An easy guide for KPP

Release 1.0

Tomas Chor

Dec 07, 2016

CONTENTS

1 Introduction **1**
1.1 Contributing 1

2 About bash **2**
2.1 What is Bash? 2
2.2 Accessing Bash 3

3 Compiling KPP **4**
3.1 Downloading into your folder 4
3.2 Making sure dependecies are installed 5
3.3 Telling your system where KPP is 6
3.4 Specifying how to compile 6

4 Running KPP **9**
4.1 The first test case 9
4.2 Understanding the test case 12

5 Modifying and improving the example **15**
5.1 Increasing the length of the simulation 15
5.2 Change in the initial conditions 17
5.3 Modifying the reactions 17
5.4 Creating a model from scratch 19

6 Possible bug fixes **24**
6.1 Can’t fine bashrc file 24

CHAPTER

ONE

INTRODUCTION

This is an unofficial guide aimed at providing additional information not covered by the official KPP manual. Specifically, we aim to make things more comprehensive for the user with few or no experience running software via command lines. We focus on the latest version of the software, release 2.2.3, which can be freely downloaded at its official webpage. We recommend any person reading this guide to keep a copy of the original manual since this guide is not meant to replace it, but as a supplement to it.

In this guide our aim is to teach an inexperienced user to download, compile, and run KPP. Furthermore, besides teaching how to run the example case in the manual, we also instruct the user on how to modify any already-existent model and to create new models.

The source for the documentation can be found at its github page and the online html version can be accessed here. You can also download this guide in pdf here. All of the codes and scripts created here as examples are available in the Github repository.

The directions in this guide have been tested in as broad a range of operational systems as was possible, but some errors are likely to arise when applying them to other systems. If that is the case, feedback is encouraged, either by email, in person, or by creating a Github issue in our page.

This guide was primarily typed and uploaded by Tomas Chor, but has had substantial help from Prof. Suzanne Paulson and Dr. Paul Griffiths, particularly for chemistry-related issues.

Contributing

You can also contribute to this guide yourself. If you find the need to correct, improve or add something, feel free to download/fork the project on Github and modify it. We appreciate if the projects could then be merged back after that (preferably with an updated version tag), but that is entirely up to you. To contribute, you have to install and use Sphinx, which is a very handy and easy-to-use tool designed to build multi-platform documentation effectively.

If you are not familiar with Sphinxs, you should probably read a tutorial (it's very easy!), but a quick way to start is to download the whole thing from Github, add some text to any file with a `.rst` extension and run `make html`. That will create a webpage (like this one) with your modifications that you can open in your browser (open the files with a `.html` extension that were created). So you can just go from there and infer the syntax from what's already written. You can actually learn Sphinx by yourself by doing this and might not even need to read a tutorial.

CHAPTER

TWO

ABOUT BASH

The official KPP manual is entirely based on Unix Shell, which are command languages that Linux distributions 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 that are 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?

First things first: what is a shell? A system shell is the name that computer engineers use to refer to the outer layer of an Operational System (OS). It is said outer 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!). Basically, a shell is a bridge between you and your machine.

These shells can be either graphical shells [1] (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. Besides its popularity among Linux distributions and Mac OSs, it is the only shell (as far as my knowledge goes by the time of writing) that can be natively installed into Windows, as we will explain in the next section.

[1] A note here is that the GUI isn’t generally considered a shell, but that is technically correct given the definition of a shell.

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 they tend to work well for that. However, running it natively is always a guarantee of no bugs, so (in the spirit of keeping it general) we adopt this option throughout this guide, only occasionally giving some remarks on other shells.

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 simple and are given in many places (such as here) so for now we will not explain them in detail. Keep in mind that since this will give you Bash running natively (even though you're on Windows), you'll be able to use all the commands that can be used on Linux, such as package installation commands (`apt install`).

If you a version of Windows older than 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. If you choose the latter (which we strongly recommend instead of the emulator), you can do it 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 straightforward 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. If you want to use this different shell (and you can) most commands should be the same, but you might have to translate a few (which should be easy Googling `bash yourcommand in csh`, for example).

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 recommended 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, and generally use one very simple command called `chsh`.

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.

CHAPTER

THREE

COMPILING KPP

We say we compile a program when we convert all the human-readable code that directly wrote into a binary executable that your machine can actually run directly. In other words, it's like we're installing a program into the computer. In this chapter we detail how to successfully download and compile KPP on your system under the Bash environment.

If from now you encounter any bug that isn't described directly in the text, we refer you to the *Possible bug fixes* section. If you find a fix for an error that isn't anywhere in the guide, we also encourage you to contribute to the guide and include that bug fix in that section. (You can check the Introduction section on how to contribute.)

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 `pwd` ``, 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 emulators, for example, which may place the `/home` directory elsewhere (you can always just google).

However, if you're insecure with navigating your directories using your terminal, it's best to do everything via this second, more straightforward, option. It uses solely commands but it's easier. First, with the terminal open somewhere (anywhere in this case) run the following commands:

```
cd $HOME
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 go to your home directory, and line two will automatically download the software there, while line three will unpack it. This will create a new directory with all the contents of the `.zip` file, so the last command line will move to the recently-created directory, which is now the KPP directory.

Note: This last set of commands can be run from any directory because we first moved to the home directory (in the first line) before downloading and unpacking everything. This was done just to make things easier and more compact, but KPP can be downloaded and run from anywhere in your system, so if for some reason you want to download, unpack and install it somewhere else, feel free to change the first line accordingly.

Make sure you're in the correct directory by entering `pwd`, which should show you the full path to 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 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` or something similar.

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. On the other hand, `sed`, is a very useful text manipulation tool that you might benefit from learning, but you also won't need to use it while running KPP, so feel free to disregard it for now.

Telling your system where KPP is

Now that we have the dependencies 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-set that change over and over again every time), 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.

Now you need to open and edit `.bashrc` from the terminal which can be done with many programs, it really depends on what is installed for you (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 find yourself a text editor with a GUI, 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.

Note: 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
```

