# DOPDEES

(Dial an Operator Partial Differential Equation Evaluator and Solver)

# User's Guide and Reference Manual

Alp H. Gencer

Version: 99.11.08

# Contents

# Chapter 1

# Introduction

DOPDEES is a multi-purpose PDE initial value solver in one spatial dimension that uses dial-an-operator paradigm for specification of the equation system. The primary aim of the code is rapid development of continuum models and may be used in a variety of systems. It is intended to be easy to use and easy to extend with new operators.

DOPDEES can be used in any system where the user wants to get solutions of coupled partial differential equations. The program was originally written to solve diffusion/reaction problems, but can be suited to any purpose. If the operators or functions the user needs are not included in the code, it is relatively easy to add them. Other nice features include allowing the user to select the integration engine, as different systems might benefit from different numerical algorithms. There are commands for grid generation and result extraction.

The user interface of DOPDEES is in Tcl, which provides a powerful way of scripting for the input. The Tcl syntax, as well as the basic Tcl commands are explained in Chapter 2. The commands that DOPDEES adds to Tcl are explained in Chapter 3. The list of operators and functions defined in DOPDEES is discussed in Chapter 4.

## 1.1 Overview of concepts

DOPDEES solves a given set of partial differential equations. The (one dimensional) space is assumed to be divided into chunks called *regions*, and it is assumed that different equations need to be solved in different regions. The solution variables are called *fields* and it is assumed that for all fields equations of the type $\partial f/\partial t = \ldots$ exist, that describe the PDEs. A field specified in one region doesn't exist in others, unless you explicitly specify it. Actually, the only communication between regions can happen through boundary transfers.

The right hand side of partial differential equations are specified as a sum of *operators*: $\partial f/\partial t = \sum \mathrm{op}_i$ This approach is called the *dial-an-operator* approach, since the user can choose the operators on the right hand side. The operators can make use of *functions*, which calculate a certain function of their inputs.

### Advantages

- Easy to learn user interface in Tcl.

- Small, easily expandable and fast code in C++.
- Different systems of equations can be specified in different regions, with different fields.
- User-selectable integration engines.

**Shortcomings**

- 1-dimensional.
- Supports only parabolic equations of type $\partial f/\partial t = \ldots$ In practice, this shouldn't be a problem, since PDEs involving higher order time derivatives may be decoupled into sets of PDEs with first order time derivatives.

## 1.2 Invoking DOPDEES

DOPDEES can be invoked in either interactive mode or can be made to process a script by giving the script name as an argument. In either case, it first searches a user preferences script, and executes it. This script is usually `~/.dopdeesrc`, but can be changed by setting the `DOPDEESRC` environment variable. The environment variable `BUSTOPDIR` has to be set to the directory contaioning the BUSTOP programs prior to using any BUSTOP program. Also, add the directory `$BUSTOPDIR/bin` to your path.

If DOPDEES is invoked with no command line arguments it goes into interactive mode. It prints out a prompt and waits for the user to type commands. After each command typed, DOPDEES processes the command and prints out the result of the command. This mode is usually not very well suited for running simulations, but may be used for quick checks.

# Chapter 2

# An overview of Tcl

## 2.1 Description of syntax

This section is intended to give a brief overview of the capabilities and syntax of Tcl, without going into the details of the "language". Users who wish to learn more about Tcl may refer to the following references:

- J.K. Osterhout. *Tcl and the Tk toolkit.* Addison Wesley (1994). ISBN 0–201–63337–X.
- B.B. Welch. *Practical programming in Tcl and Tk.* Prentice Hall (1995). ISBN 0–13–182007–9.

I think it is important to realize that Tcl is a string parser and not a programming language in the sense we think (such as C and perl, both of which have very complicated syntactical rules). Actually, Tcl shows that a language can be created with very little syntax and clever definition of commands. The basic building blocks of a Tcl script are *commands* which consist of *words* (command name and arguments) separated by spaces:

```
command arg1 arg2 ...
```

Newlines and semicolons serve as command separators. Long lines can be continued on the next line by ending the line with a backslash. Whitespace serves as word separator. The important thing to note about Tcl is that all commands, including flow control, assignment and subroutines have this generic form.

