# User's Manual to Kinetic Compiler version 1.00

Kenneth Geisshirt
kneth@osc.kiku.dk
Department of Theoretical Chemistry
H.C. Ørsted Institute
Universitetsparken 5
2100 Copenhagen
Denmark

5 October 1994

# Contents

# 1 Introduction

The kinetic compiler is a program which converts a chemical model into an equivalent simulation program.
This manual describes the kinetic compiler `kc` version 1.00. The manual will describe the program from the user's point of view, *i.e.*the input format and not the internal workings.
The internals of the compiler are documented elsewhere, Geisshirt [**?**]. This document can be obtained from the author.
The kinetic compiler forms a language much like an ordinary programming language. The only difference is the size. While real programming languages are big, the kinetic compiler's is small.
In this manual I will use `type writing` in examples and terminals in grammars. The brackets ([ and ]) will surround optional parts, while curled brackets ({ and }) will be 0, 1 or more repetitions of a part in the grammars. A vertical line (|) denotes a choice between two grammar parts. Normal parenthesis group grammar parts.

# 2 Basics

In this section I will examine the basic components of `kc`, *i.e.*numbers, names and expressions.

## 2.1 Numbers

In the language of `kc` there is only one type of numbers. It is essential floating-point numbers. They have the form:
{ `digit`} [`.` `digit`{`digit`} [(`E`|`e`) `+` |`-` `digit` `digit`]
The following numbers are legal:

```
322
5.0
0.1
5.0e+10
```

Negative numbers are supported as a part of the expressions instead of a part of the number. Please note, that the sign of the exponents have to be written, *e.g.*like the last number above.

## 2.2 Species

Species are essential to the kinetic compiler. A species consists of a name and a charge. The "grammar" of a species is:
`name` [`(` (`+`|`-`) [`number`]`)`]
A name is a letter followed by letters and digits. The following species are legal:

```
H2O
H(+)
SO4(-2)
```

The parenthesis is the charge of the species. A dot (.) denotes radicals, and if the charge is omitted, it is assumed to be zero.
Capital letters and non-capital letters are *not* the same, *i.e.*the compiler is case-sensitive. The sign of the charge must be written down, *i.e.*both minus and plus signs!

## 2.3 Concentrations

Concentrations can be specified in `kc`. They are written using the convention in chemistry, *i.e.*a species in brackets ([ and ]). The concentration of `H2O` can be written as `[H2O]`.

## 2.4 Expressions

Expressions are meant to implement general calculations in the compiler. The expressions are the same as found in ordinary programming languages. An expression (`expr`) is defined recursively by:
`expr` op `expr`
The op is an operator. It can be one of the five operators used in mathematics, *i.e.*addition, subtraction, multiplication, division, and power-raising.
They are written as:

| Operator | In `kc` | Precedence |
|---|---|---|
| Addition | `+` | 3 |
| Subtraction | `-` | 3 |
| Multiplication | `*` | 2 |
| Division | `/` | 2 |
| Power | `**` or `^` | 1 |

Expressions can be surrounded by parenthesis, *i.e.*`( expr )`. An expression can also just be a numeric constant (a number) or a symbolic constant (a name). One can in some cases also use concentrations in an expression, see section 3.2. Finally, an expression can change sign by `- expr`.
Functions can also be used in expressions. All the usual functions are implemented. An application of a function has the form:

```
f ( expr )
```

where `expr` is an expression and `f` is the function. The following functions are implemented:

| Function | Description |
|---|---|
| `sin` | sine |
| `cos` | cosine |
| `tan` | tangent |
| `asin` | the inverse of sine, *i.e.*$\sin^{-1}$ |
| `acos` | the inverse of cosine, *i.e.*$\cos^{-1}$ |
| `atan` | the inverse of tangent, *i.e.*$\tan^{-1}$ |
| `sinh` | hyperbolic sine |
| `cosh` | hyperbolic cosine |
| `tanh` | hyperbolic tangent |
| `asinh` | the inverse of hyperbolic sine |
| `acosh` | the inverse of hyperbolic cosine |
| `atanh` | the inverse of hyperbolic tangent |
| `exp` | the exponetial function ($e^x$) |
| `ln` | base $e$ logarithm |
| `log` | base 10 logarithm |