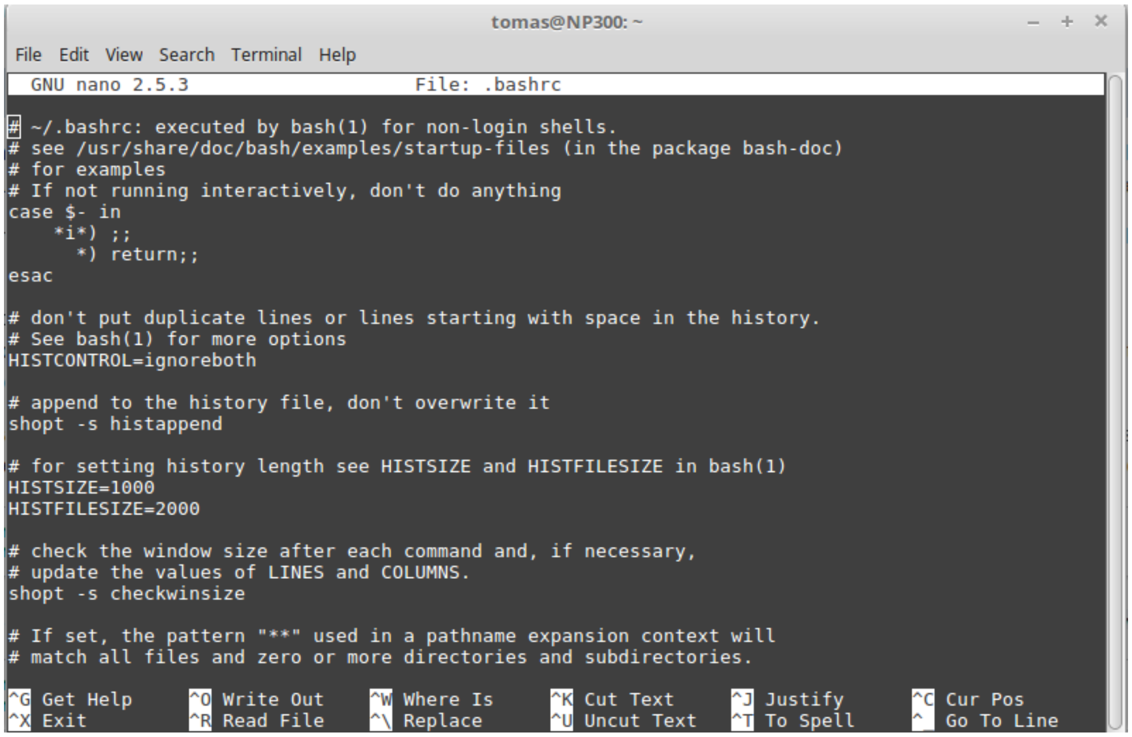
That will work if you followed exactly the previous commands and installed KPP in the home directory. If you didn't you should replace `$HOME/kpp-2.2.3` with the output of your `pwd` command which you just saved.

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



```
  GNU nano 2.5.3                File: .bashrc

# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples
# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "**" used in a pathname expansion context will
# match all files and zero or more directories and subdirectories.

^G Get Help     ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit         ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^_ Go To Line
```

Fig. 3.1: .bashrc example.

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`.

Note: If you prefer to use another compiler, put that one there instead of 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 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:

```
make
```

on the terminal. 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 successful 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 the following command:

```
kpp test
```

If the output after this command 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 following 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.

CHAPTER

FOUR

RUNNING KPP

Now that KPP is properly compiled, we proceed to running the first test case to make sure it works! It's advised to have the official KPP manual along with you during this section.

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 (type `ls $KPP_HOME/examples` to see some examples of those)
- a `.spc` file (type `ls $KPP_HOME/models` to see some examples of those)
- a `.eqn` file (type `ls $KPP_HOME/models` to see some examples of those)

We begin by creating a directory to run this first test. Let's call this directory `test1`. We can create this directory anywhere: even inside KPP's home directory, although, for the sake of simplicity, let's create it in your home directory:

```
cd $HOME
mkdir test1
```

Now let's go to that directory with `cd test1`. Following the manual, let us create a file called `small_strato.kpp` with the following contents:

```
#MODEL      small_strato
#LANGUAGE   Fortran90
#INTEGRATOR rosenbrock
#DRIVER     general
```

You can do this by typing `notepad++ small_strato.kpp` in the `test1` directory, if using Notepad++, or by using another editor of your choice (replace `notepad++` with `gedit` for example). Then just paste the content above in the file, save and exit it.

This file tells KPP what model to use (`small_strato.def`) and how to process it (most importantly for us here, it tells KPP to generate a Fortran 90 code, although it can also generate C and Matlab code). Many other options can be added to this file and you can learn more about them in the KPP manual.

If our changes to `.bashrc` are correct, then KPP should be able to find the correct model, since the `small_strato` model (given by `small_strato.def`) 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 the following lines on your screen:

```
This is KPP-2.2.3.

KPP is parsing the equation file.
KPP is computing Jacobian sparsity structure.
KPP is starting the code generation.
KPP is initializing the code generation.
KPP is generating the monitor data:
    - small_strato_Monitor
KPP is generating the utility data:
    - small_strato_Util
KPP is generating the global declarations:
    - small_strato_Main
KPP is generating the ODE function:
    - small_strato_Function
KPP is generating the ODE Jacobian:
    - small_strato_Jacobian
    - small_strato_JacobianSP
KPP is generating the linear algebra routines:
    - small_strato_LinearAlgebra
KPP is generating the Hessian:
    - small_strato_Hessian
    - small_strato_HessianSP
KPP is generating the utility functions:
    - small_strato_Util
KPP is generating the rate laws:
    - small_strato_Rates
KPP is generating the parameters:
    - small_strato_Parameters
KPP is generating the global data:
    - small_strato_Global
KPP is generating the stoichiometric description files:
    - small_strato_Stoichiom
    - small_strato_StoichiomSP
KPP is generating the driver from none.f90:
    - small_strato_Main
KPP is starting the code post-processing.

KPP has succesfully created the model "small_strato".
```

Note: If you get an error message here, go back a few steps and make sure the `$KPP_HOME` and the `$PATH` variable are set correctly, and be sure that both KPP can be found and the correct model files `small_strato` are in `$KPP_HOME/models`.

