The *pbl_met* library

State and composition, on 09. 06. 2018

M. Favaron

Table of Contents

Current state

To date, development is still in active progress, with plans to arrive at version 0.01 by 1nd-2018.

In this moment the development is carried on by Servizi Territorio srl personnel, as an investment whose primary aim is to made public, accessible for inspection and freely useable the computational core of the meteorological stations and data processing software they sell (you may get some information about them by having a look to their site, http://www.serviziterritorio.it, or contacting them at the e-mail address info@serviziterritorio.it).

In future, the author wish other people will add to the current developer base. Contributions are welcome with respect to new functions addition, but also on test, documentation, collection of success cases.

Composition

What it actually is

From the end user perspective, the *pbl_met* is a large Fortran module (and library object file) containing constants, functions and subroutines covering the area of micro-meteorological data processing in quite a broad sense.

The module is internally organized in a set of sub-modules, each devoted to a specific theme. The planned sub-modules are listed in the following table.

Module or sub-module	Thematic contents
pbl_met.f90	Container, and unique user access point. Users only need a copy of **pbl_met.mod** and **pbl_met.a** (or **pbl_met.so**, or whatever the user operating system library name is) to use the *pbl_met*.
pbl_base.f90	Basic symbols, operators and procedures used by all other sub-modules.
pbl_time.f90	Time stamps (both the legacy integer 32 bit and the new 64 bit floating point forms), and things time-related as computing sunset/sunrise, calculating solar elevation angle and declination, and similar things.
pbl_stat.f90	Statistical procedures including descriptive statistics (mean, minimum, maximum, standard deviation, skewness, kurtosis, quantiles, median) and time series analysis (auto- and cross-correlation, partial autocorrelation, basic time series management).
pbl_thermo.f90	Thermodynamics, psychrometry, radiometry, and broadly speaking all what related to solar energy flow and its perception, including thermal comfort and climate-related health.
pbl_evtrn.f90	Evapotranspiration, following the ASCE Reference Evaporation Equation approach.
pbl_wind.f90	Anemology, from the basics to eddy covariance.

Module or sub-module	Thematic contents
pbl_turb.f90	Turbulence-related indicators, based on Monin-Obukhov similarity theory.
pbl_depth.f90	Estimation of the PBL thickness ("mixing height") under stable, neutral and convective conditions.

During development, some of these modules may still be mere placeholders: they will be filled, as development will progress.

What you will not find in *pbl_met*

The *pbl_met* is dedicated at meteorological data processing, and while processing real-world data the need always arises to read them from an input, perform some cleaning, and render them to some form when finished.

All these operations, although indispensable, are extremely application and instrument dependent, and accomplishing them all would be an endless task – technology and formats also evolve continuously.

The rendering of processed data Is also situation-dependent. Imagine, for example, you want to send processed data to a dispersion model. First you will choose one, and any dispersion model has its own file formats and conventions. Once you have decided the model of your choice is, e.g, Calpuff, then you will still have to choose the format details, and maybe to adapt to undocumented quirks and, yes, bugs specific to a version.

All these possibilities would, if implemented directly in the *pbl_met*, turn the latter into a bludgeoning mess of application-specific code with some high value processing functions interspersed.

To avoid this problem, we decided to leave all nitty-gritty specifics out of the *pbl_met*. As a library, the *pbl_met* automatically targets developers and scientists as an audience, and you, as a developer and/or scientist, are surely able to read, clean and write *your* data according to *your* needs.

Some history

The new *pbl_met* inherits the spirit (and the code base) from an old product, named PBL_MET, developed in-house in Servizi Territorio by Roberto Sozzi and Daniel Fraternali, and initially used to support the company consultancy activities.

In 1993 the former PBL_MET library was officially presented at the 2nd Workshop on Harmonisation within Atmospheric Dispersion Modelling for Regulatory Purposes, held in Manno, Switzerland.

Who is curious about this may have a look to the URL

http://www2.dmu.dk/atmosphericenvironment/Harmoni/manpaper.htm

where the papers presented were listed.

From 1994 Servizi Territorio began selling the PBL_MET as a product, with discrete success, having universities and environmental agencies in Europe as their main clients.

