An easy guide for KPP

Release 0.1

Tomas Chor

Nov 15, 2016

Contents:

INTRODUCTION

This is an unofficial guide is aimed at providing additional information not covered by the official KPP manual. We focus on the latest version of the software, release 2.2.3, which can be freely downloaded at its official webpage.

The source for the documentation can be found at its github page and the online html version can be accessed here.

We recommend any person reading this guide to keep a copy of the original manual (which can be downloaded here) since this guide is not meant to replace the original manual, but to supplement it.

TWO

ABOUT BASH

The official KPP manual is entirely based on Unix Shell, which is command language which most of the Linux distrobutions use to interact with the system without a Graphical User Interface. The manual, however, assumes a non-trivial knowledge of this tool, which makes it difficult for users not experienced with terminals and command line interfaces (which includes bash, C shell, MS-DOS, PowerShell, ksh etc.) to install and run the simulations effectively. The approach adopted in this guide will be to go through the steps necessary to compile and run KPP, as stated in the manual, but taking the time to explain them a little better how to do them, and what exactly it is that they do.

We will first go over a few basic notions necessary to understand what is going to be done in the guide. If you are familiar with the concepts of system shells, you may skip the next sections.

What is Bash?

But what is a shell? A system shell is the name what computer engineers use to refer to the outer layer of an Operaional System (OS). It is said outter layer because it separates the user (you) from the core of your OS. So it separates you from the intricate group of codes that ultimately governs your machine and lets you interact with your computer using a human-readable language (and not, for example, binary!). So a shell is a bridge between you and your machine.

These shells can be either graphical shells (called Graphical User Interface, GUI, just like what you use during mundane tasks such as browsing the web and reading a PDF document) or text shells (also called terminals or Command Line Interface, CLI). Graphical shells are easier and extremely intuitive (most people use the mouse in a GUI and never needed to be told how to do it), but they are very limited. Basically all you can do is click on buttons that were previously programmed to to some task and input text.

Texts shells (terminals), however, are extremely powerful. You can do virtually anything with your computer using them. That comes to the cost of terminals not being intuitive at all. Since KPP is a complicated code for which there is no graphical interface, we need to use a terminal to compile ("install") and run it, simply because this task requires a more powerful tool then your mouse.

Bash (acronym for Bourne Again Shell) is a kind of Unix Shell used by most of the Linux systems and some Mac OSs. Some other shells can be used to perform the same tasks (the KPP manual itself also gives some commands in C Shell, which is another Unix Shell), but we focus on Bash here because it is the most common and most easily accessible. Most of Linux distributions use it, and some Mac OSs use it as well. Furthermore, it can be natively installed into Windows 10, as we will explain in the next section.

Accessing Bash

To access and use Bash, you either need a Bash emulator or to be in an operational system that supports it natively. Various emulators exist (Cygwin, cmder, MinGW, etc.) but they are not recommended because some of them contain many bugs. If you would like to try those anyway, chances are that it'll work, since we're going to be doing simple

tasks and many people use those. However, running it natively is a always a garantee of no bugs, so (in the spirit of keeping it general) that's why it's the most recommended option for this guide.

We will briefly go through your options for each of the 3 most common operational systems.

From Windows

Windows doesn't support Unix Shells natively by default, so here are the options.

If you're using Windows 10, you can natively install the Ubuntu 14.04 inside your Windows machine with the Windows 10 anniversary update, which is available for every up-to-date Windows 10 computer. Directions to do this are very somple and are given in many places (such as here) so for now we will not explain them in detail. This will give you Bash running natively on Windows. But only works for up-to-date Windows 10 computers.

If you do not have Windows 10, you can either install one of the many Bash emulators for Windows or you can install a Linux virtual machine inside your Windows computer. You can do that using Virtual Box and installing a Ubuntu-based distribution (we recommend installing either a recent version of Ubuntu or Linux Mint 18 (or greater), since these two are most suited for beginners in Linux). Again, directions on how to do this are straightfoward and exist all over the internet, so we will not spend time on steps on how to do that.