If indeed you see this output (or something very similar) it means you were successful in creating the model. Now if you list your files with the `ls` command, 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 90 codes. These codes have to be 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. To do that, let's focus for now on the `Makefile_small_strato`. This is a text file that tells your computer which Fortran compiler to use to compile, which files to use, etc. We need to modify it, so open the `Makefile_small_strato` file (again using your preferred editor) 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 doesn't start with a `#` (we say that the lines with `#` are commented and therefore the computer doesn't "see" them). 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. However, `ifort` is paid, so 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 the one we will use here. You can install gfortran with `sudo apt install gfortran` (or the equivalent installation command for your system). So, for gfortran, you should make the above lines of code look like the following:

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

When doing that we say that we "uncommented" the gfortran line, since every line that starts with a `#` is commented and not read by the system. You can save and exit the file.

Now all you have to do is run the following command:

```
make -f Makefile_small_strato which will compile your Fortran code into an
```

executable file (`.exe`) using the options we just set. You should see a lot of lines appearing on screen starting with `gfortran`, maybe some warnings, and if no error messages appear the compilation was successful.

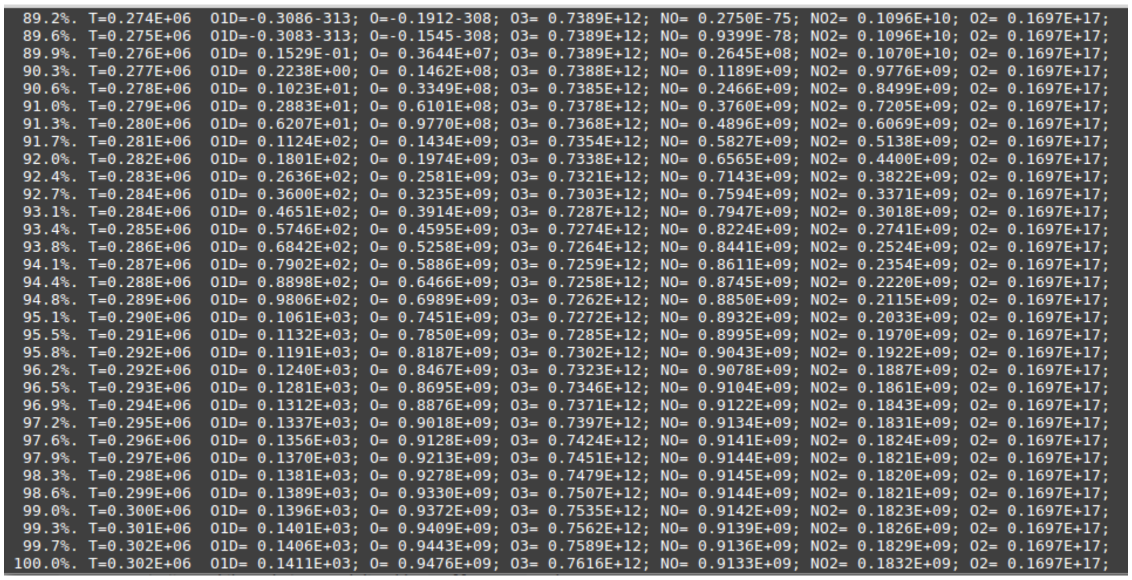
Now you'll see many more new files, including one called `small_strato.exe`, which is your executable file (run `ls` again list everything and see that). This is the executable that will actually calculate the concentrations using the model.

To test if it works, run the following command:

```
./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 successful and everything worked well! You just calculated the concentrations 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 the test case

Now let's understand why our run of `small_strato.exe` was successful and what happened. First, by running `kpp small_strato`, what we did was to tell KPP to open a file called `small_strato.kpp`, in the current directory and do what that file tells it to do. In the first line of the file there is the command

```
#MODEL      small_strato
```

which tells KPP to look for a file called `small_strato.def`. Since the file is in KPP's models directory (at `$HOME_KPP/models`), KPP had no problems finding it. This file has the initial concentrations you want to use in the model, the time step, etc. It also links two other files (`small_strato.spc` and `small_strato.eqn`), which tell KPP with chemical species and chemical equations to use (effectively defining the mechanism).

After receiving all that information, KPP finally creates a Fortran 90 code (because it says so in the `small_strato.kpp` we created) with our small stratospheric model containing our pre-defined initial conditions, time step, chemical reactions and so on.

The code, however, has to be compiled before run, so that is why we issued the command `make`, which compiles the code according to the file `Makefile_small_strato` (which is where we specified the Fortran compiler). This step creates an executable file, which has the extension `.exe` and is ready to be run. By running the `.exe` file we ran a program that got our initial concentrations of the species we defined and, based on the chemical reactions, calculated, step by step, their concentrations in each time step.

At each step, the model is not only printing the concentrations on screen, but it is also writing them into a file called `small_strato.dat`, which is a column-separated text file. This file can be used to see, plot, make calculations with the data and so on. However, you should be careful because the order of the concentrations that appear on screen isn't the same order KPP uses for the `.dat` file. You can learn about the ordering at page 7 of the KPP manual, but a good rule of thumb is to check the file with a `.map` extension (in this case, `small_strato.map`) and take a look at the `species` section. The file output order is the ordering of the variable species followed by the species on the fixed species.

In the case of `small_strato` the order printed on the file (you can check it on `small_strato.map`) is

