ezkpp Documentation

Release 0.1

Tomas Chor

Oct 25, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

This guide is aimed at providing additional information not covered by the official KPP manual. We focus on the latest version of the pogram, release 2.2.3, which can be freely downloaded at its official webpage. We recommend any person reading this manual to keep a copy of the original mnaual (which can be downloaded here) since this guide is not meant to replace the original manual, but to suplement it.

CHAPTER

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 outter 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.

CHAPTER

THREE

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. You can also enter *ls*, which will "list" everything you have on that directory. 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/user/Downloads* depending on where you are on your terminal (the first is a relative and the second is an absolute or full path).

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/myname/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.

Alternatively, you can open a terminal and run

```
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
```

which will automatically download the software, unpack it and move to the correct directory (which was created when unpacking).

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.

Be sure that FLEX (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 or by manually installing downloading and installing the file if you're emulating (with Cygwin, for example).

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 going to have to install it.

Telling your system where KPP is

Now that Flex is installed, we need to make sure that the system knows where KPP is in your system. We do that by altering a file called `.bashrc`, which sets some configurations for your terminal (if you're using a Bash terminal). This is a text file and we simply need to add some lines. We'll do that step by step.

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 to tell your terminal where KPP is.

Now you need to open and edit `.bashrc` which can be done with many programs. The best option would be to try `gedit`, `geany`, `pluma` or `abiword`, which all have graphical user interfaces (GUI). You can try all of these in the command `gedit ~/.bashrc`, or `geany ~/.bashrc` and so forth with the others. If any of those work, great! If not, you're probably going to have to use Nano by running the command `nano ~/.bashrc`, which runs on the terminal itself.

With Nano, you're going to see something like Fig. **'ref::something'_** after you open it. If you're using something else you'll probably see something more friendly, but with the same lines of code. You can ignore all those codes and jump to the last line of the file. You're going to create another line and paste

```
export KPP_HOME=$HOME/kpp
export PATH=$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/user/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 do it by pressing control X, choosing the "yes" option (by only pressing y) 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.

Specifying how to compile

If this worked, we are ready for the next step, which is telling your system how to compile KPP. First, type `locate libfl.a` and save the output. If that is no output, use `locate libfl.sh` and save the output of that. In my case the output was `/usr/lib/x86_64-linux-gnu/libfl.a`. If neither of those commands gave you an output, you might need to install the Flex-dev package with `sudo apt install flex-devel.x86_64`.

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. 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 `#`. Look for the 5 items to complete in this file. The first one is `CC`, which sets the 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 defines the options of the compiler and extra directory to include in the compilation. We will not worry about those, which unless maybe when debugging. 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, 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 file 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 file any file to work with, which is fine because we didn't give it any yet. If this worked, 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.

CHAPTER

FOUR

RUNNING KPP

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 feature when running KPP.

In order to run, our example 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 a line 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
```

These are going to be used to compile the code generated by KPP. Let's focus for now on the `Makefile_small_strato`. This tells which Fortran compiler to use 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
```

This tells the computer to use the intel Fortran compiler, `ifort`. If you are using `ifort`, you should leave it as it is. If you are using another compiler, put the `#` in front of the `INTEL` 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 what we will use here. So, assuming gfortran, make it 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. Now you'll see many more new files, including one called `small_strato.exe`, which is your executable file. To test, run `./small_strato.exe`, which will run the executable. You should see some output on the screen with concentrations, telling you that your run was successfull and that everything worked well!

Understanding and improving on the test case

CHAPTER

FIVE

UNDERSTANDING THE OUTPUT

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search