From Mac OS

If you have a Mac, you might already have Bash natively installed, since all Macs are based on Unix. To find out what your shell is, you need to open a terminal application (generally under utilities). Then type the command *echo $SHELL* and press enter. If the output of the shell is something ending in Bash, like */bin/bash*, then you're already running Bash. If it ends in something else, like */bin/ksh*, then you're running a different Unix Shell. Most commands should be the same, but if you want to use this shell you might have to translate some (which should be easy with the help of Google).

If you're running another terminal and would like to try Bash, you can either get an Bash emulator for Mac, install a Linux virtual machine (as described in the Windows section) or change your terminal to Bash. The most recomended here is to change your Shell to Bash. Instructions on how to do this are easy and can again be found in many places, including here.

From Linux

If you're running Linux you can open a terminal and run the command *echo $SHELL* to find out if you're running Bash or not. If you're not you can try to keep going with your Shell (some commands may need to be translated) or you can change your default Shell with the *chsh* command. You can find more detailed information on that in many places, such as here.

COMPILING KPP

In this chapter we detail how to successfully download and compile KPP on your system under the Bash environment.

Downloading into your folder

One of the first things to be said is: most of the commands we will use will only work if you're in the right directory (which we will always tell what it is). So when you open a terminal, that terminal is "running" in some directory in your computer. You can find out which directory that is by entering the command `pwd` which stands for "Print Working Directory". That will show you exactly where you are on your computer.

Note: You can also use the `ls` command, which will "list" everything you have on that directory to get a better sense of where you are. Also, you can use the command `tree -d ˋpwdˋ`, which shows you your current directory on top, and the subdirectories in it in a tree-like structure. Try it! This can also be used to make you get a sense of where you are and what directories are "around you".

To change directories, you can use the command `cd`, which stands for "Change Directory". So if you want to go to your downloads directory, you can type `cd Downloads`, or `cd /home/myuser/Downloads` depending on where you are on your terminal (the first is a relative path (to your current location) and the second is an absolute or full path; you can read more about relative and absolute paths here).

Note: Throughout this document, we'll generally use `myuser` to refer to your username in the system. This generally comes right after `/home/` and you should change according to your case. So if your user name is `john` you'd replace `/home/myuser` with `/home/john` in every occasion.

If you prefer to download KPP through its website manually and unpack it somewhere, you'll have to go there with your terminal. So, if I unpack it in my home directory, as soon as I open my terminal I'll have to use `cd /home/myuser/kpp-2.2.3`. This command will only work if the path is correct (it might not work on Windows, for example, which does not have a `/home` location. If you're using Bash on Windows it's better to go with the following alternative.

However, if you're insecure with navigating your directories using your terminal, it's best to do everything via this second, more straightfoward, option. It uses solely commands but it's easier. First, as soon as you open the terminal run the following commands

```
wget http://people.cs.vt.edu/~asandu/Software/Kpp/Download/kpp-2.2.3_Nov.2012.zip
unzip kpp-2.2.3_Nov.2012.zip
cd kpp-2.2.3
```

Line one will automatically download the software to your current directory and line two will unpack it. This will create a new directory with all the contants of the `.zip` file, so the last command line will move to the recently-created directory.

Make sure you're in the correct directory by entering `pwd`, which should show you that you're on the *kpp-2.2.3* directory. You can also type *ls*, which should show you a list of everything that was in the zip file:

```
cflags       drv        int                 Makefile.defs  site-lisp
cflags.guess examples   int.modified_WCOPY  models         src
doc          gpl        Makefile            readme         util
```

Making sure dependecies are installed

Now we are going to set-up the environment to compile KPP. The first step is to make sure that you have the necessary software. These are called the dependencies of a program: it is everything the program needs to be available in the system (softwares, libraries, etc.) before it's installed.