This old version of PBL_MET, now declassified, was also used extensively by Servizi Territorio to mainly write met processors for feeding meteorological data to dispersion models, and similar

things. The code base found its way trickling into the Mimolab “modeling lab” and the SARA micro-meteorological station data center, which Servizi Territorio developed within the LIFE/MoNIQA project, and constituted the backbone of the many versions of the ultrasonic anemometer data processing system ECOMET, which Servizi Territorio developed in support of research activities by various research institutions. And, others – the list is quite long and, I suppose, close to irrelevance to you, the reader.

The legacy PBL_MET code may still be found (without manual) in the *PBL_MET_old* directory in the GitHub space for *pbl_met*.

Reasons for a change

As an internal product, the old PBL_MET had a defect, which detracted from its main benefits: it was not possible to an environmental protection agency to inspect it, and check its assumptions made sense to them. This is hardly acceptable in regulatory applications.

Meanwhile other scopes became important, as the need of treating large volumes of data from new instrument types like three-dimensional ultrasonic anemometers and SODAR, SODAR/RASS systems. As the cost of these instruments decreased, the priority of PBL_MET shifted from “allowing to estimate quantities the mechanical sensors cannot sample” to “supporting new instruments by mining useful quantities from their data sets”.

Other changes of course occurred in user perception, and a full list may not be appropriate here. It is worth mentioning, among them, an increased demand for sophisticate *real-time* data processing.

All these changes suggested the old PBL_MET was to be upgraded, and made accessible for free study and use.

Why Fortran

On these days languages like Python and R have gained momentum and are widely diffused among scientific users.

Yet consider this: to date, all meteorological codes and almost all dispersion models are written in some version of the Fortran language.

So, if you wish to integrate your code into an existing legacy application (written in Fortran), of interchanging data easily with it, then you need to use Fortran. Sure you might use other languages as well, but the task of making them inter-operate with existing Fortran code would be a daunting (computer science) task, and you would soon end up writing most of your code trying to let Python calls be executed in Fortran the exactly way you should, fixing interoperability nasty bugs, and fighting with versions and variants. Instead of focusing on meteorology.

The reason so much scientific code is “still” written in Fortran is manifold.

First, the scientific community is used to Fortran: knows it, and peruses it fluently.

Second, millions lines of legacy code would make a gargantuan task to port them to another, supposedly “better” language.

Third, Fortran has a simple structure, allowing little fancy in the direction of breeding nasty bugs (in comparison, think the infinite possibilities you have to producing subtly not-working code a language like C gifts you through the mechanism of pointers).

Fourth, Fortran compiled code is extremely efficient, in both execution time and memory resources.

Fifth, Fortran allows to natively address parallel execution on SIMD and multi-core CPUs (and to date practically all of them are), using language constructs which are easy to learn and remember, efficient, and reliable.

Sixth and last, but not for importance, Fortran is a *modern* language. Many little-informed people, mainly from the computer science and IT arena, claim Fortran "is dead". This is, simply speaking, plain false. Today's Fortran does not constrain you any more in the rigid "fixed form" code it had until 1990. In addition, it has powerful object oriented constructs (would you need using them), and provisions for coding-at-large. It permits using pointers, although in a way much safer than C. It allows you to perform single-line matrix and vector computations similar to Matlab's, or to craft parallel code. It's also an *evolving* language, with the standard Fortran 2018 under way. Learning it is still a must for *scientific* users, and will remain so for decades in future. Claims of death or irrelevance is more a consequence of the traditional lack of communication between the scientific and computer science / IT communities, than of reality.

So, when deciding in what language to code the new *pbl_met*, we found all work done by the real world itself, and modern Fortran emerged as the natural answer.