```
time, O1D, O, O3, NO, NO2, M, O2
```

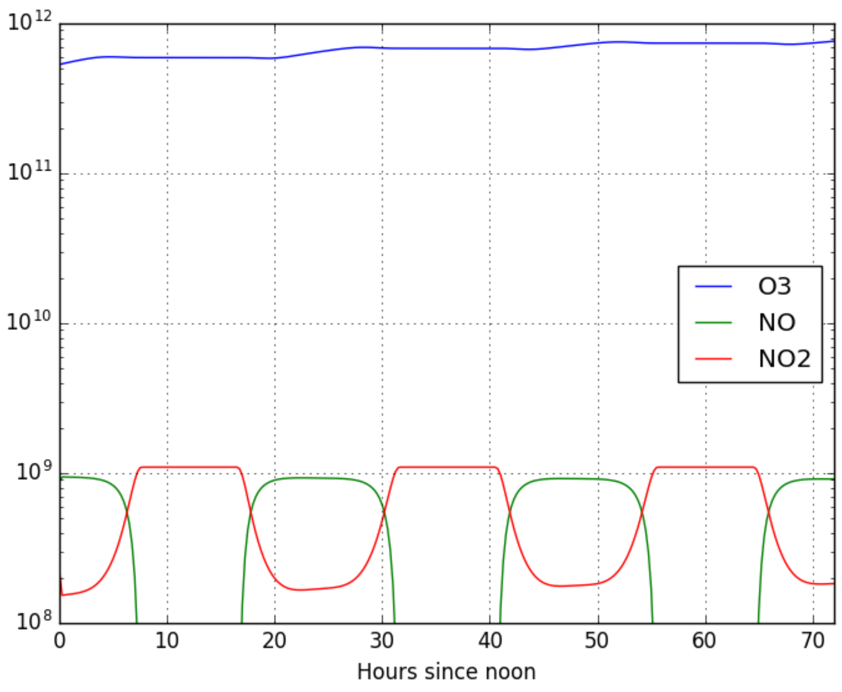
The time is always going to be the first column, and it is always going to be in hours since the start of the simulation. Since the solar forcing matters here, we need to keep track of the time of day that the simulation started. In this case it was at noon, because that's the way the `.def` file is set (we will talk about this in more detail in the sections to come).

We can read that data in many ways. I present below a quick python script to plot the concentrations as a function of the hour of the day

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3 concs = pd.read_csv('small_strato.dat', index_col=0, delim_whitespace=True,␣
  ↪header=None).apply(pd.to_numeric, errors='coerce')
4 concs.columns = ['O1D', 'O', 'O3', 'NO', 'NO2', 'M', 'O2']
5 concs.index.name = 'Hours since noon'
6 concs.plot(ylim=[1.e8, None], logy=True, y=['O3', 'NO', 'NO2'], grid=True)
7 plt.savefig('test1_time.png')
```

Note: KPP has a small issue with formatting and sometimes prints a number that can't be read because some strings are missing. For example, printing `3.4562-313`. This can't be normally read and it's supposed to be `3.4562E-313` and this (apparently) only happens when the number is close to machine-precision (which we would interpret as zero). The program above takes this issue into consideration (in line 3) when reading the file, but you should pay attention to that when trying to read with by other means.

If you have ever seen python before, this code should be pretty intuitive. If you haven't you can still use it easily (depending on how you got python, you might have to install python's `pandas` package). This code generates the following plot of the concentrations:



We can see that the NOx concentrations follow the solar cycle, which is indicative that the model is indeed working properly. However we see that the O3 concentrations still haven't stabilized. This tells us that we need to run the model for longer. Let us take this chance to modify the `small_strato` example a bit, try and make the O3 concentrations stabilize and learn how to alter/create models.

CHAPTER FIVE

MODIFYING AND IMPROVING THE EXAMPLE

Increasing the length of the simulation

There are two ways to make modifications on the model. The first, which works for simple changes, is to modify the Fortran code itself. The second is to change the KPP model itself (the `.def` files etc.) before it gets compiled. This latter method is more general, so this is the one we will focus on this guide. Since all we want for now is to increase the total time, we will base ourselves in the original `small_strato` model and only modify this parameter.

First, create another directory (anywhere you want) called `test2` (with `mkdir test2`) and enter it (with `cd test2`). Now create a file called `my_strato.kpp` (with `notepad++ my_strato.kpp` or `gedit my_strato.kpp` or whichever text editor you ended up using) and paste the following lines in the file:

```
#MODEL      my_strato
#LANGUAGE   Fortran90
#INTEGRATOR rosenbrock
#DRIVER     general
```

At this point if you run `kpp my_strato.kpp` you should get an error saying `"Fatal error : my_strato.def:  Can't read file"`. Which appears because we instructed KPP to search for the file `my_strato.def`, which doesn't exist anywhere. So we first must create the `my_strato.def` file, which ultimately defines the `my_strato` model.

Let us define our model based on `small_strato`, since for now all we want to do is to modify the time length of the simulation. In order to preserve the original `small_strato.def` we'll copy it and call it `my_strato.def`, this way we can do any modification on `my_strato` and the original `small_strato` will be safe. You can copy the file from the `models` directory into our working directory (`test2`) by issuing the following command:

```
cp $KPP_HOME/models/small_strato.def my_strato.def
```

Note: When we run KPP from any directory (say, `test2`), KPP will first look for the files in the current directory and then in its home directory. So can either put our model files in our KPP "models" directory or in our current directory. In this guide we'll always prefer to keep the model we create/modify in the current directory.

Now you should open the file we just created (for example with `notepad++`) and find the lines that look like:

```
#INLINE F90_INIT
        TSTART = (12*3600)
        TEND = TSTART + (3*24*3600)
        DT = 0.25*3600
        TEMP = 270
#ENDINLINE
```

These are the lines that define the start, end, time step and global temperature of the model. For now, we'll change only the end time. Notice that they are given in seconds. For example, `3*24*3600` is the amount of seconds in 3 days, meaning that at the moment the simulation is set to run for 3 days, which we saw is not enough. Let us then replace `3` with `30`, meaning we will run it for 30 days, to be sure that equilibrium is reached. Now that line should read `TEND = TSTART + (30*24*3600)`. Please also note that `TSTART` is set to `12*3600`, or 12 hours, which means that the starting time of the simulation is at noon.