Be sure that FLEX (which is a public domain lexical analizer) is installed on your machine. You can run `flex --version` and if it is installed you should see something like `flex 2.6.0`. If instead you see something like `flex: command not found` then it means that it is not installed and you're going to have to install it by running `sudo apt update && sudo apt install flex` if you're running Linux natively (depending on your Linux distribution) or by manually installing downloading and installing the file if you're emulating (with Cygwin, for example). A quick google search should tell you how to install it easily. Note: if `flex` isn't available for you, you might need to install the Flex-dev package with `sudo apt install flex-devel.x86_64`.

Be also sure that `yacc` and `sed` are installed by typing `which yacc` and `which sed`. If you see something like `/usr/bin/sed` or `/usr/lib/yacc` then they are installed. If you see an error message, then you're also going to have to install it manually. Again, a quick google search should tell you how to do it, although it is very rare that these packages aren't installed.

Note: `flex` and `yacc` have to do with lexical analysis and it's not specially important to know exactly what they do. Suffices to say that they are used internally by the compiler to generate the executable file, but you will never have to use them directly when compiling/using KPP. `sed`, however, is a very useful text manipulation tool, but you also won't need to use it while running KPP.

Telling your system where KPP is

Now that we have the dependecies installed, we need to make sure that your computer knows where KPP is in your system. We do that by altering a file called `.bashrc`. This file is a simple text file (so can you easily open and read it, as you'll see) with some very simple commands. Every time you start a terminal that file is "read" internally by the terminal and executed. So inside that file you can put any command that you could type in the terminal. Thus, generally, if you want to change something in your terminal so that the change takes place every time you start it (so you don't have to re-change it over and over again every time you open it), that's the place to do it. For our purposes we simply need to add a couple of lines. We'll do that step by step.

Note: If you're using a terminal other than Bash the `.bashrc` file will probably have a slightly different name (like `.cshrc` e.g.) and the commands might also differ a bit, but the process and the ideas are the same! You'll just have to probably do some quick googling.

First, in the directory where you unpacked KPP, run the command `pwd` to print the present working directory and copy its output. You'll need this for the next steps.

Now you need to open and edit `.bashrc` which can be done with many programs, it really depends on what is installed for your (or what you would like to install). The best options would be an editor that runs with a GUI. For Windows users the best option is probably `notepad++`, while for Linux users `gedit` is generally the default GUI option. You can try these (and any other GUI plain text editors you know) with the commands `gedit ~/.bashrc`, or `notepad++ ~/.bashrc` and so forth with the others.

If any of those work, great!, you can edit the file in an intuitive GUI editor. If not, you're either going to have to install a GUI text editor, or use Nano by running the command `nano ~/.bashrc`. Nano is a very handy text editor which runs on the terminal itself, however, it's not as eye-pleasing and not as intuitive as the GUI-based ones.

Fig. 3.1: .bashrc example.

If you're forced to use Nano, you should probably read this very quick tutorial to learn how to open, save and close files. It's not as intuitive, but it's very easy.

Once you open `.bashrc`, you're going to see something like Fig. *.bashrc example.* (in this case open with Nano). Don't worry about the lines of code. They're probably going to be different for you and that's OK; it really varies a lot from system to system. You can ignore all those codes and jump to the last line of the file. After the last line you'll include the following

```
export KPP_HOME=$HOME/kpp-2.2.3
export PATH=$PATH:$KPP_HOME/bin
```

except that you should replace `$HOME/kpp` with the output of your `pwd` command. For example, if the output of `pwd` was `/home/myuser/Downloads/kpp-2.2.3` you should write

```
export KPP_HOME=/home/user/Downloads/kpp-2.2.3
export PATH=$PATH:$KPP_HOME/bin
```

After this is done, you are going to save and exit. If you're using any option with a GUI this should be straightforward. With Nano you can save and exit by pressing control X, choosing the "yes" option (by hitting the "y" key) when it asks you to save, and then pressing enter when asked to confirm to name of the file to save to.

