DOPDEES

(Dial an Operator Partial Differential Equation Evaluator and Solver)

User's Guide and Reference Manual

Alp H. Gencer

Version: 97.221 (August 09, 1997)

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 and this is why it comes with operators suited for such systems. 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.
- The concept of "modules". Modules are collections of equations and parameters that describe a certain model.

Shortcomings

- Inheritly 1-dimensional (although can theoretically be expanded to more dimensions, it is very difficult to do so).
- No support for moving boundaries or adaptive griding.
- 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 Installation

What you will need

DOPDEES is written in C++ and uses Tcl. Thus you will need:

- Tcl library version 7.5 or higher and tcl.h include file.
- A C++ compiler that is capable of doing automatic template instantiation. We have tested only Delta's `DCC` for SGI machines and GNU software's `g++` version 2.7.x with the repository patches applied. Apparently compilers from Borland and other companies are also templates-capable.

If `g++` on your system does not have the repository patches applied, which is most likely the case, you have to recompile `gcc` and `g++` on your system. To do this, obtain `gcc` source code from `ftp://prep.ai.mit.edu/gnu/gcc.2.7.2.2.tar.gz`, apply the patch which is supplied with DOPDEES code as `mathlib/gcc-2.7.2-repo.gz`, and compile following the instructions in `gcc` distribution. gcc (and g++) 2.8.x will have the patches built in.

Editing the Make.local file

Edit the file `DOPDEES/Make.local`. If your system architecture is listed there, you don't need to do anything, except editing `DOPDEES` and `MATH++` (and `KPM`, if you want to include KPM). If your host type is not listed there (to find out your host type enter the command `echo $HOSTTYPE`), create a new section for it, and tell what compiler (`CXX`), compiler flags (`CXXFLAGS`), dependency flags (`DEPFLAG`), Fortran flags (`FFLAGS`) and Fortran libraries (`FTNLIB`) to use. Also specify the location of Tcl library (`TCLLIB`) and Tcl include file (`TCLINC`).

Compilation

Type `gmake` in the topmost DOPDEES directory and the program should compile. Please use GNU's `gmake` utility as the make files are incompatible with some versions of `make`. If things break, do a `gmake clean`, another `gmake` send the output to me.

The executable will be `DOPDEES/bin/dopdees`. It is advisable that you add the `DOPDEES/bin` directory to your path because this directory also includes some conversion scripts to and from Alamode and SUPREM file formats.

The manual will be latex'ed as well, and can be found in the `DOPDEES/doc/manual.dvi` file. Included with DOPDEES is a software package called UNITS, for simple unit handling in Tcl. The documentation for this package can be found in `DOPDEES/UNITS` directory.

Version Numbering

DOPDEES is in continuous evolution, thus things change quite rapidly. Usually, you may expect your input files from older versions to work with the newer ones, but occasionally some operators or functions are modified with regard to parameters. Thus, no backward compatibility can be guaranteed for now.

Version numbers for DOPDEES are of the form `yy.ddd`, where `yy` denotes the year of the release and `ddd` denotes the day (within the year) of the release. For example, version 97.001 was released the first day of 1997.

1.3 Invoking DOPDEES

DOPDEES can be invoked in either interactive mode or can be made to process a script. 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.

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.

There are two ways to make DOPDEES to process a script: Either invoke DOPDEES with the script as the argument, or add a line like `#!/full/path/to/dopdees` as the first line in your script and make your script executable. This will work only on UNIX systems that support it.

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
<function>	A function/field/scalar
%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 a function, 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.

Fields can be single valued (that is, have a single value at each grid point), or vectorial (that is, have multiple components at each grid point). Components of vectorial fields can be accessed as `name(idx)` where `name` is the field name, and `idx` is the component number. Component numbers start from 0.

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.

- `structure regions`

  Returns a list of regions in the current structure.

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 called `%region%` as the rightmost region in the current structure. You cannot insert a new region between two old regions. The new region is also made the active region.

- `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 grid |x| |dx|`

  Inserts a grid point at `|x|` in the active region with spacing `|dx|`. The grid spacing is changed smoothly if the specified `|dx|`'s are different. IMPORTANT: In consecutive `region grid` commands the specified `|x|`'s *must* be in increasing order. Thus, after specifying two grid points, you cannot insert new ones between them. `|dx|` cannot be zero. If both `|x|` and `|dx|` are omitted, a list of grid points in the active region is returned.