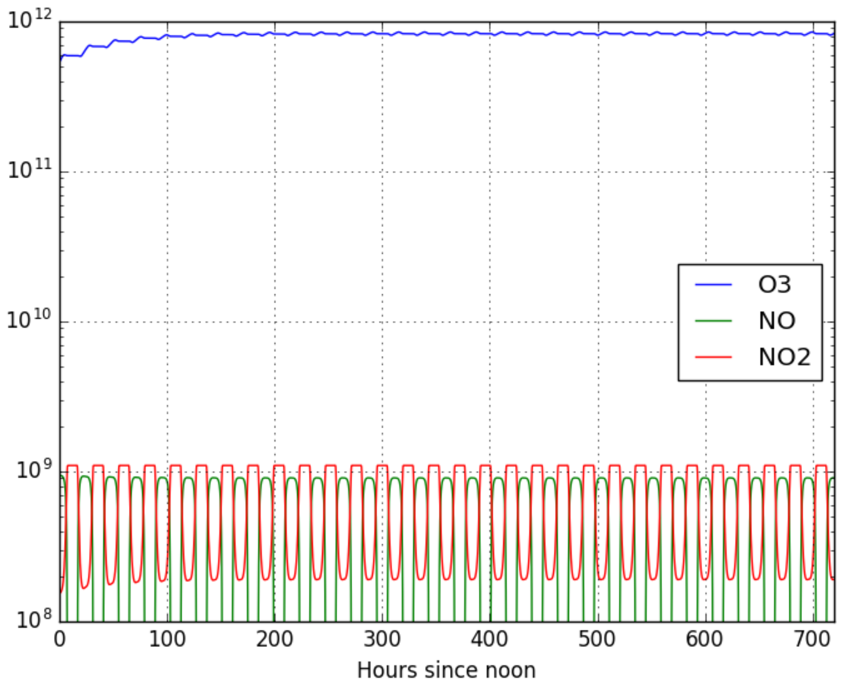
After this small change we are ready to test run the model. Run `kpp my_strato.kpp`, which should end with a `succesfully created the model` message. Now, just like with the previous example, open again the file `Makefile_my_strato` and uncomment the line that says `COMPILER = GFORTRAN`, so we can use gfortran instead of Intel. After this is done compile the model with:

```
make -f Makefile_my_strato
```

Again, if everything goes well, the `my_strato.exe` file should be created. You can run `ls -tr` (which lists every file on your directory ordered by creation time) and see that the last file listed should be the `.exe` file.

We can now run the model with `./my_strato.exe`, which should now take 10 times longer to complete, since we are running it for 10 times as long as before. We can use the Python code given in the last section to read the results. We just need to adjust the name inside the code from `small_strato` to `my_strato`.

The plot of the results is



Now we can see that the solution reaches equilibrium after roughly 200 hours. This, however, was only a minor change in the model. Let us now learn how to change some other parameters and examine how the model reacts.

Change in the initial conditions

We try this next change in the same model as before (`my_strato`). Let's open `my_strato.def` and consider an atmosphere with more NO (simulating a more polluted condition). Where it says `NO = 8.725E+08;`, we make it read `NO = 9.00E+09`, which is roughly a 10 times increase in the NO concentration. Let us also change the O3 initial condition and make the O3 line read `O3 = 5.00E+10;`, simulating an atmosphere that has a lower initial O3 concentration.

Let us also change the line that reads:

```
#LOOKATALL          {File Output}
```

We will make it read

```
#LOOKAT O3; NO; NO2; {File Output}
```

This latter change makes the program write only time, O3, NO and NO2 into the output file. This simplifies the reading process and saves space, but by doing that we are assuming that we're not interested in the other species.

Note: This will change the order of the output in the file. But again, checking the `.map` will give you the correct order. This time you'll have to consider only the species you specified to be on the output, so, e.g., in this case the order in the `.map` file is: 1 = O1D, 2 = O, 3 = O3, 4 = NO, 5 = NO2. But since we are writing only O3, NO and NO2, our output file will have the order (time,) O3, NO, NO2 (which are numbers 3, 4, 5, respectively). This will have to be done every time the `#LOOKAT` parameter is used.

We again go through the same steps: run it with `kpp my_strato.kpp`, change the compiler to gfortran, compile if with `make -f Makefile_my_strato` and run it with `./my_strato.exe`. This time, since the output file changed, we have to change the code to read it correctly:

```
import pandas as pd
from matplotlib import pyplot as plt
concs = pd.read_csv('my_strato.dat', index_col=0, delim_whitespace=True, header=None,␣
↪dtype=None).apply(pd.to_numeric, errors='coerce')
concs.columns = ['O3', 'NO', 'NO2']
concs.index.name = 'Hours since noon'
concs.plot(ylim=[1.e8, None], logy=True, y=['O3', 'NO', 'NO2'], grid=True)
plt.savefig('test21_time.png')
```

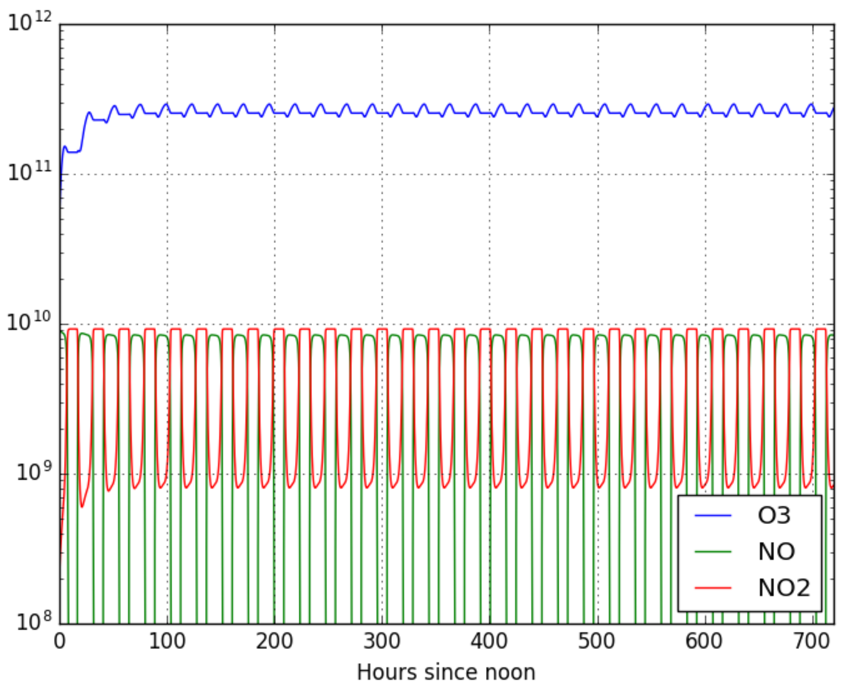
This code produces the following plot:

From which we can see that the concentration of ozone stabilized more quickly in this case. As you can see, we can play around with the initial conditions as much as we want and analyse that result of model. In fact, we encourage you to do so. However, let us focus this guide on the next step and modify some more fundamental aspects of the model: the reaction rates.

Modifying the reactions

Now we will alter the reaction rates of some reactions in the model. Keep in mind that these alterations are not meant to be realistic. They are simply done here for the sake of learning how the model works.

Begin again by creating a `test3` directory anywhere and going into it (`mkdir test3 && cd test3`). In this directory, create a file called `strato3.kpp` with the following contents:



```
#MODEL      strato3
#LANGUAGE   Fortran90
#INTEGRATOR rosenbrock
#DRIVER     general
```

This file tells KPP to look for the `strato3.def` file. So let us create this file by again copying the `small_strato.def` file to our current working directory. You can do that with:

```
cp $KPP_HOME/models/small_strato.def strato3.def
```

Open the file (`notepad++ strato3.def`) and find the first two lines which originally read

```
#include small_strato.spc
#include small_strato.eqn
```

Which still tells KPP to look for the original `small_strato` model files when defining the species (`.spc`) and chemical equations (`.eqn`). You should modify these lines to the following:

```
#include strato3.spc
#include strato3.eqn
```

Also, you should do same modification we did in the last example. That is to change the length of the run from 3 to 30 days by modifying the line that reads `TEND = TSTART + (3*24*3600)` to make it read `TEND = TSTART + (30*24*3600)`, and to change the line that reads `#LOOKATALL` to `#LOOKAT O3; NO; NO2;`.

If you try to run KPP now you'll again get an error because those files still don't exist. Let's create them by copying the original `small_strato` files, which can be done with the following commands:

```
cp $KPP_HOME/models/small_strato.spc strato3.spc
cp $KPP_HOME/models/small_strato.eqn strato3.eqn
```

Now if you try running KPP it should work. But this is still not what we want; this is just the `small_strato` mechanism with another name, so let us move to the actual changes.

If you check the `strato3.spc` file you'll see that it only the definitions of the species used, which wouldn't make much sense to change for now since we'll be using the same species, so we will leave it how it is. Now we focus on the `strato3.eqn` file. If you open it you'll find the following lines:

```
#EQUATIONS { Small Stratospheric Mechanism }

<R1>  O2   + hv = 2O            : (2.643E-10) * SUN*SUN*SUN;
<R2>  O    + O2 = O3            : (8.018E-17);
<R3>  O3   + hv = O   + O2      : (6.120E-04) * SUN;
<R4>  O    + O3 = 2O2           : (1.576E-15);
<R5>  O3   + hv = O1D + O2      : (1.070E-03) * SUN*SUN;
<R6>  O1D  + M  = O   + M       : (7.110E-11);
<R7>  O1D  + O3 = 2O2           : (1.200E-10);
<R8>  NO   + O3 = NO2 + O2      : (6.062E-15);
<R9>  NO2  + O  = NO  + O2      : (1.069E-11);
<R10> NO2  + hv = NO  + O       : (1.289E-02) * SUN;
```

Just for the sake of learning, let us change the photolysis rate (last reaction) to make it a lot slower. We will make the last line read:

```
<R10> NO2  + hv = NO  + O       : (1.289E-06) * SUN;
```

Note: This is 4 orders of magnitude slower than it previously was and it may not be realistic! We only make this change for the sake of illustration, so that the output change is easier to see.

You can actually change not only the reaction rate for the equation, but also modify the equations itself here and add (or remove) equations. For now we will leave the equations the way they are.

Now we go through the same steps of running `kpp strato3.kpp`, changing the compiler to gfortran and running `make -f Makefile_strato3`. If everything goes well, we'll see the `strato3.exe` created. After running `./strato3.exe` sure enough `strato3.dat` is created, which we can plot with the same python code from the last example (only changing the name of the file of course):

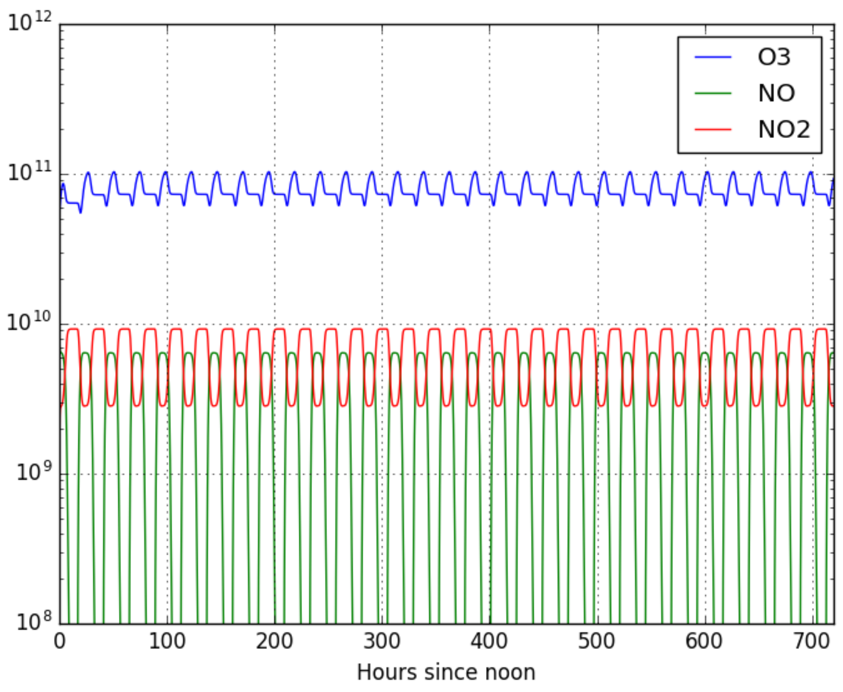
Note: This process of running KPP, then changing the Makefile, then compiling, etc., is pretty cumbersome and straightforward. So we included a file called `updatenrun.sh` in the directory `test3` that can be found in the github repo. This is a bash script that does these steps automatically. To run it, you enter `sh updatenrun.sh modelname`. In this case, for example it shoudl be used with `sh updatenrun.sh strato3`.