Now your terminal will know where KPP is the next times you start it. But for the changes to make effect you need to close this terminal and open another one. So just close the terminal you were working with, open a new one. Now, if everything worked properly, you should be able to type `cd $KPP_HOME` and go automatically to your KPP directory. If this worked, we are ready for the next step, which is telling your system how to compile KPP.

Specifying how to compile

Now we actually compile (which is a way of installing) KPP. First, type `locate libfl.a` and save the output. If there is no output, use `locate libfl.sh` and save the output of that. These commands tell you where the Flex library is, which we assured was installed somewhere in the system during the last section. In my case the output was `/usr/lib/x86_64-linux-gnu/libfl.a`.

Now in your KPP directory, use the same text editor as before to open a file called `Makefile.defs`, which sets how Bash is going to make the executable code for KPP (i.e., it only gives instructions to your computer on how to compile it). So type `gedit Makefile.defs`, or `nano Makefile.defs` and so on, depending on the editor you're using.

Once again, you'll see a lot of lines with comments, and the only lines that matter are those that don't start with `#`. There should be 5 lines like this in this file. The first one starts with `CC`, which sets the C Compiler. In this guide we will use the Gnu Compiler Collection, `gcc`. So make sure that the line which starts with `CC` reads `CC=gcc`.

Next, since we made sure that Flex was installed, make sure the next important line reads `FLEX=flex`. On the third step, set the next variable (`FLEX_LIB_DIR`) with the output we just saved without the last part. So in my case the output saved was `/usr/lib/x86_64-linux-gnu/libfl.a`, so the line will read `FLEX_LIB_DIR=/usr/lib/x86_64-linux-gnu`. You should, of course, replace your line accordingly.

The next two items define some possible extra options for the compilation and extra directories also to include in the compilation. We will don't have to worry about those, unless maybe if we need to debug the program for some reason. Now you can save and close/exit the file.

If we did everything correctly we can compile KPP simply by running the `make` command. Many warnings are going to appear on the screen (that's normal), but as long as no error appears, the compilation will be successful. You can be sure it was successsful by once again running `ls` and seeing that there is now one extra entry on the KPP directory called `bin`:

```
bin           doc        gpl                 Makefile        readme      util
cflags        drv        int                 Makefile.defs   site-lisp
cflags.guess  examples   int.modified_WCOPY  models          src
```

Now let's test it by running `kpp test`. If the output is something like

```
This is KPP-2.2.3.

KPP is parsing the equation file.
Fatal error : test: File not found
Program aborted
```

then we know it worked. This tells you the version of KPP and that it couldn't find any file to work with, which is fine because we didn't give it any yet. If this works, you can skip to the next section.

If, however you get an output similar to `kpp:  command not found...` then chances are that `bin` is a binary executable file, while it should be a directory containing the binary file. This should not happen, according to the

manual, but for some reason it (very) often does. We need simply to rename that executable file and put it a directory called `bin`. This can be done with the followinf command:

```
mv bin kpp && mkdir bin && mv kpp bin
```

Try this command and then try `kpp test` again. You should get the correct output this time, meaning that the system could find KPP successfully. g

RUNNING KPP

Now that KPP is propoerly compiled, we proceed to running the first test case to make sure it works!

The first test case

We now follow the manual and begin running the Chapman stratospheric mechanism as a test case. This will allow us to illustrate some key features when running KPP.

In order to run a simulation on KPP, it needs three things:

- a `.kpp` file (from the KPP directory, type `ls examples` to see some examples of those)
- a `.spc` file (type `ls models` to see some examples of those)
- a `.eqn` file (type `ls models` to see some examples of those)

We begin by creating a directory to run this first test. Let's call this directory `test1` and create it with `mkdir test1`. We go to that directory with `cd test1`. Let's follow the manual and create a file called `small_strato.kpp` with the following contents:

```
#MODEL      small_strato
#LANGUAGE   Fortran90
#DOUBLE     ON
#INTEGRATOR rosenbrock
#DRIVER     general
#JACOBIAN   SPARSE_LU_ROW
#HESSIAN    ON
#STOICMAT   ON
```

You can do this by typing `nano small_strato.kpp` in the `test1` directory, if using Nano, or by using another editor of your choice. Then just paste the content above in the file, save it and exit it.