A special "command" is the pound character (`#`), which is the "comment command". Thus you can type a pound and then a comment *whenever Tcl is expecting a command.* If at the current position of the script, Tcl isn't expecting the beginning of a command, (meaning it's not the beginning of a line or it is not the first character after a semicolon) the pound is interpreted literally.

### 2.1.1 Substitutions

As mentioned, Tcl is basically a string parsing language. Thus everything happens in the form of strings flying back and forth. A very important concept is the concept of *string substitutions* which gives you a very powerful way of manipulating strings. There are three kinds of string substitutions that can occur in a Tcl script:

### Backslash substitutions

This is similar to many languages: a backslash followed by a character has a special meaning. Favorite examples are `\n` for newline, `\077` for octal codes, `\$` for a literal dollar sign, etc. Backslash-newline sequences are replaced by a space, and this is why you can continue on the next line when you end a line with backslash.

### Variable substitution

A dollar sign followed by a variable name causes a variable substitution. All variables in Tcl have string values, thus this substitutions is literally a string insertion. `$day` will substitute the value of variable *day* at this point of the script. If the context is such that the variable name cannot be identified, curly braces must be used, such as in `${day}th day of week`.

### Command substitution

This is one of the most powerful features of Tcl. A command enclosed in square brackets like `[command args]` causes the output of the command to be substituted at this point of the script. This works much like the back ticks in a shell script. This provides the way to manipulate the outputs of Tcl commands.

## 2.1.2 Grouping

The words are separated by whitespace. It is important to note that *groping occurs before substitutions.* Thus in a construct like `word1 $var word3`, even if the variable's value contains a space, it is considered to a single word.

Tcl provides you with two ways of *grouping.* When we talk about grouping, we mean the clumping of words (separated by whitespace) into a single word.

### Grouping with substitutions

The first way to quote is with double quote signs (`"`) as in `"This is a quote."`. All characters until the next quote sign are count as a single word, even if spaces occur within the quotation. Substitutions of all three kinds do occur within double quotes. Thus in a construct like `"The price is $price."` the value of the variable *price* will be substituted. If you want to use a double quote character within double quotes, you may use a backslash substitution (`\"`).

### Grouping without substitutions

If you group with curly braces `{ }`, no substitutions will occur (not even backslash substitutions). For any opening brace, Tcl finds the matching closing brace and treats everything within as a single word. This is a very powerful technique for grouping large portions of Tcl scripts into a single word.

## 2.2 Tcl commands

As you can see, we have described the whole Tcl syntax in just two pages. With this string manipulation techniques, and the built-in commands described below, it is amazing how much you can do with Tcl. In description of the syntax below, command names appear **boldface** and arguments appear *slanted*.

### 2.2.1 Variable commands

These are the commands in Tcl that manipulate variables:

- **expr** *expression*

  The **expr** command concatenates all of its arguments into a single string and evaluates it as a mathematical expression. It returns the resulting number, or an error message if the evaluation fails. **expr** understands most C operators, as well as some built-in math functions. Examples:

  - `expr 7+3` : returns in 10.
  - `expr $a + 3` : returns 3 plus the value of variable *a*.
  - `expr [command]/5` : returns the result of **command** divided by 5.
  - `expr log($b)` : returns the natural logarithm of the value of variable *b*.

- **set** *variable value*

  The **set** command sets the variable *variable* to the value *value*. If value is omitted it returns the value of the variable. Thus `[set variable]` is equivalent to `$variable`. Examples:

  - `set a 3` : sets *a* to 3.
  - `set b $a+4` : sets *b* to 3+4 (and not 7).
  - `set b [expr $a+4]` : sets *b* to 7.

- **unset** *variable1 variable2 ...*

  The **unset** command deletes all the variables that are specified as arguments to it.

- **incr** *variable ?count?*

  Increment *variable* by *count*, by 1 if *count* is not specified.

- **eval** *args*

  Evaluates its arguments and returns the result. Useful when you need a second round of evaluations.