Code documentation approach

This is an example *pbl_met* function, whose purpose is to compute the product $\rho \cdot C_p$, and whose text you may find in source file **pbl_thermo.f90**:

```
! Product of air density and the constant-pressure atmospheric thermal capacity,
! given dry bulb temperature and atmospheric pressure.
!
FUNCTION RhoCp(Td, Pa) RESULT(rRhoCp)

      ! Routine arguments
      REAL, INTENT(IN)             :: Td         ! Dew point temperature (K)
      REAL, INTENT(IN), OPTIONAL   :: Pa         ! Air pressure (hPa)
      REAL                         :: rRhoCp     ! Product of air density and
                                                 ! constant-pressure thermal
                                                 ! capacity

      ! Locals
      REAL  :: Rho
      REAL  :: Cp

      ! Compute the information desired
      IF(PRESENT(Pa)) THEN
            ! Pressure is available: use complete formula
            Rho = AirDensity(Td, Pa)
            Cp  = 1005.0 + (Td - 250.0)**2/3364.0     ! From Garratt, 1992
            rRhoCp = Rho * Cp
      ELSE
            ! Pressure not available on entry: use the simplified relation
            rRhoCp = 1305. * 273.15/Td
      END IF

END FUNCTION RhoCp
```

And this is a similar purpose function, from the legacy PBL_MET code:

```
C&@   ======================================================================
      Function RHOCP(T)
C     ======================================================================
*     This routine returns the value of the product of air density and
*     constant pressure specific heat.
*
```

```
*     Input data:
*                T = air temperature (Kelvin)
*
*     Output data:
*                RHOCP = density * specific heat
*                        -9999. if error
*
c     Servizi Territorio coop a r.l.- Cinisello B. (MI) Italy - 1993
c     ---------------------------------------------------------------
      if(T.LT.0.) then
                   RHOCP = -9999.
                   return
      endif
c
      RHOCP = 1305. * 273.16/T
C
      return
      end
```

Although not entirely representative of the average *pbl_met* function (RhoCp indeed is very small and "simple"), it still allows to notice some things:

- The new *pbl_met* code has "lighter" comments, compared to the legacy PBL_MET. That a variable is input or output is not just said in words and hopefully followed in code (something you should verify yourself when using the legacy PBL_MET), but is *declared* using Fortran constructs: so, it is the *Fortran compiler* who checks whether an input is really an input, or an input/output or something else.
- The *pbl_met* version of **RhoCp** is backward-compatible with its legacy counterpart, but also *extends* It through an additional function argument, the air pressure, which is declared OPTIONAL. Once again, there is no comment explaining that we, the library author, wish you, the library user, will use the pressure or not: this is written in Fortran, and checked/enforced by the compiler.
- The *pbl_met* function uses the now-standard free-form source convention, while the old PBL_MET version still uses fixed-form.
- All variables are declared in the *pbl_met* version, with a comment on right illustrating the variable meaning when relevant. In the old PBL_MET all variables are "declared implicitly", so to check which they are you should read all the function code and take notes yourself; if you mis-type a variable and use implicit declaration you may then incur in bugs which can be extremely difficult to find.

In short: the *pbl_met* code is written assuming *code itself is the first-instance documentation*. Fortran constructs, and not wishful comments, are used to say important things like which the inputs and outputs are. Long, self-descriptive variable names are used instead of short ones, unless they correspond (as in preceding case) to symbols widely accepted by the scientific or engineering community.

Making the code as self-commenting as possible diminishes the likelihood that purely textual comments, if present, say something different from what codes really does – a thing, you may easily imagine, whose consequences are of, at best, generating a lot of confusion.

Last, you may notice the comment line stating a sort of ownership of legacy RhoCp by a company, in the revised version is gone. The reason dates back to the time when PBL_MET was proprietary code, and is no longer needed today, the new *pbl_met* being open source. Besides, the "ownership claim" actually claimed nothing: it is not a valid copyright statement, nor the indication of a

trademark, but just the name of a company which may have been (but also not) involved in the PBL_MET development, or documentation, or testing, or …

Of course, the coding style adopted demands some knowledge of modern Fortran, and some attention: code is designed to be studied, in line with the spirit of open source. Study promotes real and unambiguous understanding, but always demands an effort. We assume you, as a scientific user, are knowledgeable and motivated to engage in this study, if you need to understand. (As a crude joke I may add that if you need to understand, but are not knowledgeable enough, expect an immediate effortless result, and are not willing to admit and correct your lack of knowledge, then you are not a likely user of *pbl_met*, but rather one form or the other of "end user", and would use your time in a better way by looking elsewhere).

Current library contents

Of course, by this report date.

