Public License
td_algo	mine	GNU Public License
varcomp	mine	GNU Public License
callback	Internet, employed to call a user function from numerical subroutines	free (read the license)
f2c	netlib/f2c, a header that I have not found in gcc	free (read the license)
lapack and blas	netlib/lapack, linear algebra	free (read the license)
minit	Internet, linear programming solver	free (read the license)
toms	netlib/toms, numerical solvers	free for research (read the license)

In the tables below there are lists of files from my libraries with small comment on what you will find there. The files go in the order, in which I would recommend to get acquainted with them.

The files in the PHASE library

file	Classes	description
general.h and general.cpp	gError, CanNotCompute, CheckInf, PutTab, ObjToString, limits, global, StateTp, StateX, FirstDerivative, SecondDerivative, MixedDerivative, instringstream, SGML, parser	Some definitions and small objects employed throughout the library
calc.h and calc.cpp	calculator	A simple interpreter
coef.h and coef.cpp	coef, CalculatorWithCoefs	Unknown parameters to be determined in the optimizer
elem.h and elem.cpp	elem	Chemical element
formula.h and formula.cpp	MolecularFormula, formula	Molecular formula and chemical formula
func_Tp.h, func_Tp_imp.h, func_Tp.cpp	Ref_func_Tp, func_Tp, simple_Tp, null_Tp, Cp_zero, Cp_const, Cp_BB2, Cp_BB2_Tref, Cp_BB4, IVT_Tp, SGTE_Tp, ideal_gas, V_const, alpha_const, alpha_kappa_const, alpha_kappa_const2, compound_Tp, complex_Tp, calc_Tp	Function in temperature and pressure
species.h, species.cpp	Ref_species, species, matrix, GetNu	Species and handling of the formula matrix
func_tpx.h, func_tpx_imp.h func_tpx.cpp, interact.h, interact.cpp	SymMatrix, RefFuncTpx, FuncTpx, NullFuncTpx, IdealMixing, Reference, interaction, RedlichKister, HochArpshofen, Polynomial	Function in temperature, pressure, and mole fractions
phase.h, phase.cpp	ArrayFirstDerivative, ArrayMixedDerivative, RefPhase,	The abstract definition of the phase and the simple phase

	ObjFromProxy, phase, RefSimple, NullPhase, PointPhase, NumericalDerivatives	models
intvars.h, intvars.cpp	RefInternalVariables, OneInternalVariable. InternalVariables	The abstract definition of the solution model with the internal variables
simple.h, simple.cpp	SimpleSolution	The solution without internal variables
cuox_or.h, cuox_or.cpp	CuOx_ordered_plane	Y247-like solutions
cuox_pl.h, cuox_pl.cpp	CuOx_plane	Y123-like solutions
ass_sol.h, ass_sol.cpp	AssociatedSolution, AssociatedSolutionBasis, AssSolOneReact, AssSolManyReact, associated_solution	Associated solution models

The files in the TD_ALGO library

file	Classes	description
td_algo.h, td_algo_imp.h, td_algo.cpp	convert, RefAlgorithm, algorithm, SimpleAlgorithm, InputValue, OutputValue, compute, PassThrough, PhaseProperty, reaction	The abstract definition of the algorithm and the simple algorithms
phase_eq.h, phase_eq.cpp	PhaseEquilibrium	Algorithm for the phase equilibrium
output.h, output.cpp	RefOutput, output, NullOutput, comparison, CycleOutput, ComputeOutput, OutputFile	The abstract definition of the output and the simple objects

The files in the VARCOMP library

file	Classes	description
common.h, data.h, common.cpp, data.cpp	data	Experimental points
residual.h, residual.cpp	residual	Residual
opt.h, opt.cpp, hooke.cpp	RefOptimizer, optimizer, tensolve, LBFGSB, Hooke, ZXSSQ	Optimizers
sumsqr.h, static.cpp, sumsqr.cpp, sumsqrin,cpp, sumsqrut.cpp, sumsqrpr.cpp	series	The implementation of the algorithm to maximize the likelihood function
post_an.h post_an.cpp	SeriesOutput, ResidualOutput, SpinodalOutput	Post-analysis
global.cpp	globals	Support for the globals object
assess.cpp		A stub to compile assess

6. Question, discussion, bugs, etc

The library presented is far from being complete. I consider it just as TDLIB’00. I have set up a discussion list

`tdlib@td.chem.msu.su`
for those who interested in these matters. Your questions, comments, bug reports are welcome.

In order to subscribe send a line with one word `subscribe` to
`tdlib-request@td.chem.msu.su`
If you would like just to receive notifications for the next updates, send a line with one word `subscribe` to
`update-request@td.chem.msu.su`

7. Acknowledgment

The library presented was partly made possible by Grant 99-03-32629 from the Russian Foundation for Basic Research and by Grant 99079 from Russian State Program "High Temperature Superconductivity".