### 2.2.2 List commands

Tcl provides a data structure called *list*, which is nothing but a list of strings. There are numerous commands that manipulate lists, we have selected a few of them:

- **list** *arg1 arg2 ...*

  The **list** command takes all of its arguments and makes a list out of them, and returns the list. Even if any of the arguments is a list itself, it still will be a single element in the new list.

- **concat** *arg1 arg2 ...*

  Similar to **list**, but **concat** concatenates multiple lists together, thus arguments that are lists have their elements separated.

- **lindex** *list i*

  Returns the *i*th element of *list*.

- **llength** *list*

  Returns the number of elements in *list*.

- **lappend** *listVariable arg1 arg2 ...*

  Appends elements to an existing list.

### 2.2.3 Control flow commands

Control flow, like anything else, is handled by commands in Tcl, many of which should be familiar to experienced programmers. The most important ones are:

- **if** *expr* **then** *body1* **elseif** *expr2 body2 ...* **else** *bodyn*

  This is how **if** command is implemented in Tcl. *expr* is an expression which will be evaluated by **if** (thus, you don't need to specify `[expr ...]`), and if it is true, *body1* will be executed as a Tcl script, if it is false, testing will go on. Don't forget to quote the bodies with braces, otherwise they will be evaluated before **if** gets a chance to decide which one to execute. It is also a good idea to put the expressions in braces to prevent any surprises (such as when including spaces in the expression). Don't forget to put spaces between the closing and opening braces, since this is how Tcl breaks up the command into "words". Example:

  ```
  if { $x < 0 } {
    set $abs [expr -$key]
  } else {
    set $abs $key
  }
  ```

- **while** *expr body*

  The **while** command evaluates *expr* and executed the *body* repeatedly until *expr* becomes false. It is crucial that you supply the *expr* in braces, otherwise it will get evaluated once before the while command is executed, and you'll never get out of the loop. As usual, *body* must be supplied within braces, as well and the space between the two arguments can't be omitted.

- **foreach** *variable list body*

  The **foreach** command assigns *variable* the value of each element in the *list* and executes the *body*.

- **for** *initial expr final body*

  The **for** command first executes *initial* as a script, and then executes *body* followed by *final* until *expr* becomes false. Don't forget to put all arguments in braces.

- **source** *file*

  Executes *file* as a Tcl script and returns the result of last command in that file.

- **exec** *command*

  Executes the shell program *command* and returns the output.

- **exit**

  Terminates the current script.

### 2.2.4 Procedures

You can define your own commands as Tcl procedures. Once defined, a procedure behaves like any other command. You can define your procedures using:

- **proc** *name args body*

  The **proc** command is used to define procedures. *name* is the name by which the procedure (command) will be referred. *params* is a list of parameters. If the last parameter is the special **args** then all parameters not assigned to the ones before will be lumped as a list to **args**. *body* is the procedure body. The example below reverses the order of its arguments and returns a list:

  ```
  proc reverse {args} {
  # Start with empty list
    set result {}
    for {set i [expr [llength $args]-1] } {$i>=0} {incr i -1} {
      lappend result [lindex $args $i]
    }
    return $result
  }
  ```

# Chapter 3

# DOPDEES commands

As with any Tcl application, DOPDEES adds a few commands to Tcl. When you use DOPDEES, in addition all Tcl commands, you may use the commands described in this section.

DOPDEES commands expect certain arguments to be of a certain type. We have used some conventions for the arguments of commands throughout this manual. They are listed in Table 3.1. None of the delimiter characters may actually be typed, they are just used to indicate the type of the argument in an easy manner.

| Argument | Meaning |
|---|---|
| `|number|` | A floating point number (scalar) |
| `#field#` | A field name |
| `<evaluable>` | An evaluable |
| `%region%` | A region name |
| `"string"` | A string |
| `literal` | Word must be typed literally |

Table 3.1: Conventions for arguments used throughout this manual.

Note that whenever a DOPDEES command expects an evaluable, you may use any of the following:

- A function that you explicitly specify (with the **func** command, see below)
- A valid field name in the active region.
- A floating point number.