Source	**Category**	**Contents**
pbl_base.f90	Symbols	`real, public, parameter :: NaN = Z'7FC00000'` `real(8), public, parameter :: NaN_8 = Z'7FF8000000000000'`
	Operators	`operator(.valid.)(value) result(lIsValid)` `operator(.invalid.)(value) result(lIsInvalid)`

	Procedures	`subroutine toUpper(sString)`
		`subroutine toLower(sString)`
		`function gammaP(a, x, iMaxIterations) result(gP)`

	Members of type(IniFile)	`function iniRead(this, iLUN, sIniFileName) result(iRetCode)`
		`function iniDump(this) result(iRetCode)`
		`function iniGetString(this, sSection, sKey, sValue, sDefault) result(iRetCode)`
		`function iniGetReal4(this, sSection, sKey, rValue, rDefault) result(iRetCode)`
		`function iniGetReal8(this, sSection, sKey, rValue, rDefault) result(iRetCode)`
		`function iniGetInteger(this, sSection, sKey, iValue, iDefault) result(iRetCode)`

pbl_depth.f90		`To date still not filled (will contain mixing height related things)`

pbl_depth.f90	Procedures	`function ColtureLAI(rVegetationHeight, iColtureType) result(rLAI)`
		`function AerodynamicResistance(zw, zh, u, h) result(ra)`
		`function Evapotranspiration(Pres, Temp, Vel, Rn, G, es, ea, Zr, vegType) result(ET)`

Source	Category	Contents
pbl_simil.f90		To date still not filled (will contain Monin-Obukhov similarity related things)
pbl_stat.f90	Procedures	subroutine RangeInvalidate(rvX, rMin, rMax) subroutine PairInvalidate(rvX, rvY) subroutine RangeClip(rvX, rMin, rMax) function GetValidOnly(rvX) result(rvValidX) function Mean(rvX, rValidFraction) result(rMean) function StdDev(rvX, rMeanIn, rValidFraction) result(rStdDev) function Skew(rvX, rMeanIn, rStdDevIn, rValidFraction) result(rSkewness) function Kurt(rvX, rMeanIn, rStdDevIn, rValidFraction) result(rKurtosis) function Cov(rvX, rvY) result(rCov) function QuantileScalar(rvX, rQuantile, iType) result(rQvalue) function QuantileVector(rvX, rvQuantile, iType) result(rvQvalue) function AutoCov(rvX, rvACov, iType) result(iRetCode) function AutoCorr(rvX, rvACorr, iType) result(iRetCode) function CrossCov(rvX, rvY, rvCCov, iType) result(iRetCode) function CrossCorr(rvX, rvY, rvCCorr, iType) result(iRetCode) function PartialAutoCorr(rvACov) result(rvPACorr) function EulerianTime(rDataRate, rvX, rMaxEulerianTime, rEulerianTime, rvACorr) result(iRetCode) function RemoveLinearTrend(rvX, rvY, rMultiplier, rOffset) result(iRetCode)
	Members of type(TimeSeries)	function tsCreateEmpty(this, n) result(iRetCode) function tsCreateFromTimeSeries(this, ts, lForceWellSpacedMonotonic) result(iRetCode) function tsIsEmpty(this) result(lIsEmpty) function tsCreateFromDataVector(this, rvValues, rTimeFrom, rDeltaTime) result(iRetCode) function tsCreateFromTimeAndDataVectors(this, rvTimeStamp, rvValues) result(iRetCode) subroutine tsTimeShift(this, deltaTime) subroutine tsTimeReorder(this) function tsGetSingleItem(this, iItemIdx, rTimeStamp, rValue) result(iRetCode)