This file tells KPP what model to use and how to process it. You can learn more about this in the KPP manual, but basically our file is telling KPP to use the `small_strato` model, output the code in Fortran 90 with double precision using the Rosenbrock integrator.

If our changes to `.bashrc` are correct, then KPP should be able to find the correct model, since the `small_strato` model is located in the `models` directory, in the KPP home directory. We test this by running KPP on our recently created file with `kpp small_strato.kpp`.

You should see many lines appear onscreen, the last one of them saying `KPP has succesfully created the model "small_strato"`, which means you were sucessfull. Now if you type `ls`, you'll see many new files:

```
Makefile_small_strato           small_strato.map
small_strato_Function.f90       small_strato_mex_Fun.f90
```

```
small_strato_Global.f90            small_strato_mex_Hessian.f90
small_strato_Hessian.f90           small_strato_mex_Jac_SP.f90
small_strato_HessianSP.f90         small_strato_Model.f90
small_strato_Initialize.f90        small_strato_Monitor.f90
small_strato_Integrator.f90        small_strato_Parameters.f90
small_strato_Jacobian.f90          small_strato_Precision.f90
small_strato_JacobianSP.f90        small_strato_Rates.f90
small_strato.kpp                   small_strato_Stoichiom.f90
small_strato_LinearAlgebra.f90     small_strato_StoichiomSP.f90
small_strato_Main.f90              small_strato_Util.f90
```

Most of them end with a `.f90` extension, which tells us they are Fortran codes. These codes have to compiled into an executable file which is what will actually process and run the kinetic model. So the next step is to compile every one of those code together into one executable and run it.

Let's focus for now on the `Makefile_small_strato`. This tells your computer which Fortran compiler to use to compile, some options and etc. Open the `Makefile_small_strato` file and find where it says

```
#COMPILER = G95
#COMPILER = LAHEY
COMPILER = INTEL
#COMPILER = PGF
#COMPILER = HPUX
#COMPILER = GFORTRAN
```

Each of the lines is a different Fortran compiler, and your computer is only going to see the line that doens't start with a `#`. So, currently, these lines are telling the computer to use the intel Fortran compiler, `ifort`.

If you are using `ifort`, you should leave it as it is. Since `ifort` is paid, chances are you are using another compiler. If this is the case, put the `#` in front of the `INTEL` options and take it out of the line which has the name of your compiler. If you don't know which compiler you have, chances are you have gfortran, which is free and what we will use here. You can also install gfortran with `sudo apt install gfortran`.

So, assuming you're using gfortran, make the lines read

```
#COMPILER = G95
#COMPILER = LAHEY
#COMPILER = INTEL
#COMPILER = PGF
#COMPILER = HPUX
COMPILER = GFORTRAN
```

Save and exit the file.

Now all you have to do is run `make -fMakefile_small_strato`, which will compile your fortran code into an executable using the options we just set. You should see a lot on lines appearing on screen starting with `gfortran` and if no error messages appear the compilation was successful.

Now you'll see many more new files, including one called `small_strato.exe`, which is your executable file (run `ls` again to see that). This is the executable that will actually calculate the concentrations using the model.

