pbl_met examples

An Example: a Simple Dew Point Calculator

Mauri Favaron

Introduction

In this example we'll see how to build a simple dew point calculator.

It will not possess a fancy graphic interface, or a dialog input windows: it will just use the command line to pass the program the value of temperature (in °C) and relative humidity (in %). Then, just a push to the `Return` key, and you're done.

Calculating the dew point is a breeze with *pbl_met*: If you look for in the manual, you will find a subroutine, or function, or class member already written and doing what you like. This is one of those lucky, although common, cases.

What is dew point, first of all?

To say it simple, the dew point is the temperature at which the water contained in moist air at a given ambient temperature and relative humidity condensates to liquid phase.

The dew point makes a lot of sense at temperatures higher than 0 °C. But condensation may also occur below 0 °C, and in this specific cases meteorologists speak of "frost-point". For us, this subtle distinction does not apply.

Of course, for condensation to occur temperature must *decrease* from the current one. This means, dew point will always be smaller than air temperature. The difference between dew point and air temperature gets larger as humidity decreases: this again is quite intuitive a fact, as a high relative humidity means a large amount of water still available in gas phase, at a partial pressure close to saturation: get air just a small temperature decrease, and you're done. Saturation pressure increases a bit, until it reaches the water vapor partial pressure, and condensation occurs.

Why is dew point important?

The dew point has many practical applications, some of which very important.

It is, for example, the decrease of air temperature towards and below the dew point to form those beautiful *orographic clouds* when airflow bypasses a cliff on a fair weather day. If you look attentively, you will see the clouds on top of hills continuously form and dissolve, in a dynamical equilibrium. If you love photography, this phenomenon is a wonderful subject for a time-lapse video.

Dew point has to do with many engineering processes. Like, for example, the thick columns of smoke you sometimes see arising from an industrial plant - the stacks, but also other unit operations like, most notably, condensers. The "smoke" is actually water vapour (for the most part), and the fact you sometimes see it and some others do not depends on how close the release temperature was to the dew point. In general, cold humid days are most ideal for this kind of observation. The important thing in this case is that the plant is releasing into the atmosphere a really huge amount of water vapour, and if it condenses close to the release point artificial precipitations may occur (which may cause some damage), and most surely you will change the relative humidity balance.

These are just two cases, but you may find quite many on yourself, from biology to engineering.

So, definitely, dew point is not something "abstract": it impact the lives of us humans and other creatures as well. And, of course, the whole local and global ecosystem.

A dew point calculator, then. But, which type?

If you surf the web, in less than a minute you may find many dew point calculators. If you select one of them, quite often you will find a lovely page allowing you to fill a temperature and a relative humidity fields, click on a button, and the answer will pop up almost immediately.

This works good, if you are connected to the Internet. But, if you are at a remote site, chances are good that no Internet connection is available.

Another important thing to consider, when using a dew point calculator from the web, is that dew point calculations are done approximately, that some approximations are better than others, and that in some cases you have no cue from the web page on which approximation is used. So, the number you get as a result may not be accurate enough for your application, and you do not know.

With the *pbl_met*, on the contrary, you may easily trace the method used. If you desire, you may even check its implementation, and decide whether you like it or not (in the latter case, please tell us by filing a push request on *pbl_met* GitHub space: we love taking proposal into account and, if sensible - that is, the math is sound, the implementation is good, and the method passes tests - to incorporate in the library code).

Last, a *pbl_met* based implementation is lightweight, and fast. This may not be a problem, with something as simple as computing a dew point, but it comes to be of importance as problem scale grows (*pbl_met* routines are designed to deal with enormous-scale problems as well).

Some details more on what we want

So: our calculator will be a command-line application.

It's time we say something more - that is, we lay out some basic requirements. I'm so sorry we aren't doing this together, because this phase benefits greatly from peer-with-peer brainstorming. But, I'm still in the position of showing you something of this process. Sorry, also, for it not being quite a "linear" thing, as many engineers would appreciate. But on the other hand, most creative endeavours, even the simplest, tend to occur in web-thinking mode...

Then, let's weave the beginning of our (hopefully solid) web of requirements, by posing a question, and trying to answer it: How many dew point calculations will have to be made upon a single activation of our calculator?

We have in principle two big possibilities: just one dew point, or many.

Solving "many" (similar) problems with a single program is after all what makes computers attracting: they may perform huge bunches of work without even feel bored (or just *feel* something), and will not question the sanity of our orders.

On the other hand: would treating many cases increase the program complexity? Likely so? My experience-laden answer is a resounding "Yes!": treating more than one case would oblige us to write codesorting through them, and this may be an order of magnitude more code lines than the ones (likely just one) necessary to actually compute the dew point.

So, is the "one case only" solution better?

Maybe, not. Sure, it looks simple. But, also, it is terribly inefficient: to compute more than one value the whole executable code must be re-loaded in memory, launched and executed.