Every structure has a region called “Ambient”, which surrounds all regions in the structure. That is, the leftmost and rightmost regions in a structure are called “Ambient”. In “Ambient” no grid or fields can be defined, but functions can be defined. The functions defined in region “Ambient” can be called from any region.

## 3.1 The `structure` command

The `structure` command is used for actions related to the current structure in memory. The `structure` can be abbreviated down to `stru` without conflicting with other commands. These are the actions that can be performed by the `structure` command:

- `structure load "filename"`

  Loads the structure file denoted by `"filename"` into the memory (no quotes are actually necessary). The syntax of a structure file is described in the appendix. Returns an error message if an error occurs, otherwise returns nothing.

- `structure save "filename"`

  Saves the current structure in memory to the file denoted by `"filename"`. Returns an error message if an error occurs, otherwise returns nothing.

- `structure clear`

  Deletes the current structure in the memory.

## 3.2 The `region` command

The `region` command is used for creating, selecting and performing various operations on regions. Here are its options:

- `region create %region%`

  Creates a new region named `%region%` to the right of the current region. The new region is also made the active region.

- `region remove`

  Removes the current region from the structure. Ambient is made active.

- `region select %region%`

  Makes the region denoted by `%region%` active. All functions and operators until the next `region select` command will apply to the active region.

- `region current`

  Returns the name of the current region.

- `region neighbors`

  Returns a list of the names of the regions surrounding the current region.

- `region list`

  Returns a list of regions in the structure.

- `region eval <eval>`

  Returns information about the evaluable `<eval>` in the current region. If the evaluable has a time independent value, this value is returned. If the evaluable is a field, the name of the field is returned. If the evaluable is a function, the formula for this function is returned. Very useful in debugging the program.

## 3.3 The `field` command

The `field` command is used for creating, deleting and performing various operations on fields. It acts only on fields in the current region. Here are its options:

- `field create #name# <eval>`

  Creates a field with the name `#name#`, and initializes it to value calculated by the `<eval>`. The fields created previously can be used in the script and the field "x" stands for current *x* coordinate. Any field is local to the active region and will not reappear in the next one.

- `field set #name# {C0 C1 C2 ...}`

  Creates a field with the name `#name#`, and initializes it to the list of values specified. Useful for reading a saved structure.

- `field interpolate #name# lin|log filename`

  Creates a field with the name `#name#`, and initializes it to values interpolated from the table in the file `filename`. The file must list one x and one y value at each line. `lin` or `log` must be specified, which tells whether the interpolation will be linear or logarithmic.

- `field delete #name#`

  Deletes the field named `#name#`.

- `field list`

  Returns a list of fields in the active region.

- `field tabulate <eval1> ...`

  Tabulates the values taken by `<eval>`s. As usual, the `<eval>`s can be field names or functions. If the x coordinates are to be tabulated as the first column, simply specify `x` as the first `<eval>`.

- `field x {|x0| |dx0| |x1| |dx1| ...}`

  Creates a grid in the current region with grid points at `|x|`s and spacings of `|dx|`s. Grid points are added to make the spacing equal to `|dx|`s.The grid spacing is changed smoothly if the specified `|dx|`s are different. IMPORTANT: `|x|`s *must* be in increasing order. `|dx|` cannot be zero.

- `field xset {|x0| |x1| |x2| ...}`

  Creates a grid in the current region with grid points only at `|x|`s.

`field create`, `field set` or `field interpolate` commands cannot be used before the grid is specified. Only one `field x` or `field xset` command may be specifed in a region.

## 3.4 The `func` command

The `func` command is used to specify a function. The syntax is:

```
func "name" arg1 arg2 ...
```

where `arg1`, `arg2`, etc are arguments to the function. It returns the function created. The function will be defined in the active region only, and must be respecified if one wants to use in another region (except when the active region is "Ambient", in which case the specified function can be used in any region).

If `"name"` is omitted, it returns a list of available functions in DOPDEES. The list of available functions can be found in Section 4.3.

