Revision of EMEP deposition modules

Dave Simpson

February 9, 2005

1 Introduction

The deposition code which was constructed prior to 2004 was designed to process meteorological data as available for the EMEP model's 50x50 km^2 grid-squares. A number of steps were needed in order to get from this 'raw' data to meteorological data as required over a specific landuse class for the deposition code. These processing steps complicated the original code and do not allow for an easy use of the same code for other users, for example with real meteorological data from a field-site.

The code is therefore being re-written in order to generate a more modular system which can be used both within the EMEP model and by external users. This re-coding is being done in several steps, building up first the base-modules which later code will require:

Step 1 Gst_ml module

- embodies the calculation of the environmental factors (f_T, f_light, etc.) and calculates g_stomatal and g_sun. A test-driver code `Test_Gsto` sets values of T, PAR, etc. and calls Gsto_ml.

Step 2 Radiation_ml

- calculates radiation terms, from zenith angle to PAR. Output example gives changes in g_sto and g_sun over one day.

Step 3 Rsurface_ml + Rb_ml

- calculates Rsur for wet and dry surfaces, as well as Rb. Now the call to g_stomatal is from within the Rsurface subroutine.

Step 4 Read in hourly NWP met data and convert to local meteorology ... to be done ...

With each step there is a test-driver to allow the code to be run with simple inputs. For example, `Test_Gsto.f90` sets some values of temperature, humidity, LAI, etc. and calls `Gsto_ml`. `Test_Rad.f90` calls the radiation routines every hour over a day to calculate g_sto and g_sun.

2 Code structure

Modern Fortran stresses the use of modules for storing data and for keeping related-subroutines together. I have therefore attempted to structure the new code around a few modules - keeping separate themes in the different modules. For example, all parameters needed to calculated f_temp, f_swp, etc. are essential to the g_stomatal calculation, and so the module Gsto_ml contains subroutines to read the parameters need for the f-factors, and to calculate these. This module makes no assumptions about phenology or vegetation characteristics - these are left for the user to specify.

A few modules serve as the basis of the new code. One module in particular, `LocalVariables_ml` is designed simply as a container of data relavent to the local landuse, and which can be used by other modules in many ways. In fortran'90, any routines which has `useLocalVariables_ml` can read and/or change the LAI values stored in this module.

This organisation means that the user can choose how to set local variables in many ways. For example, the EMEP model might have many steps to get from NWP data to local values of u_*, whereas another user has measured values which can be used directly. The code makes no assumptions in the first instance about where local u_* values come from.

Further, we do not pass specific meteorological variables through subroutine arguments, but rather just 'use' `LocalVariables_ml`. This allows use to call say `Rsurface_ml` with very few arguments (probably just land-use code and

debug-flags, as well as outputs Rsur, Rb), so that there is no restriction forced on the meteorological data which `Rsurface_ml` can use. As long as the data is contained in `LocalVariables_ml` it can be used or neglected at will.

3 Modules

The following sections give more details on each module, and list the 'public' routines and variables. Modules are listed loosely in order from the simple base modules to the more complex modules which depend on these.

3.1 LocalVariables_ml

– stores meteorology and local vegetation characteristics for a specific land-cover. For example, Ts_C, u_*, RH, h, LAI are stored here. The local values of PARsun, PARshade and LAIsunfrac are also stored here since these vary with landuse LAI and albedo.

```
   real, public, save ::  &
        Ts_C          &      ! Surface temperature in degrees C
       ,psurf         &      ! Surface pressure, Pa
       ,precip        &      ! Precipitation at ground, mm/hr
       ,wetarea       &      ! Area (fraction) of grid square assumed wet
       ,rh            &      ! Relative humidity, fraction (0-1)
       ,vpd           &      ! Vapour pressure deficit  (kPa) ! CHECK UNITS
       ,swp           &      ! Vapour pressure deficit  (kPa) ! CHECK UNITS
       ,cl            &      ! Cloud-cover
!
! Micro-met
       ,ustar         &      ! friction velocity, m/s
       ,invL          &      ! 1/L, where L is Obukhiov length (1/m)
! Vegetation
       ,LAI           &      ! Leaf area index (m2/m2)
       ,SAI           &      ! Surface area index (m2/m2)
       ,hveg          &      ! Height of veg.      (m)
       ,d             &      ! displacement height (m)
       ,z0            &      ! roughness length    (m)
!
! Radiation
       ,PARsun        &      ! photosynthetic active radn. for sun-leaves
       ,PARshade      &      !  " " for shade leaves
       ,LAIsunfrac           ! fraction of LAI in sun
```

```
! Chemistry
       ,so2nh3ratio         !  for CEH deposition scheme

   logical, public, save :: &
       is_wet              !  true if precip > 0
```

3.2 LandClasses_ml

– defines the number of possible land-classes (NLANDUSE), and gives some basic codes and characteristics - for example if the landuse is forest, crop, or water. Landuse-associated datafiles needed by other modules must contain the same number of land-classes and use the same landuse-code. (A user will usually work with just one of the defined land-use classes - there is no requirement to use all.)

```
type, public :: land_class
   character(len=3)  :: code
    character(len=13) :: name
    logical           :: Gcalc   ! calculate Gsto?  Set F for bulk Rs
    logical           :: forest  !
    logical           :: crops   !
    logical           :: water   !
    real              :: b       ! in-canopy resistance factor
    real              :: albedo  ! fraction
    real              :: RgsS    !
    real              :: RgsO    !
 end type land_class

 type(land_class), public, dimension(NLANDUSE) :: landuse
```