To test if it works, run `./small_strato.exe`, which will run the executable. You should see some output on the screen with concentrations, like Fig. *Output concentrations of the first test case.*

If this is the case, then your run was successfull and everything worked well! You just calculated the concetrations of the compounds in the `small_strato` model with the pre-defined initial conditions.
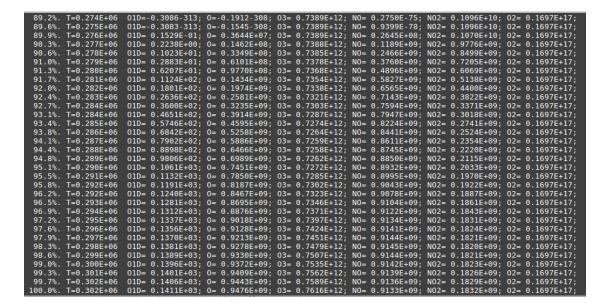
```
 89.2%. T=0.274E+06  O1D=-0.3086-313; O=-0.1912-308; O3= 0.7389E+12; NO= 0.2750E-75; NO2= 0.1096E+10; O2= 0.1697E+17;
 89.6%. T=0.275E+06  O1D=-0.3083-313; O=-0.1545-308; O3= 0.7389E+12; NO= 0.9399E-78; NO2= 0.1096E+10; O2= 0.1697E+17;
 89.9%. T=0.276E+06  O1D= 0.1529E-01; O= 0.3644E+07; O3= 0.7389E+12; NO= 0.2645E+08; NO2= 0.1070E+10; O2= 0.1697E+17;
 90.3%. T=0.277E+06  O1D= 0.2238E+00; O= 0.1462E+08; O3= 0.7388E+12; NO= 0.1189E+09; NO2= 0.9776E+09; O2= 0.1697E+17;
 90.6%. T=0.278E+06  O1D= 0.1023E+01; O= 0.3349E+08; O3= 0.7385E+12; NO= 0.2466E+09; NO2= 0.8499E+09; O2= 0.1697E+17;
 91.0%. T=0.279E+06  O1D= 0.2883E+01; O= 0.6101E+08; O3= 0.7378E+12; NO= 0.3760E+09; NO2= 0.7205E+09; O2= 0.1697E+17;
 91.3%. T=0.280E+06  O1D= 0.6207E+01; O= 0.9770E+08; O3= 0.7368E+12; NO= 0.4896E+09; NO2= 0.6069E+09; O2= 0.1697E+17;
 91.7%. T=0.281E+06  O1D= 0.1124E+02; O= 0.1434E+09; O3= 0.7354E+12; NO= 0.5827E+09; NO2= 0.5138E+09; O2= 0.1697E+17;
 92.0%. T=0.282E+06  O1D= 0.1801E+02; O= 0.1974E+09; O3= 0.7338E+12; NO= 0.6565E+09; NO2= 0.4400E+09; O2= 0.1697E+17;
 92.4%. T=0.283E+06  O1D= 0.2636E+02; O= 0.2581E+09; O3= 0.7321E+12; NO= 0.7143E+09; NO2= 0.3822E+09; O2= 0.1697E+17;
 92.7%. T=0.284E+06  O1D= 0.3600E+02; O= 0.3235E+09; O3= 0.7303E+12; NO= 0.7594E+09; NO2= 0.3371E+09; O2= 0.1697E+17;
 93.1%. T=0.284E+06  O1D= 0.4651E+02; O= 0.3914E+09; O3= 0.7287E+12; NO= 0.7947E+09; NO2= 0.3018E+09; O2= 0.1697E+17;
 93.4%. T=0.285E+06  O1D= 0.5746E+02; O= 0.4595E+09; O3= 0.7274E+12; NO= 0.8224E+09; NO2= 0.2741E+09; O2= 0.1697E+17;
 93.8%. T=0.286E+06  O1D= 0.6842E+02; O= 0.5258E+09; O3= 0.7264E+12; NO= 0.8441E+09; NO2= 0.2524E+09; O2= 0.1697E+17;
 94.1%. T=0.287E+06  O1D= 0.7902E+02; O= 0.5886E+09; O3= 0.7259E+12; NO= 0.8611E+09; NO2= 0.2354E+09; O2= 0.1697E+17;
 94.4%. T=0.288E+06  O1D= 0.8898E+02; O= 0.6466E+09; O3= 0.7258E+12; NO= 0.8745E+09; NO2= 0.2220E+09; O2= 0.1697E+17;