Note that whenever a DOPDEES command expects an evaluable, you may use any of the following:

- The result of a `func` command as in `[func "name" args]`
- A valid field name in the active region. The function is then a function returning the value of that field.
- A floating point number. Then the function is a constant function.

The `func` command returns a symbolic name for the defined function. Any given function will be calculated only *once* at each time step, even if it is used in multiple places. To use a function in multiple places, store the result of the `func` command in a variable, and substitute it to every place you want to use the function.

## 3.5 The `op` command

The `op` command is used to specify operators in the current region. The syntax is:

```
op "name" #lhs1# ... arg1 arg2 ...
```

Similar to functions, each operator has an effect only in one region, namely the active region. `#lhs#` ... specify the left hand side field(s) of the operator, and `arg1 ...` are the arguments to the operator.

If `"name"` is omitted, it returns a list of available operators in DOPDEES. The list of available operators can be found in Section 4.1.

## 3.6 The `solver` command

The `solver` command is used to specify the actions and parameters of the PDE solver. The `solver` command may be abbreviated down to `sol` without conflicting with other commands. Here are the actions taken by the `solver` command:

- `solver tolerance |reltol| |abstol|`

  Sets the relative tolerance to `|reltol|` and absolute tolerance to `|abstol|` (optional). The default relative tolerance is to 0.01.

- `solver engine "ODE_engine"`

  Specifies which integration engine to be used. The default engine is `dvode-nj`. Currently the following engines can be specified:

  - `rkc`: Runge-Kutta-Chebyshev ODE solver written by B.P. Sommeijer, J.G. Verner and L.F. Shampine. This is a non-stiff integration engine.
  - `lsode`: Livermore Solver of Ordinary Differential Equations written by A.C. Hindmarsh. It uses between Adam's formula for non-stiff problema and Backward Differences formula for stiff problems. Four variants of this engine have been implemented in DOPDEES:
    - `lsode-ns`: Non-stiff `lsode`. Uses Adam's fomula.
    - `lsode-nj`: Stiff `lsode` with numerical evaluation of the (banded) jacobian.
    - `lsode-aj`: Stiff `lsode` with analytical evaluation of the (banded) jacobian. This requires more coding than `lsode-nj`, but is faster for systems with more than 5 fields in at least one region.
    - `lsode-fj`: Stiff `lsode` with numerical evaluation of the full jacobian. This is very slow, but must be used when an `odefunc` function is used.
  - `dvode`: Variable-coefficient Ordinary Differential Equation Solver written by P.N. Brown, A.C. Hindmarsh and G.D. Byrne. Again, four variants of this engine have been implemented in DOPDEES:
    - `dvode-ns`: Non-stiff `dvode`.
    - `dvode-nj`: Stiff `dvode` with numerical evaluation of the (banded) jacobian.
    - `dvode-aj`: Stiff `dvode` with analytical evaluation of the (banded) jacobian. This requires more coding than `dvode-nj`, but is faster for systems with more than 5 fields in at least one region.
    - `dvode-fj`: Stiff `dvode` with numerical evaluation of the full jacobian. This is very slow, but must be used when an `odefunc` function is used.

- `solver run |start_time| {timelist} {script}`

  This command actually starts the PDE solver and runs it from the time specified by `|start_time|` to each time in the list `{timelist}`. If a `{script}` has been specified, it will be run after every run to times in the `{timelist}`. This is useful for extracting intermediate results. Before the `{script}` is executed, the Tcl variable *t* is set to the current time.

## 3.7 Supporting commands

The following set of commands are actually not defined by DOPDEES, but by an initialization script, which is run every time DOPDEES is invoked. They support native DOPDEES commands for increases in productivity.