So, neither solution may be considered ideal, and then we need to make a choice, not just to take a simple decision.

Choices always involve some feeling of loss - that's natural and inevitable - but, as their trivial relatives, decisions, may be addressed rationally.

By posing and answering other questions.

For example: which the purpose of this example is? Now the answer is quite simple and univocal: the purpose is *not* to devise the most efficient dew point calculator possible, but to illustrate a possibility, and show how to change it from an idea to reality.

Execution cost, then, is secondary.

This given, we may opt for the single shot version.

Now: how interaction with users will occur? I can imagine two possibilities, maybe two-and-half:

- Execution occurs purely from the command line, with temperature and relative humidity passed as command line parameters.
- Or, execution starts from the command line, but actual access to data occurs through a file in some appropriate form (CSV?) containing temperature and relative humidity (this is a case very similar to the previous point's).
- Or even, execution may flow interactively: the program starts, asks users to enter, say temperature, reads it, then asks for relative humidity, gets it, and finally computes and prints the result.

Each approach offers their own advantages and disadvantages. The first looks clean and economical, but demands user learn what "command line parameters" are - something unfamiliar to people accustomed to graphical user interfaces. The second offers easy extendability to the multiple-problem case, but at the price of demanding a data file in addition to the program. The third looks more graphical user interface like, but is clumsy and, let me say, inelegant (personal opinion - emulating a GUI on a character terminal is not really fashionable).

Once again, we must choose.

And once again, considering simplicity is paramount, I choose the first, execution from pure command line.

Ours is a very simple program, and we have figured out anything important we have to do; in a more realistic situation we could face not just two, but dozens of choices, and be careful one does never violate or interfere with some others.

Nonetheless, we may step to the actual implementation.

Well, almost...

How to compute dew point with the *pbl_met* library?

This question is the realm of systematic action, and not of more or less informed choices: we have to find which actual routine to use, and look how it is made.

You might use at least two possible approaches, one more suitable to straight-to-bread shopping mall types, and the other to the affectionate browsers who look and check the form shape and price of any goods exposed.

The lazy straight-to-result people will scan through the PDF copy of the manual for something like "dew" and "point", find the routine name, and go to its description.

The explorers, on the other side, may want to check the manual beginning, discover that anything thermodynamical belongs to `pbl_thermo` sub-module, look there, discover many interesting and potentially useful functions, and find the desired routine name.

Neither of the two approaches is "best". The former may be quicker, but is also exposed to misunderstanding type errors (possibly macroscopic). The latter is surely slower, but allows to build a wide view of the library possibilities, which opens the way to creative solutions in other cases.

In both cases, the end result is, to paraphrase the movie *The Blues Brothers*, a name, and a last known address.

In this specific case, I made all the dirty work for you and tell the routine to use is

```
    FUNCTION DewPointTemperature(Td, Ur) RESULT(Dp)

        ! Routine arguments
        REAL, INTENT(IN)    :: Td       ! Dry bulb (that is "ordinary")
temperature (K)
        REAL, INTENT(IN)    :: Ur       ! Relative humidity (%)
        REAL                :: Dp       ! Dew point (K)

        ... other things ...
    END FUNCTION DewPointTemperature
```

A first implementation attempt

All those choices and things known and acknowledged, we may try with a first version:

```
program DewPoint

    use pbl_met

    implicit none

    ! Locals
    integer             :: iRetCode
    character(len=16)   :: sBuffer
    real                :: temp     ! Temperature, in °C
    real                :: relh     ! Relative humidity, in %

    ! Get temperature and relative humidity from the command line (or print
a helpful
    ! message if something looks wrong
    if(command_argument_count() /= 2) stop
    call get_command_argument(1, sBuffer)
    read(sBuffer, *, iostat=iRetCode) temp
    if(iRetCode /= 0) stop
    call get_command_argument(2, sBuffer)
    read(sBuffer, *, iostat=iRetCode) relh
    if(iRetCode /= 0) then
        print *, "Error: invalid relative humidity!"
        stop
    end if

    ! Perform the calculation using pbl_met, and print it on the fly
    print *, "Dew point temperature = ", DewPointTemperature(temp + 273.15,
relh) - 273.15, " °C"

end program DewPoint
```

Notice the reiterated conversions from Celsius degrees to Kelvin, and back: this, because the *pbl_met* dew point routine "thinks" in Kelvin, while the data we decided to support are expressed in Celsius degrees (as most often happens with measurements).

You may see, even in a so simple program most code is devoted to get input data from the command line, and deal with exceptional cases.

The latter task is always essential, and no professional (nor well written amateur) code can do without. What you might find questionable is, the actual way to "deal" with anomalies is just stop the program as soon as the first is detected. The net result is to avoid invoking the function passing blatantly nonsensical data to it: in a sense, it protects the *pbl_met* against user mismanagement. But, it does nothing to let the user really understand whether they have used the function the wrong way, why, and what to do to the next time to avoid problems.