Source	Category	Contents
		function tsGetTimeStamp(this, rvTimeStamp) result(iRetCode) function tsGetValues(this, rvValues) result(iRetCode) function tsGetTimeSpan(this, rMinTimeStamp, rMaxTimeStamp) result(iRetCode) function tsGetTimeSubset(this, ts, timeFrom, timeTo) result(iRetCode) function tsGetMonth(this, ts, iMonth) result(iRetCode) function tsPutSingleItem(this, iItemIdx, rTimeStamp, rValue) result(iRetCode) function tsSize(this) result(iNumValues) function tsIsSameTimes(this, ts) result(lTimesAreSame) subroutine tsSummary(this, iNumValues, rValidPercentage, rMin, rMean, rStdDev, rMax, rSkew, rKurt) subroutine tsRangeInvalidate(this, rMin, rMax) function tsTimeMonotonic(this) result(lIsMonotonic) function tsTimeQuasiMonotonic(this) result(lIsMonotonic) function tsTimeGapless(this) result(lIsGapless) function tsTimeWellSpaced(this, rTimeStep, iNumGaps) result(iWellSpacingType) function tsAggregateLinear(this, iTimeDelta, iFunction, ts, ivNumDataOut) result(iRetCode) function tsAggregateLinear2(& this, iTimeDelta, & rvTimeStamp_Reduced, rvMean, & rvStDevOut, rvMinOut, rvMaxOut, ivNumDataOut &) result(iRetCode) function tsAggregatePeriodic(& this, iPeriodLength, iTimeDelta, & rvMean, & rvStDevOut, rvMinOut, rvMaxOut, ivNumDataOut &) result(iRetCode) function tsMovingAverage(this, ts, rTimeWidth, iMode) result(iRetCode)
pbl_thermo.f90	Procedures	function ClearSkyRg_Simple(Ra, z) result(Rso) function ClearSkyRg_Accurate(timeStamp, averagingPeriod, lat, lon, zone, Pa, Temp, Hrel, Kt_In) result(Rso) function ExtraterrestrialRadiation(timeStamp, averagingPeriod, lat, lon, zone) result(ra)

Source	Category	Contents
		function NetRadiation(Rg, albedo, fcd, Ea, Ta) result(Rn) function Cloudiness(rvElAng, rvRg, rvRg3, rSunElevThreshold, rvFcd) result(iRetCode) FUNCTION WaterSaturationPressure(Ta) RESULT(es) function E_SAT_1(T) result(rEsat) function PrecipitableWater(Ea, Pa) result(W) FUNCTION D_E_SAT(T) RESULT(DEsat) FUNCTION WaterVaporPressure(Tw, Td, Pa) RESULT(Ew) FUNCTION RelativeHumidity(Tw, Td, Pa) RESULT(RelH) FUNCTION AbsoluteHumidity(Td, Ea) RESULT(RhoW) FUNCTION AirDensity(Td, Pa) RESULT(Rho) FUNCTION RhoCp(Td, Pa) RESULT(rRhoCp) function LatentVaporizationHeat(rTemp, iCalculationType) result(rLambda) FUNCTION WetBulbTemperature(Td, Ur, Pa, RoughTol, FineTol, MaxIter, Method) RESULT(Tw) function AirPressure1(rZ) result(rPk) function AirPressure2(rZ, rTemp, rZr, iCalculationType) result(rPk) function VirtualTemperature(Temp, ea, P) result(Tv) FUNCTION DewPointTemperature(Td, Ur) RESULT(Dp) FUNCTION SonicTemperature(Td, Ur, Pa, RoughTol, FineTol, MaxIter, Method) RESULT(Ts) function GlobalRadiation_MPDA(C, sinPsi) result(Rg) function CloudCover_MPDA(Rg, sinPsi) result(C) function NetRadiation_MPDA(land, albedo, Td, Rg, C, z0, zr, vel) result(Rn) function BruntVaisala(Td, z) result(N)
pbl_time.f90	Procedures	function JulianDay(iYear, iMonth, iDay) result(iJulianDay) subroutine UnpackDate(iJulianDay, iYear, iMonth, iDay) function Leap(ia) result(isLeap) function DoW(iJulianDay) result(iDayOfWeek) function DoY(ia,im,id) result(iDayOfYear) subroutine PackTime(iTime, iYear, iMonth, iDay, iInHour, iInMinute, iInSecond) subroutine UnpackTime(iTime, iYear, iMonth, iDay, iHour, iMinute, iSecond) function calcJD(year, month, day) result(jd) function calcTimeJulianCent(jd) result(T)