### 3.7.1 The `writetofile` command

The `writetofile` command writes a string to a file. The syntax is:

```
writetofile filename string
```

Here `filename` is the name of the file to be used (the file is created, or if it exists it is overwritten), and `string` is the string. Typical usage for this command is to write a profile, returned by `field tabulate` command, to a file:

```
writetofile cb.out [field tabulate x CB]
```

### 3.7.2 The `integrate` command

The `integrate` command is used to calculate the integral over the spatial coordinate of a profile. This is useful for calculating the total dose. The syntax is:

```
integrate profile |numcomp|
```

Here, `profile` is the output of a `field tabulate` command. `|numcomp|` is the number of expressions in the profile, and if omitted, is assumed to be 1 (the profile of a single variable). Typical usage:

```
puts [integrate [field tabulate x CB]]
puts [integrate [field tabulate x CI CV] 2]
```

### 3.7.3 The `arrhenius` command

For quantities which have an Arrhenius type dependece on temperature, the `arrhenius` command may be used:

```
arrhenius PRE ENERGY ?UNIT?
```

The `arrhenius` command will calculate the value of the quantity by assuming an Arrhenius dependence on temperature with `PRE` being the pre-factor and `ENERGY` being the activation energy in eV. The command assumes that a global variable called `TEMPK` has been set to the temperature in Kelvin. `UNIT` is optional and if omitted the quantity is assumed to be unitless (i.e. having units of 1).

# Chapter 4

# Operator and Function Reference

## 4.1 Bulk operators

- `op func #field# <function> +|-`

  `funcop` adds or subtracts the value returned by `<function>` to the residual of `#field#`. It can be used to built arbitrary zeroth order operators.

- `op diff #C# <D> <F>`

  `diff` is the diffusion operator (second order spatial derivative). It adds the following to the residual of $C$ at grid point $i$:

$$\frac{(D_{i+1}+D_i)\frac{F_{i+1}-F_i}{x_{i+1}-x_i}-(D_i+D_{i-1})\frac{F_i-F_{i-1}}{x_i-x_{i-1}}}{x_{i+1}-x_{i-1}}$$

- `op ddx #C# <F> <kk>`

  `ddx` is the first spatial derivative operator. `<F>` is the function to be taken the derivative of, and `<kk>` is a function that will multiply the evaluated derivative.

- `op upwind #C# <F> <v>`

  `upwind` is another version of the first spatial derivative operator. It adds the follwing to the residual of `#C#`:

$$-v\frac{\partial F}{\partial x}$$

  A backward difference is used when $v > 0$, otherwise a forward difference is employed.

- `op cluster #fieldD# #fieldC# <Css> <k> |Cmin|`

  Implements a one-moment clustering model of dopant `#fieldD#` into cluster `#fieldC#`. If the concentration of `#fieldD#` is larger than the solid solubility (`<Css>`), that is, the clusters are forming and growing, then the following is added to the residual of `#fieldD#` and subtracted from the residual of `#fieldC#`:

$$k\left((C_D - C_{\text{ss}})C_C + 2(C_D - C_{\text{ss}})^2\right)$$

If $C_D < C_{\text{ss}}$, that is if the clusters are dissolving the second term is set to zero. `Cmin` denotes the cluster concentration, below which the model will be turned off (to avoid numerical problems when all clusters have been desolved).

## 4.2 Interface operators

Interface operators have the following generic syntax:

```
op operator_name "other_region" #field# arg1 arg2 ...
```

Obviously, one needs to specify on what interface the operator will be effective. `"other_region"` is the name of one of the neighboring regions of the current region, and the operator will apply to the interface of current region with `"other_region"`.

- `op mixed "other_region" #field# <Cint> <Ceq> <sigma>`

  Specifies a mixed boundary condition of the type:

  $$-D \left.\frac{\partial C}{\partial x}\right|_{x=\text{interface}} = \sigma(C^{\text{int}} - C^{\text{eq}})$$

  This operator adds the following to the residual of `#field#` at the interface of current region with `"other_region"` ($\Delta x$ is the grid spacing at the interface):

  $$-\frac{\sigma}{\Delta x}(C^{\text{int}} - C^{\text{eq}})$$

- `op transfer "other_region" #field1# #field2# <C1> <C2> <kk>`

  Boundary transfer operator at the boundary of current region and `"other_region"`. Note that these two regions must have a common interface. It first calculates the following quantity:

  $$R = -kk(C_1 - C_2)$$

  where $C_1$ is the value of `<C1>` at the interface in the current region and $C_2$ is value of `<C2>` at the other side of the interface. `<C1>` must be an evaluable in the current region and `<C2>` an evaluable in the neighboring region. `|kk|` is a multiplier.