Let me close this section with an example:

```
A*exp(-dE/(k*T))-1.5*C^2
```

## 2.5 Running `kc`

Running the compiler is easy. It is done by typing the following on the command-line:
`kc` [options] `< input-file`
The compiler creates the output files. All options are a hyphen (minus sign) followed by a letter. There are the following options:

| Option | Description |
|---|---|
| q | quiet mode |
| h | Help |
| m | Mode |

The last option (m) is followed by a number. The mode is a code generator, which generates code to a specific program. To see which modes `kc` supports, run the program with option h (`kc -h`). In the present version the following code generators are supported

| Mode | Code generator | Description |
|---|---|---|
| 2 | KGadi | A solver for reaction-diffusion systems |
| 3 | kci | A dynamical simulator written in ANSI-C |
| 5 | KnCont | Continuation program |
| 6 | Finn | Calculating various properties |

# 3 The input

In this section I will explain the overall structure of the input to the `kc`. The input to `kc` consists of three parts; definitions, the model and constraints. Each part is in the following subsections. The three parts are written in the same order in the input file.

## 3.1 Definitions

In this part of the input file, one can define constants, which can be used later in the input file. A (symbolic) constant is a name, which has been assigned a value. The constants are written as:

```
const = expr;
```

The compiler must be able to evaluate the expression at that moment, *i.e.*the expression is not allowed to contain any undefined constants. The constant can also be a string. In that case the expression on the left hand side is a sequence of letters and digits surrounded by quotation marks. Note that it is not possible to used full stops (.), dashed (-) or anything like that in strings. In this part the user can also specify which symbolic names, which are going to be used in a continuation. The parameter declaration is

```
#paramter {param expr, expr, expr, expr, expr};
```

These five values are used in the continuation, but I will not describe them here, since the supporting mode is not yet fully functional.

## 3.2 The model

The model is a set of chemical reactions and/or a set of ordinary differential equations. The reactions are written in the "usual" way, *i.e.*a coefficient and a species plus another coefficient and a species and so on. Two different reactions can be used; uni-directional and bi-directorial. They are written using `->` and `<->`. The bi-directorial reaction can also use `<=>`.
The rate constants are written after the reaction. A reaction always begin by a number. A legal reaction is:

```
1: H(+) + HO(-) <-> H2O; k>=1.0; k<=2.0;
```

The default code generating uses the law of mass action. This default can be overriden. One can either use power law or general expressions. The first is simple. After the reaction, one can write the power constants. Their grammar is `c(species) = expr`. When using the power law, the user must still write the rate constants.
The general expression kinetic is also very simple to write down. Instead of rate constants, the user can write a general expression. They are thought to be used in enzyme kinetics, but may also be useful in other areas. A legal input of this kind is

```
1: A + B <-> C; v>=2*[A]; k<=10
```

Note that it is possible to use a mix of the three kinetics (mass action, power and general) in the same model.
Often the parameter in a continuation is a rate constant. In this case, the rate constant is just set equal to the name of the parameter, *i.e.*the name declared in at the `#parameter` directive.
Ordinary differential equations are normally written as

$$\frac{dx}{dt} = f(x),$$

and the compiler's grammar is

```
name' = expr;
```

where `name` is the $x$ and `expr` is the velocity field specified by $f$.

## 3.3 Constraints

In the last part of the input, all the constraints and initial conditions are placed. It is also possible to specify constants for the species.
The constraints have the form `[species] = expr`. The constraints reduces the number of dynamical variables in the model. Chemical equilibrium is also modelled this way. A legal constraint is `[A] = 12-[B]`.
Initial conditions specify which values the concentrations of the species have at $t = 0$. This is done by `[species](0) = expr`. If no initial concentration

is given, the compiler assumes, that the concentration is zero. For dynamical variables defined by ordinary differential equations, the initial value can also be specified. This is done by `name(0) = expr`.
To each species one can assign a number of constants. Some code generators may use *e.g.*the diffusion coefficient of the species. The constants will have a name followed by the species in parenthesis, *e.g.*`D(A) = 1e-5`.

# 4 The code generators

For each code generator, there are various constants and parameters, which can be given a value. In this section I will discuss the various code generators.