Source	Category	Contents
		function SunRiseSunSet(yy, mo, dy, lat, lon, zone) result(sunRiseSet) function SinSolarElevation(yy, mo, dy, hh, mm, ss, lat, lon, zone, averagingPeriod) result(sinBeta) function SolarDeclination(yy, mo, dy) result(sunDecl) subroutine UnpackDate8(iJulianDay, iYear, iMonth, iDay)
	Members of type(DateTime)	function dtFromEpoch(this, rEpoch) result(iRetCode) function dtToEpoch(this) result(rEpoch) function dtToIso(this) result(sDateTime) function timeEncode1(rvTimeStamp, iPeriodLength, iStepSize, ivTimeCode) result(iRetCode) function timeEncode2(ivTimeStamp, iPeriodLength, iStepSize, ivTimeCode) result(iRetCode) function timeGetYear1(rvTimeStamp, ivYear) result(iRetCode) function timeGetYear2(ivTimeStamp, ivYear) result(iRetCode) function timeGetMonth1(rvTimeStamp, ivMonth) result(iRetCode) function timeGetMonth2(ivTimeStamp, ivMonth) result(iRetCode) function timeGetYearMonth1(rvTimeStamp, ivYearMonth) result(iRetCode) function timeGetYearMonth2(ivTimeStamp, ivYearMonth) result(iRetCode)
pbl_turb.f90		To date still not filled (will contain turbulence related things)
pbl_wind.f90	Procedures	function PolarToCartesian2(polar, interpretation) result(cartesian) function PolarToCartesian3(polar, interpretation) result(cartesian) function CartesianToPolar2(cartesian, interpretation) result(polar) function CartesianToPolar3(cartesian, interpretation) result(polar) function ClassVelScalar(vel, rvVel) result(iClass) function ClassVelVector(vel, rvVel) result(ivClass) function ClassDirScalar(dir, iNumClasses, iClassType) result(iClass) function ClassDirVector(dir, iNumClasses, iClassType) result(ivClass) function VectorDirVel(rvVel, rvDir) result(polar) function ScalarVel(rvVel) result(vel)

Source	Category	Contents

		`function UnitDir(rvDir) result(dir)`
		`function WindRose(vel, dir, rvVel, iNumClasses, iClassType, rmWindRose) result(iRetCode)`
		`function VelDirMean(vel, dir, scalar, rvVel, iNumClasses, iClassType) result(rmMean)`
		`function VelMean(vel, scalar, rvVel) result(rvMean)`
		`function DirMean(dir, scalar, iNumClasses, iClassType) result(rvMean)`

Testing

Testing is necessary for a library like *pbl_met*, in which advanced topics of the physics of the Planetary Boundary Layer are addressed along with statistics and number-crunching-ware, to mention just two.

Test is organized per-sub-library and, within of any sub-library, per procedure.

You may find test code at URL https://github.com/serv-terr/pbl_met/tree/master/test, and sub-directories.

To date the following test procedures exist (and grow):

t_pbl_base.f90

t_pbl_stat.f90

t_pbl_thermo.f90

t_pbl_time.f90

t_pbl_wind.f90

Others will add, with time.

Development roadmap

To date, things are quite simple: the next step is filling the parts of library which are still empty, using extant legacy PBL_MET and other old codes as a basis, filling in the gaps with new code.

Indicatively by Spring 2019 the library core will be complete, with tests. This will be version 0.1, and the full-fledge development management using GitHub instruments will start. In particular, pull requests and issues will be evaluated and managed regularly.

Until then, we will pursue an “open development” informal path in which I can imagine most of the contribution will come from the current contributors, with both original and refactored legacy code added and tested, and development will concentrate on providing demonstrational material, example code, and other things alike.

Although still missing an official version name, we encourage you to get the *pbl_met* code, use it, and let us know what you think about, including proposal of new parts, and constructive critics. In this moment the number of developers involved is still very small (2), both having other things to do in order to survive, so, please consider we cannot provide direct assistance on mundane things as how to compile, or what to do with the library. In case you have a strong need we can’t maybe

fulfil in this initial phase, please consider contributing the solution (source code included) on yourself.

If you work in a university, please consider quoting us if you use the *pbl_met* for a paper or presentation. In the moment you may simply refer to the GitHub project space URL, that is

https://github.com/serv-terr/pbl_met

In the next future a paper is planned being written, and will be the entry point for any citation. If you use our work, and make us know, we will be encouraged to continue using part of our spare time to proceed.

In future, the path of donations will also be explored – to date the only "donor" is Servizi Territorio srl, but having others interested people and companies providing resources may help in doing better and more useful things in a more systematic manner.