 94.8%. T=0.289E+06  O1D= 0.9806E+02; O= 0.6989E+09; O3= 0.7262E+12; NO= 0.8850E+09; NO2= 0.2115E+09; O2= 0.1697E+17;
 95.1%. T=0.290E+06  O1D= 0.1061E+03; O= 0.7451E+09; O3= 0.7272E+12; NO= 0.8932E+09; NO2= 0.2033E+09; O2= 0.1697E+17;
 95.5%. T=0.291E+06  O1D= 0.1132E+03; O= 0.7850E+09; O3= 0.7285E+12; NO= 0.8995E+09; NO2= 0.1970E+09; O2= 0.1697E+17;
 95.8%. T=0.292E+06  O1D= 0.1191E+03; O= 0.8187E+09; O3= 0.7302E+12; NO= 0.9043E+09; NO2= 0.1922E+09; O2= 0.1697E+17;
 96.2%. T=0.292E+06  O1D= 0.1240E+03; O= 0.8467E+09; O3= 0.7323E+12; NO= 0.9078E+09; NO2= 0.1887E+09; O2= 0.1697E+17;
 96.5%. T=0.293E+06  O1D= 0.1281E+03; O= 0.8695E+09; O3= 0.7346E+12; NO= 0.9104E+09; NO2= 0.1861E+09; O2= 0.1697E+17;
 96.9%. T=0.294E+06  O1D= 0.1312E+03; O= 0.8876E+09; O3= 0.7371E+12; NO= 0.9122E+09; NO2= 0.1843E+09; O2= 0.1697E+17;
 97.2%. T=0.295E+06  O1D= 0.1337E+03; O= 0.9018E+09; O3= 0.7397E+12; NO= 0.9134E+09; NO2= 0.1831E+09; O2= 0.1697E+17;
 97.6%. T=0.296E+06  O1D= 0.1356E+03; O= 0.9128E+09; O3= 0.7424E+12; NO= 0.9141E+09; NO2= 0.1824E+09; O2= 0.1697E+17;
 97.9%. T=0.297E+06  O1D= 0.1370E+03; O= 0.9213E+09; O3= 0.7451E+12; NO= 0.9144E+09; NO2= 0.1821E+09; O2= 0.1697E+17;
 98.3%. T=0.298E+06  O1D= 0.1381E+03; O= 0.9278E+09; O3= 0.7479E+12; NO= 0.9145E+09; NO2= 0.1820E+09; O2= 0.1697E+17;
 98.6%. T=0.299E+06  O1D= 0.1389E+03; O= 0.9330E+09; O3= 0.7507E+12; NO= 0.9144E+09; NO2= 0.1821E+09; O2= 0.1697E+17;
 99.0%. T=0.300E+06  O1D= 0.1396E+03; O= 0.9372E+09; O3= 0.7535E+12; NO= 0.9142E+09; NO2= 0.1823E+09; O2= 0.1697E+17;
 99.3%. T=0.301E+06  O1D= 0.1401E+03; O= 0.9409E+09; O3= 0.7562E+12; NO= 0.9139E+09; NO2= 0.1826E+09; O2= 0.1697E+17;
 99.7%. T=0.302E+06  O1D= 0.1406E+03; O= 0.9443E+09; O3= 0.7589E+12; NO= 0.9136E+09; NO2= 0.1829E+09; O2= 0.1697E+17;
100.0%. T=0.302E+06  O1D= 0.1411E+03; O= 0.9476E+09; O3= 0.7616E+12; NO= 0.9133E+09; NO2= 0.1832E+09; O2= 0.1697E+17;
```

Fig. 4.1: Output concentrations of the first test case.

Understanding and improving on the test case

Now let's understand why our run of `small_strato.exe` was successful and what happened. First, by running the `.exe` file we ran a program that got some initial concentrations of the substances defined for `small_strato` and

FIVE

UNDERSTANDING THE OUTPUT

SIX

INDICES AND TABLES

- genindex
- modindex
- search