8. Literature

1. T.M. Gordon. Software and Data for Thermodynamics and Phase Equilibrium Calculations in Geology, http://ichor.geo.ucalgary.ca/~tmg/Research/thermo_links.html.
2. C.W. Bale. Web Sites in Inorganic Chemical Thermodynamics, http://www.crct.polymtl.ca/fact/websites.htm.
3. Proceedings of the 1995 Ringberg Workshop on Unary Data, CALPHAD, 1995, v.19, N 4.
4. The GNU Manifesto, http://www.gnu.org/gnu/manifesto.html.
5. The GNU General Public License, http://www.gnu.org/copyleft/gpl.html.
6. GNU's bulletin, http://www.gnu.org/bulletins/bulletins.html.
7. W.R. Smith, R.W. Missen. Chemical Reaction Equilibrium Analysis: Theory and Algorithms. 1982, Wiley and Sons Inc., 364 p.
8. E.B. Rudnyi. Statistical model of systematic errors: linear error model. Chemometrics and Intelligent Laboratory Systems. 1996, v. 34, N 1, p. 41-54. The earlier version is available at http://www.chem.msu.su/~rudnyi/pcmler.html.
9. E.B. Rudnyi. Statistical model of systematic errors: An assessment of the Ba-Cu and Cu-Y phase diagram. Chemometrics and Intelligent Laboratory Systems, 1997, v. 36, p. 213-227. Some information is available at http://www.chem.msu.su/~rudnyi/bacuy/.
10. V.V. Kuzmenko, I.A. Uspenskaya, E.B. Rudnyi. Simultaneous assessment of thermodynamic functions of calcium aluminates. Bulletin des Societes Chimiques Belges, 1997, v. 106, N 5, p. 235-243. The preprint is available at http://www.chem.msu.su/~rudnyi/aluminates/.
11. E.B. Rudnyi, V.V. Kuzmenko, G.F. Voronin. Simultaneous assessment of the YBa2Cu3O6+z thermodynamics under the linear error model, J. Phys. Chem. Ref. Data, 1998, v. 27, N 5, p. 855-888. The preprint is available at http://www.chem.msu.su/~rudnyi/Y123/.
12. ThermoCalc, http://www.met.kth.se/tc/tc.html.
13. E.B. Rudnyi, G.F. Voronin. Classes and objects of chemical thermodynamics in object oriented programming. 1. A class of analytical functions of temperature and pressure, CALPHAD. 1995, v. 19, N 2, p. 189-206.
14. E.B. Rudnyi. Classes and Objects of Chemical Thermodynamics in Object-Oriented Programming. 2. A Class of Chemical Species. Second Electronic Computational Chemistry Conference, November 1995. http://www.chem.msu.su/~rudnyi/species/.
15. B. Stroustrup. The C++ Programming Language (Third Edition), Addison-Wesley, 1997.
16. A Gentle Introduction to SGML, Chapter two of Guidelines for Electronic Text Encoding and Interchange (TEI P3), http://www-tei.uic.edu/orgs/tei/sgml/teip3sg/
17. C.F. Goldfarb. The SGML Handbook. Oxford: Oxford University Press, 1990. 688 p.

18. Thermodynamic properties of individual substances (in Russian). ed. V.P. Glushko. Moscow, Nauka, in four volumes, 1978 - 1982.
19. R.G. Berman, T.H. Brown, Contrib. Mineral. Petrol. 89, 168 (1985).
20. A.T. Dinsdale. SGTE data for pure substances. CALPHAD, 1991, v. 15, N 4, p. 317-425.
21. K.C. Chou, Y.A. Chang. A study of ternary geometrical models. Ber. Bunsenges. Phys. Chem., 1989, v. 93, N 6, p. 735-741.
22. H.L. Lukas, J. Weiss, E.-Th. Honig. Strategies for the Calculation of Phase Diagram. CALPHAD, 1982, v. 6, N 3, p. 229-251.
23. M. Hoch. The application of the Hoch-Aprshofen model to liquid systems with compound-forming tendencies and a miscibility gap. CALPHAD, 1987, v. 11, N 2, p. 225-236.
24. M. Hillert, L.I. Staffansson. The regular solution model for stoichiometric phases and ionic melts. Acta Chem. Scand. 1970, v. 24, p. 3618-3626.
25. V.A. Lysenko. A thermodynamic model for liquid systems containing nonmetallic elements. Inorganic Materials. 1998, v. 34, N 9, p. 926-931.
26. A.D. Pelton, S.A. Degterov, G. Eriksson, C. Robelin, Y. Dessureault. The modified quasichemical model. I. Binary solutions. 1999, to be published.
27. The ASCEND IV Syntax and Semantics for release 0.9, http://methi.ndim.edrc.cmu.edu:8888/pdfhelp/syntax.pdf, see also http://www.cs.cmu.edu/~ascend/pdfhelp.htm.
28. Real Programmers Don't Use Pascal, DATAMATION, 1983, v. 29, N 7, p. 263-265.
29. Y.Jiang, W.R. Smith, G.R. Chapman. Global Optimality Conditions and their Geometric Interpretation for the Chemical and Phase Equilibrium Problem. SIAM J. Optimization, 1995, November issue.
30. A.V. Antipov, E.B. Rudnyi, Zh.V. Dobrokhotova. Thermodynamic Assessment of the Bi-Se system. Inorganic Materials, 2001, N2. The preprint in Russian is available at http://td.chem.msu.su/publ/bise.doc.