- `region field #name# {script} |atol|`

  If arguments are omitted, returns a list of fields in the active region. With arguments, creates a field with the name `#name#`, and initializes it to value calculated by executing the `{script}`. If the specified field name is a vector, then a vectorial field is created, and `{script}` must return a list of values, rather than a single one. The fields created previously can be used in the script (and be referred to as `$field`), and `$x` stands for the current x coordinate. Examples:

  - `{expr 1}`: initializes the field to a constant value 1.
  - `{expr 1e20 - 4e28*pow(($x-0.5e-4),2}`: initializes to a parabola.
  - `{expr 1e20 * exp(-$x/0.2e-4)}`: initializes to a decaying exponential.

  The `interpolate` command (see Section 3.6) can be used to initialize fields with values interpolated from a file. Any field you specify is local to the active region and will not reappear in the next one.

  The last parameter, `|atol|` denotes the absolute tolerance associated with this field, and if ommitted is assumed to be 1. To use absolute tolerances that are different for each field, don't specify and `|atol|` argument in the `solver tolerance` command.

- `region tabulate {expr1} ...`

  Tabulates the values taken by `expr`'s against grid coordinates. `expr1`, `expr2`, etc. are expressions involving fields in the active region, which can be referred using variable substitution (as in `$field`). The command will evaluate the each expression at each grid point and return a table, with the first column being the x value and subsequent columns being the values of the expressions. The expressions may be any valid Tcl expression, enclosed in braces. A single field name (as in `{$name}` will evaluate to the value of that field.

- `region evaluate func1 ...`

  Same as `region tabulate`, but evaluates function values instead of simple expressions of solution fields. The command will evaluate the each function at each grid point and return a table, with the first column being the x value and subsequent columns being the values of the functions. The function may be any valid DOPDEES function defined with the `func` command (see below).

- `region current`

  Returns the name of the current region.

- `region neighbor "side"`

  Returns the name of the neighboring region at the `"side"` side of the current region.

There are a couple of points to be careful when creating a new structure:

- `region grid` command cannot be used before selecting a region, or after specifying fields in a region.

- `region field` command cannot be used before the grid for a region is *completely* specified.
- If you use only one `region grid` command for a region, then that region will contain only a single grid point.

3.3 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.

If `"name"` is omitted, it returns a list of available functions in DOPDEES. If `"name"` is `clear` then all functions defined until now will be erased. The list of available functions can be found in Section 4.3.

Note that whenever a DOPDEES command expects a function, 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.

3.4 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. If `"name"` is `clear` then all operators defined until now will be erased. The list of available operators can be found in Section 4.1.

3.5 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. IMPORTANT: If an `|abstol|` parameter is specified then this absolute tolerance is used for *all* fields and any absolute tolerances specified by `region field` commands will be ignored.

- `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.

3.6 The `interpolate` command

The `interpolate` command creates an "interpolator" which can be used to interpolate field values from a tabular listing in a file. It has the following syntax:

```
interpolate "name" lin|log "filename" |numcomp|
```

Here, `"name"` is the name of the interpolator to be created, `"filename"` is the name of the file where the tabular data resides, and `|numcomp|` is the number of components, if the field is vectorial (may be omitted if the field is scalar). `lin` or `log` must be specified, which tells whether the interpolation will be linear or logarithmic.

After an interpolator is created with the `interpolate` command, it can be called with the given name as `name |x|`. For the given x value, it will return an interpolated list of y values. This list can be passed directly to the `region field` command, making the interpolator a good choice when one wants to initialize fields with values interpolated from a table.

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 `region tabulate` or `region function` commands, to a file:

```
writetofile cb.out [region tabulate {$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 `region tabulate` or `region function` 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 [region tabulate {$CB}]]
puts [integrate [region tabulate {$CI} {$CV}] 2]
```

Chapter 4

Operator and Function Reference

4.1 Bulk operators

- `op funcop #field# <function>`

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

- `op diff #C# |D|`

  `diff` is the *constant diffusivity* diffusion operator. If you need a non-constant diffusivity use `gradsq`. `diff` adds the following to the residual of C at grid point i:

$$D\,\frac{\frac{C_{i+1}-C_i}{x_{i+1}-x_i}-\frac{C_i-C_{i-1}}{x_i-x_{i-1}}}{\frac{1}{2}(x_{i+1}-x_{i-1})}$$

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

  `gradsq` is a more flexible version of the diffusion operator. This operator treats non-uniform diffusivities correctly. 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 bimole #fieldA# #fieldB# <CA> <CB> |eqA| |eqB| <kR>`