3.3 My_DryDrep_ml

– specifies the gases to be used, and the relation to the Wesely indices. Here we just set for ozone, using:

```
  integer, public, parameter :: NDRYDEP_CALC = 1

  integer, public, parameter, dimension(NDRYDEP_CALC) :: &
       DRYDEP_CALC = (/ WES_O3 /)
```

3.4 Io_ml

– provides some routines to check, open and close files. Ensures that all required input files are actually present, and simplifies reading files where the first few lines

(headers) should be skipped.

```
public :: check_file   !  checks that file exists and stops if required
public :: open_readfile    !  checks that file exists and opens if required
public :: wordsplit    !  Splits input text into words

logical, public :: fexist                       ! true if file exists
integer, public, parameter :: NO_FILE = 777     ! code for non-existing file
integer, public, save :: ios                    ! i/o error status number
character(len=120), public :: io_msg            ! i/o error message
```

3.5 Radiation_ml

– provides essential radiation terms and consists of a number of routines to calculate these.

From the code:

```
!/ Subroutines:
public :: ZenAng         ! => coszen=cos(zen), zen=zenith angle (degrees)
public :: ClearSkyRadn   ! => irradiance (W/m2), clear-sky
public :: CloudAtten     ! => Cloud-Attenuation factor
public :: CanopyPAR      ! => sun & shade PAR  values, and LAIsunfrac

!/ Functions:
public :: daytime        ! true if zen < 89.9 deg


real, public, save :: solar       ! => irradiance (W/m^2)
real, public, save :: Idrctn      ! => irradiance (W/m^2), normal to beam
real, public, save :: Idfuse      ! => diffuse solar radiation (W/m^2)
real, public, save :: Idrctt      ! => total direct solar radiation (W/m^2)

real, public, save :: zen         !   Zenith angle (degrees)
real, public, save :: coszen      ! = cos(zen)
```

3.6 Gsto_ml

– Conatins the data and routines needed to calculate stomatal conductance:

```
!/--------- Subroutines -------------------------------------------------

public :: Init_gsto       !- Reads in f-factors, Initialises
public :: g_stomatal      !- produces g_sto and g_sun
```

```
  !/--------- Variables available from this module -----------------------

  real, public, save  :: &
          g_sto          & ! stomatal conductance (m/s)
       ,  g_sun            ! g_sto for upper-canopy sun-leaves


  real, public, save :: & !/ g_sto factors (0-1):
        f_phen     &            ! phenology (age)
       ,f_temp     &            ! temperature
       ,f_vpd      &            ! vapour pressure deficit
       ,f_light    &            ! light
       ,f_swp                   ! soil water potential
```

Also, for each landuse we set the various parameters, e.g.:

```
type(gf), private, parameter, dimension(NLANDUSE) :: g =(/ &
! g_sto  variables read previously from Gsto_inputs.dat   ..............
!-------------------------------------------------------------------------------
!   LU     gmax  fmin flight     ftemp   fphe Sfphen Efphen   fVDP          fSWP
!                            min opt max min  len   len   max    min    SWPmax  P
!-------------------------------------------------------------------------------
 gf("CF ",160,  0.1  ,0.0083,  1,18,36,  0.2, 130, 130,  0.6  , 3.3,  -0.76, -1.2
,gf("DF ", .....
```

3.7 CEH_ml

— For other gases than ozone. Ignore for now.

3.8 Rb_ml

Straightfoward - calculates Rb

3.9 Rsurface_ml

– Calculates `Rsur_dry` and `Rsur_wet` for all gases specified in `My_DryDep_ml`.

From the code:

```
public   ::  Rsurface

  subroutine Rsurface(lu,debug_flag,Rsur_dry,Rsur_wet,errmsg)
```

```
! Output:

  ! bulk canopy surface resistances (s/m):
   real,dimension(:),intent(out) :: Rsur_dry   !  Rs  for dry surfaces
   real,dimension(:),intent(out) :: Rsur_wet   !  Rs  for wet surfaces
   character(len=*), intent(out) :: errmsg
```

3.10 Makefiles

I have made small Makefiles for each step also. Thus, to run `Test_Rsur` I would do:

```
cp Makefile.Rsur Makefile
make
Test_Rsur
```

4 Step2 vs. Step1

As well as the `Radiation_ml`, changes were made to `Gsto_ml` (to use `Io_ml`).

5 Step3 vs. Step2

New modules `Wesely_ml`, `CEH_ml`, `My_DryDep_mk`, `Rb_ml` and `Rsurface_ml`.

6 F90 vs. F Language

All code is written for the F compiler, allowing it to run under any f90/f95 system.

F90 is a huge language, which is backwardly compatible with Fortran-77, Fortran-IV and presumably Fortran-I from the 1950s. This need to keep things compatable means that code can be written in a very large number of ways - this can be confusing!

F is a new teaching language which is a pure subset of F90. This means that anything that compiles with F will compile with any F90 compiler. The reverse is not true - far from it! F uses only modern coding practices, and is very strict in

this. Sometimes the strictness is annoying, e.g. one has to write `read(unit=10,fmt=*)` instead of just `read(10,*)` as F90 would allow. `open` statements are even more wordy in F. On the other hand, some things enforced by F help any code. For example, all variables have to be declared, and all subroutine arguments defined as `intent(in)`, `intent(out)` or `intent(inout)`.

A free F compiler can be obtained at `www.fortran.com`.

A free F90 compiler can be obtained at `www.intel.com/software/products/compilers/`. This free for non-commercial use anyway. I am not sure if the Windows version is equally free - I use the Linux version. Intel also has a good debugger idb which is useful, although not perfect for F90.