  The operator adds $R/\Delta x^+$ to the residual of `#field1#` in the current region, at the interface, and subtracts $R/\Delta x^-$ from the residual of `#field2#` in the other region, at the interface. $\Delta x^+$ and $\Delta x^-$ denote the grid spacing on the current region side and the other region side of the interface, respectively.

## 4.3 Functions

Currently, the following functions have been implemented:

- `func sum <F1> <F2> ...`

  Returns the sum of `<F>`s.

- `func prod <F1> <F2> ...`

  Returns the product of `<F>`s.

- `func div <F1> <F2>`

  Returns the quotient of `<F1>` over `<F2>`.

- `func power <F> |power|`

  Returns `<F>` to the power `|power|`, if `<F>` is positive, zero otherwise.

- `func sqrt <F>`

  Returns the square root of `<F>`, if `<F>` is positive, zero otherwise.

- `func log <F>`

  Returns the natural logarithm of `<F>`, if `<F>` is positive, zero otherwise.

- `func exp <F>`

  Returns the exponential of `<F>`.

- `func min <F1> <F2>`

  Returns either the minimum of `<F1>` or `<F2>`.

- `func max <F1> <F2>`

  Returns either the maximum of `<F1>` or `<F2>`.

- `func time`

  Returns the current integration time.

- `func carrierconc <Cnet> |ni|`

  Returns electron concentration divided by intrinsic carrier concentration ($n/n_i$). Given the net donor concentration (`<Cnet>`) and intrinsic carrier concentration (`|ni|`), it returns the following value:

  $$(C^{\mathrm{net}}/2n_i) + \sqrt{(C^{\mathrm{net}}/2n_i)^2 + 1}$$

- `func diffusivity <nni> |D0| |D-| |D+| |D=|`

  Returns the diffusivity calculated according to the following formula. The function assumes that $np = n_i^2$.

  $$D^0 + (n/n_i)D^- + (p/n_i)D^+ + (n/n_i)^2 D^=$$

- `func arrh <pre> <energy> <temp>`

  Returns an Arrhenius function, depending on the Kelvin temperature (`<temp>`). The functions returns the following value ($k_B = 8.62 \times 10^{-5}\,\mathrm{eV}$):

  $$C_{\mathrm{pre}} \exp\left(\frac{E}{k_B T)}\right)$$

# Appendix A

# A word about the units

DOPDEES doesn't have any sense of *units*. Thus, the user must use a consistent set of units for all variables. The input grid must also be specified in the unit system of choice. Usually, it is best to choose a standard unit system, such as m-kg-s (MKS) system.

For systems in VLSI processing, the standard unit system is cm-s. Thus, concentrations are given in $\mathrm{cm}^{-3}$, diffusivities in $\mathrm{cm}^2/\mathrm{s}$ and distances (including the input grid) should be given in cm. DOPDEES doesn't do a $\mu$m to cm conversion like some other simulators.

However, this system of units results in a great discrepancy in the order of magnitudes of the resulting variables. The concentrations are up to the order of $10^{22}$, but diffusivities are down to the order of $10^{-18}$, such that there are 40 orders of magnitude difference. This may result in numerical problems if a concentration and a diffusivity are used as two fields in the same file. The reason for this huge difference in order of magnitudes is that cm is not an appropriate distance unit for VLSI systems. Almost all processes take place in distances on the order of $0.1\mu$m in todays VLSI processes. The appropriateness of s as a unit can be discussed, as the time range for processes is anywhere from 1s to 2 days. But it is certainly advisable to use $\mu$m as a unit in VLSI systems. Thus, concentrations should be given in $\mu\mathrm{m}^{-3}$, diffusivities in $\mu\mathrm{m}^2/\mathrm{s}$ and distances (including the input grid) in $\mu$m. This closes the order of magnitude difference from 40 down to 20.

Included with DOPDEES distribution is a package called UNITS, which has simple commands for resolving unit discrepancies. Please read the documentation for this package.