We can see that once again the final result changed. This time, since NO2 is photolizing a lot slower, we see less NO in comparison with the previous plot. We encourage you to try different reaction rates and initial conditions and see what is the result in the model. With the `updatenrun.sh` script (check the note above) it should be easy!

Now that we have modified the `small_strato` example in several ways, let take it a step further and create a new model from scratch.

Creating a model from scratch

Now we do one more step and create a completely new model with our own set of reactions. Basically for our new model to be complete we should give it the initial conditions, numerical constraints, species and reactions list. Let us start with the KPP file and move on from there.



We will try to simulate a very small tropospheric model, which we will call `ttropo` (meaning tiny tropospheric; let's write it like that just because it's easier). First let's create a new directory for our test with `mkdir ttropo` and move to that new directory with `cd ttropo`. Now we create the main KPP file with `notepad++ ttropo.kpp` and put the following lines in it:

```
#MODEL      ttropo
#LANGUAGE   Fortran90
#INTEGRATOR rosenbrock
#DRIVER     general
```

Which means tells KPP to look for the `ttropo.def` file. If you run KPP now it will finish with an error because it won't find it. But we will create that later. Let us first define our mechanism, i.e., our chemical reactions.

We create the `ttropo.eqn` file (e.g. with `notepad++ ttropo.eqn`). Now we will put our reactions in that file, following the syntax that we saw in the previous example. We choose a simplified set of tropospheric reactions that can be written as:

```
#EQUATIONS { Tiny Tropospheric Mechanism }
<R2>  O     + O2   = O3           : (8.018E-17);
<R1>  NO2   + hv   = NO   + O     : (1.289E-02) * SUN;
<R3>  NO    + O3   = NO2  + O2    : (6.062E-15);
<R41> O3    + hv   = O    + O2    : (5.500E-04) * SUN;
<R42> O3    + hv   = O1D  + O2    : (6.000E-05) * SUN*SUN;
<R5>  O1D   + M    = O    + M     : (7.110E-11);
<R6>  O1D  + H2O   = 2OH          : (2.2E-10);
<R7>  CO    + OH   = CO2  + HO2   : (2.2E-13);
<R9>  HO2   + NO   = OH   + NO2   : (8.3E-12);
<R10> OH   + NO2   = HNO3         : (1.1E-11);
<R11> HO2 + HO2    = H2O2         : (5.6E-12);
<R12> O3   + HO2   = OH + 2O2     : (2.0E-15);
<R13> H2O2 + hv    = 2OH          : (1.366E-5)  * SUN;
```

```
<R14> H2O2        = H2O2aq        : (3.3000e-03);
<R15> HNO3        = HNO3aq        : (2.4000e-03);
```

So copy and paste those lines into `ttropo.eqn`, save and exit.

Note: Again, some of these reaction constants might not be exactly accurate for tropospheric conditions, so please double-check if you plan on using them for professional means, since the objective here is to only present this as an example.

Now we create the species file, which has to have all the species we used in the reactions above properly defined. We can define a species as being variable (when its concentration can vary according to the kinetics) or fixed (when its concentration is a constant). In this case, we define only `M`, `H2O` and `O2` as fixed quantities and the other ones as variables:

```
#include atoms

#DEFVAR
O    = O;              { Oxygen atomic ground state }
O1D  = O;              { Oxygen atomic excited state }
O3   = O + O + O;      { Ozone }
NO2  = N + O + O;      { Nitrogen dioxide }
NO   = N + O;          { Nitric oxide }
HNO3  = H + N + O+O+O;
H2O2  = H+H + O+O;
CO2   = C + O + O;
CO    = C + O;
OH    = O + H;
HO2   = H + O + O;
H2O2aq = IGNORE;
HNO3aq = IGNORE;

#DEFFIX
M    = O + O + N + N;{ Atmospheric generic molecule }
O2   = O + O;          { Molecular oxygen }
H2O  = H + H + O;      { Water }
```

Again, copy and paste those lines into `ttropo.spc`, save, exit, and let's proceed to the `.def` file. Create `ttropo.def` with `notepad++ ttropo.def`. In that file you will write the following lines:

```
#include ttropo.spc
#include ttropo.eqn

#JACOBIAN SPARSE_LU_ROW      {Use Sparse DATA STRUCTURES}
#DRIVER general
#DOUBLE ON
#STOICMAT ON

#LOOKATALL;                 {File Output}}
#MONITOR O3;N;O;NO;O1D;NO2; {Screen Output}

#CHECK O; N;                   {Check Mass Balance}

#INITVALUES                    {Initial Values}
CFACTOR = 1.    ;              {Conversion Factor}
```

```
O3  = 7.65E+11 ;
NO  = 2.55E+10 ;
NO2 = 0.0E+09 ;
O2  = 1.697E+19 ;
M   = 2.550E+19 ;
H2O = 3.0E+17;
CO  = 2.55E13;

OH     = 0.;
HO2    = 0.;
H2O2   = 0.;
H2O2aq = 0.;
HNO3   = 0.;
HNO3aq = 0.;
O1D    = 0. ;
O      = 0. ;

#INLINE F90_INIT
       TSTART = (12*3600)
       TEND = TSTART + (15*24*3600)
       DT = 0.2*3600
       TEMP = 270
#ENDINLINE
```

You can see that with this set of definitions we chose to run the model for 15 days, with a time step of 0.2 hours and that many of the initial concentrations are set to zero. Note also that we are again starting the simulation at noon.