Making users know what to do in case of troubles

This is quite personal, but in my (strong) opinion a well-crafted program always gives indications if something go wrong.

This objective may be accomplished in many ways. The one I prefer is to print usage instructions whenever the wrong number of command line arguments is given and, would circumstances suggest (this is one case), when some argument value is found invalid.

As usage instructions may need being printed from different code locations, it is better to embed them into a subroutine or function, and call it whenever necessary.

The following code block illustrates the second, final, version of our calculator.

```
! Simple example program showing how to compute the dew-point, given
"regular"
! (dry bulb) temperature and relative humidity - just the stuff you may
hope
! to get from a thermo-hygrometer.

program DewPoint

    use pbl_met

    implicit none

    ! Locals
    integer             :: iRetCode
    character(len=16)   :: sBuffer
    real                :: temp     ! Temperature, in °C
    real                :: relh     ! Relative humidity, in %

    ! Get temperature and relative humidity from the command line (or print
a helpful
    ! message if something looks wrong
    if(command_argument_count() /= 2) then
```

```
        call helpfulMessage()
        stop
    end if
    call get_command_argument(1, sBuffer)
    read(sBuffer, *, iostat=iRetCode) temp
    if(iRetCode /= 0) then
        print *, "Error: invalid temperature!"
        print *
        call helpfulMessage()
        stop
    end if
    call get_command_argument(2, sBuffer)
    read(sBuffer, *, iostat=iRetCode) relh
    if(iRetCode /= 0) then
        print *, "Error: invalid relative humidity!"
        print *
        call helpfulMessage()
        stop
    end if

    ! Perform the calculation using pbl_met, and print it on the fly
    print *, "Dew point temperature = ", DewPointTemperature(temp + 273.15,
relh) - 273.15, " °C"

contains

    subroutine helpfulMessage()

        ! Routine arguments
        ! --none--

        ! Locals
        ! --none--

        ! Print some helpful info
        print *, "dewpoint - Simple dew point calculator given temperature
in °C"
        print *, "           and relative humidity in %"
        print *, ""
        print *, "Usage:"
        print *, ""
        print *, "  ./dewpoint <temperature> <relative_humidity>"
        print *, ""
        print *, "The program prints the answer, in °C."
        print *, ""
        print *, "This is a simple example of _pbl_met_ use."
        print *, ""

    end subroutine helpfulMessage
end program DewPoint
```

Notice the first four lines, missing in the preceding version: they are comments, illustrating synthetically what can the program do for its users.

Compiling the program

The program seen so far is in source form. For it to be useable, it must first of all be compiled to executable code.

Let's now assume you've installed *pbl_met* as shown in its manual, and copies the example subdirectory as well.

In this case, all you have to do is to open a terminal windows, navigate to the `example` directory, send it the command

```
make
```

and wait until compilation completes. If you have not seen error messages, you can check you really obtained the executable.

Running the program

Easy job. From the terminal window you send the command

```
./dewpoint <Temp> <RH>
```

on UNIX and Linux systems (including Mac OS/X), and

```
dewpoint <Temp> <RH>
```

under Windows.

The two placeholders, `<Temp>` and `<RH>` are a way I often use when documenting procedures, and stands for the umeric values of temperature and relative humidity respectively.

Here's one example:

```
./dewpoint 50 90.0
```

Once the command above is given, the dewpoint calculator executes and (almost instantaneously) prints the following response:

```
  Dew point temperature =    47.1846008      °C
```

How to (seriously) test the program

The single run I've shown in the preceding session does prove nothing about the correctness of the dewpoint calculator - at most, it tells us whether the single evaluation at that specified temperature and relative humidity is sufficiently close to a reference value (it does, incidentally).

In order to gain sufficient confidence, you should perform *various* tests, both with sensible (within physical limits) and nonsensical data, compare the results with a suitable reference, and check the difference is small enough.

Criteria must be established, to avoid testing the calculator with any possible and impossible input values. This "brute force" approach is unfeasible however, the nummber of temperature / humidity combinations being ridiculously large (and many orders of magnitude beyond the possibilities of even the largest extant computers).

A more astute way is to perform tests on a carefully selected sample of cases, eventually small, chosen to exercise any path in code execution. This approach is often feasible, and in our case also very quick, the dew point *pbl_met* routine containing very few paths.

But before even performing the first test, an appropriate *reference* should be used. The decision is sometimes not simple, especially when the choice of reference values or functions is not unique. The accuracy of this reference should better be finer than the accuracy of the actual formula used in *pbl_met*. But, can be done after some understanding of the problem terms is constructed.

One advantage of using a library as *pbl_met* is that a large part of the testing activity - namely, that related to the library functions and elements themselves - has been already made, so that all you remain to do is checking "only" your application. We'll not really see how to do - tests have already been made on the dewpoint calculator for you - but its worth is easily understood by developers.