  Specifies a reaction of type:

$$A+B\Leftrightarrow\emptyset$$

  `|eqA|` and `|eqB|` are equilibrium concentrations of `#fieldA#` and `#fieldB#`, respectively. `<CA>` and `<CB>` are functions returning their actual (effective) concentrations respectively. `<kR>` is the rate of recombination. This operator adds the following to the residual of *both* `#fieldA#` and `#fieldB#` (and thus needs to be specified only *once* in the input file:

$$-k_{\mathrm{R}}(C_AC_B-C_A^*C_B^*)$$

- op k_fg_h #fieldF# #fieldG# #fieldH# |Keq| |k|

  Specifies a reaction of type:

  $$F + G \Leftrightarrow H$$

  with |Keq| being the equilibrium concentration, and |k| being the forward rate (having units of 1/time). This operator subtracts the following forward rate from the residual of #fieldF# and #fieldG and adds the same to the residual of #fieldH#:

  $$k(K_{\text{eq}}C_F C_G - C_H)$$

- op k_fg #fieldF# #fieldG# <CF> <CG> <k>

  Specifies a reaction of type:

  $$F \Leftrightarrow G$$

  <CF> and <CG> are functions returning the actual (effective) concentrations of #fieldF# and #fieldG# respectively and <k> is a function returning the rate constant. This operator adds the following to the residual of #fieldG# and subtracts the same from the residual of #fieldF#:

  $$k(C_F - C_G)$$

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

  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.

4.2 Interface operators

Interface operators have the following generic syntax:

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

As one can observe, one needs to specify on what side of the current region the operator will be effective. "side" can be l or t for the interface that is at the top or left side of the active region, or r or b for the other side. Note that the side is specified relative to the *region* and not the interface.

- op mixed #field# "side" <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 (Δx is the grid spacing at the interface):

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

- op segreg #field1# "side" #field2# <C1> |mseg| |k|

  Specifies segregation at the boundary. It first calculates the following quantity:

$$R = -k\left(C_1 - \frac{C_2}{m_{\text{seg}}}\right)$$

  where C_1 is the concentration at the interface in the current region (given by <C1>) and C_2 is the concentration of #field2# at the other side of the interface. Naturally, #field1# and #field2# should be the same if the field names don't change from region to region. |mseg| is the segregation coefficient and |k| is the transfer rate.

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

4.3 Functions

Currently, the following functions have been implemented:

- func prod <F1> <F2>

  Returns the product of <F1> and <F2>.

- func div <F1> <F2>

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

- func sum |C1| <F1> |C2| <F2>

  Returns the sum of products as |C1|<F1> + |C2|<F2>.

- func pow <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 <F1> or <F2> whichever is smaller.

- func xcoord

  Returns the x-coordinate of each grid point.

- func time

  Returns the current integration time.

- func odefunc %region% #field#

  Returns the value of #field# in region %region%, first grid point. Useful for solving ODEs in one region and then using their values in other regions. NOTE: You CAN'T use engines lsode-nj, lsode-aj, dvode-nj and dvode-aj when using this function.

- 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 arrh |pre| |energy| <temp>

  Returns an Arrhenius function, depending on the Celsius temperature (<temp>). Useful in temperature ramp simulations. The functions returns the following value (k=8.62×10^{-5} eV):

$$C_{\mathrm{pre}} \exp\left(\frac{E}{(273 + \vartheta) * k}\right)$$

- 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 efieldfactor <Cnet> |ni|

  Returns the diffusivity enhancement due to an electric field drift component, given the net donor concentration and intrinsic carrier concentration. It returns the following value:

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

Appendix A

Format of the structure file

The structure file has a strict, yet simple format, which is used both for input and output. Under normal circumstances the user will not need to be concerned with the structure of this file, as DOPDEES provides commands to manipulate structures. It is included here merely for reference.

The file is pretty linear and has the following general shape:

```
s numregions
r Region1 numfields numgrids
x-values
....
f Field1
field1-values
....
f Field2
field2-values
....
r Region2 numfields numgrids
.........
```

There may be no deviation from this strict syntax. The following single letter commands are recognized:

- `s |numregions|`: Starts a structure. Must be the first command in the file.
- `r %region% |numfields| |numgrids|`: Start a new region with the name `%region%`.
- `f #field#`: Starts a new field. If the field name has the form `field(3)` this means that it will have 3 components, and thus be a vector.

As can be observed, the regions can contain different number of fields, and grid spacing need not be uniform. The x's must be in increasing order, but may be discontinuous from one region to another.

Appendix B

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 cm^{-3}, diffusivities in cm^2/s and distances (including the input grid) should be given in cm. DOPDEES doesn't do a μm 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 0.1μm 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 μm 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 μm. 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, which can be found in the directory `DOPDEES/UNITS`.