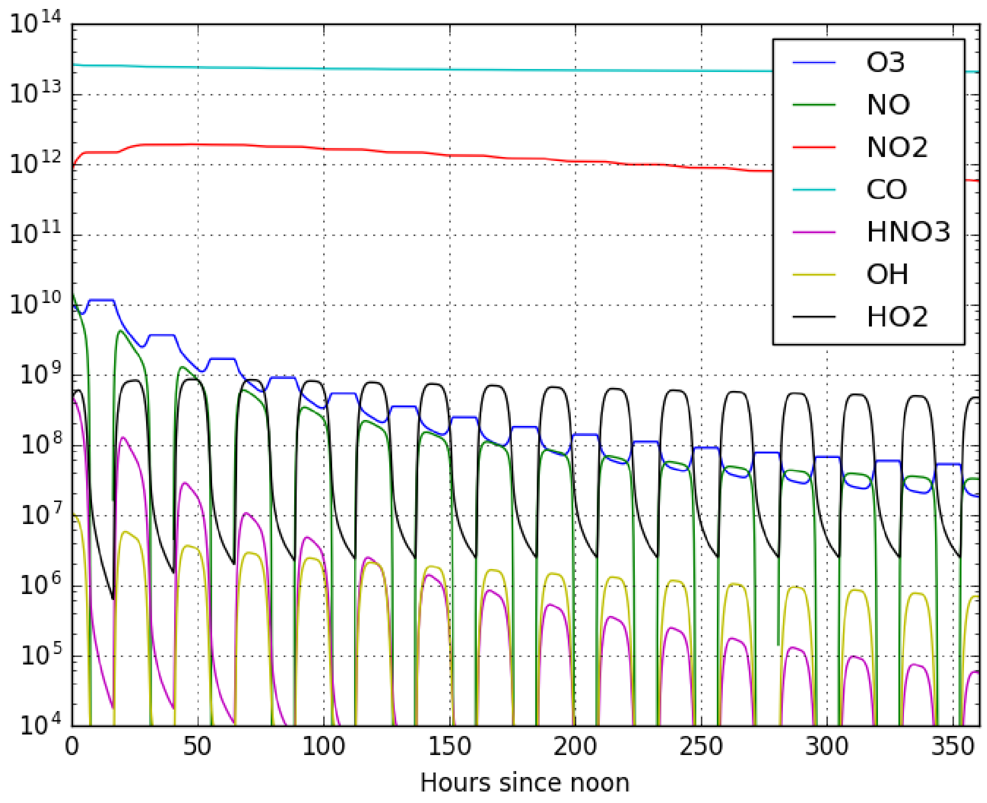
With these files we have the complete `ttropo` model and are ready to run it. We can use the `updatenrun.sh` script as `sh updatenrun.sh ttropo` (you'll have to copy it to the current directory with `cp` first). It should run successfully now. Note that we again have to check out `ttropo.map` to find out the order of the species in the output file. We can use the following Python script to plot the results (the correct output order is already included in it):

```
import pandas as pd
from matplotlib import pyplot as plt

concs = pd.read_csv('ttropo.dat', index_col=0, delim_whitespace=True, header=None,␣
↪dtype=None).apply(pd.to_numeric, errors='coerce')
concs.columns = ['CO2', 'H2O2aq', 'HNO3aq', 'HNO3', 'H2O2', 'CO', 'O1D', 'O', 'OH',
↪'HO2', 'O3', 'NO', 'NO2', 'M', 'O2', 'H2O']
concs.index.name = 'Hours since noon'
concs.plot(ylim=[1.e4, None], logy=True, y=['O3', 'NO', 'NO2', 'CO', 'HNO3', 'OH',
↪'HO2'], grid=True)

plt.savefig('test4_time.png', bbox_inches='tight')
plt.show()
```

The output of this model can be seen in this figure:

As you can see, in 15 days we ran the model for, it hasn't reached equilibrium yet. This system of equations will take longer to reach equilibrium state than previous models. You can, once again, investigate the effects of different initial concentrations in the final result, change the reactions and reaction rates, or even fix some quantities in the `.spc` file.

The creation of any new model from scratch follows the same paths that we just described here. So for other models, following these steps should produce the correct result. For model detailed information not included here, we refer the user to the official manual for KPP.


10^{14}
10^{13}
10^{12}
10^{11}
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{4}
0
50
100
150
200
250
300
350
Hours since noon
O3
NO
NO2
CO
HNO3
OH
HO2

CHAPTER

SIX

POSSIBLE BUG FIXES

Note: This section is meant to become a list of common bugs that might arise along with a way to solve them.

Can't fine bashrc file

If you could not find your `.bashrc` file, it might mean that you either don't have one, or that it is located somewhere else. If you're using C shell, then you should be actually looking for `.cshrc`. If that is indeed the case, replace `bashrc` by `cshrc` everywhere it appears. Furthermore, environment definitions are made slightly different in C shells. Instead of typing `export KPP_HOME=$HOME/kpp` for example, you would have to type `setenv KPP_HOME $HOME/kpp`. For other (less common) shells, we advice you to google these definitions. They should be easy to find. When in doubt of which shell you're using, type `echo $SHELL` and check the output.

If you still can't find your `.bashrc` or `.cshrc`, chances are you're using an emulator, and not running natively. If this is the case, google for the location of the `.bashrc`-equivalent in your shell emulator (be it, Cygwin, Mingw, cmder or others).