## 4.1 kci

Kci is able to solve ordinary differential equations numerically. There is a number of parameters, which can be get.

| Parameter | Description | Default value |
|---|---|---|
| dtime | interval between printouts | 2.0 |
| etime | time for endning simulation | 200.0 |
| htime | initial step size | 1.0 |
| stime | initial time | 0.0 |
| epsr | relative tolerence | $10^{-5}$ |
| epsa | absolute tolerence | $10^{-10}$ |
| epsmin | minimal precision | machine precision[1] |
| method | integration method (see below) | 1 |
| stepadjust | | 0.9 |
| maxinc | | 1.5 |
| mindec | | 0.5 |
| scaling | | 0 |
| htimemin | minimal step size | $10^{-20}$ |
| datafile | name of output file | kinwrkdat |
| printafter | time to start printing | 0 |
| mode | mode (see below) | 0 |
| prnmode | printing mode (see below) | 0 |
| debug | debug level (see below) | 0 |

The "method" determines which integration scheme to use, and it can have the following values:

| Value | Method |
|---|---|
| 1 | Calahan |
| 2 | Rosenbrock (RKFNC) |
| 3 | 4th order Runge-Kutta |
| 4 | Generalised Runge-Kutta |
| 5 | Rosenbrock for nonautonomous systems |
| 7 | Generalised Runge-Kutta for nonautonomous systems |

The "mode" parameter decides different ways of using the simulator. The "mode" 0 is default. The table below gives the possibilities:

| Mode | Description |
|---|---|
| 0 | Ordinary simulation |
| 1 | Make perturbations |

The perturbation is a simple feature: At a specified time, a vector is added to the concentration vector (one can have negative values - simulates dilution). The time for the first perturbation is given by the parameter "ptime", while new perturbations are done by an interval "dptime". If "dptime" is zero (default) only one perturbation is done.
In general the output file is readable by GNUplot. At the begining of the file, the names of the dynamical variables are found. Some modes may write additional information in the file and at the end. The parameters "prnmode" and "debug" determine the additional information printed.

| prnmode | Description |
|---|---|
| 0 | equidistant and extrama points |
| 1 | equidistant points only |

| debug | Description |
|---|---|
| 0 | none |
| 1 | time, steplength, and dynamical variables |
| 2 | as 1 + control parameters |
| 3 | initial values of control parameters |

In order to make the work easier for the user, there exists a small script. The script runs the kinetic compiler, a C compiler *etc.*The script is called kci[2].

[2]Kinetic Compiler and Integrator.

If the model is stored in a file called `foo.des`, a simulation is performed by `kci foo.des`. It should be noted that the MS-DOS version of the kinetic compiler uses a similar batch file (the name is the same).

## 4.2 Kin

Kin is a simulation package for chemical reaction, or to be more precise, it is a solver of ordinary differential equations. Calahan's method is used to solve stiff problems. This mode is obsolent.
The following parameters can be set as constants in the input to `kc`:

| Parameter | Description | Default value |
|---|---|---|
| tb | | 1 |
| dt | step between prints | 1 |
| etime | end time | 10 |
| hb | | 1 |
| epsr | relative precision | $1 \cdot 10^{-3}$ |
| epsa | absolute precision | $1 \cdot 10^{-20}$ |
| mode | run mode | none |
| ptime | perturbation time | none |

The "mode" parameter have the following meaning:

| Value | Description |
|---|---|
| 0 | Ordinary simulation |
| 1 | Make a perturbation at `ptime` |

If initial concentrations are specified, they will be used. The concentrations used in the perturbation are declared as species related constants, see section 3.3. The name of the constant is `pert`, and an example is:

```
pert(X)  = 0.1;
```

The output will always be stored on the file `kinwrk.dat`, which is readable by GNUplot[3].
The easiest way to use this mode, is to use the script `kkin`. Let us assume that the input file is called `model.des`. The run is then done by the command: `kkin model.des`.

[3]GNUplot is a plotting program, which can be obtained by anonymous ftp.

## 4.3 KGadi

This code generator supplies code to a simulator of reaction-diffusion systems. Therefore diffusion coefficients must be specified. They are specified by the species-related constant `D`. Diffusion coefficients for variables specified by differential equations cannot be defined in the input to `kc`. The user must edit the routine `init_diff_const` in the file `model.c` by hand. There are the following constants to be used:

| Parameter | Description | Default value |
|---|---|---|
| mgrid | grid points, horizontal | none |
| ngrid | grid points, vertical | none |
| length | length of system | 100 |
| dt | time step | 2 |
| update | time between saving on disk | 10 |
| print1 | time before saving | 100 |
| print2 | time for ending saving | 100 |
| mode | run mode | 0 |

The "mode" parameter determines which kind of simulation to perform. If the parameter is not defined, it will be a ordinary simulation, where the concentrations are saved every "update" second. The table below shows the possibilities.

| Value | Description |
|---|---|
| 0 | Ordinary simulation. |
| 1 | Generates a sequence of "images" between "print1" and "print2". |

The easist way to use this mode, is to use the script called `rdsim`. If the model is in file `file.mod`, a simulation is performed by the command `rdsim file.mod`.