RoadSurf user manual

Virve Karsisto Markku Kangas

May 8, 2023

Contents

1 Introduction **2**

2 Requirements & setup **2**

3 RoadSurf library **2**
3.1 RoadSurfVariables . . . 2
3.2 RoadSurf module . . . 2

4 RoadSurf physics **3**
4.1 Ground heat profile . . . 3
4.1.1 Volumetric heat capacity and heat conductivity . . . 4
4.2 Temperature field and layer height initialization . . . 5
4.3 Surface energy balance . . . 5
4.4 Net radiation . . . 5
4.5 Albedo . . . 6
4.6 Sensible heat flux . . . 6
4.7 Latent heat flux . . . 8
4.8 Relaxation . . . 9
4.9 Coupling . . . 9
4.10 Storage terms . . . 10
4.10.1 Precipitation type . . . 10
4.10.2 Wear . . . 10
4.10.3 Changes in storage terms . . . 11

5 Example implementation **13**
5.1 Running . . . 13
5.2 Data flow . . . 14
5.3 Input . . . 14
5.3.1 Required variables . . . 14
5.3.2 Required parameters . . . 14
5.3.3 Input for shadowing algorithm and sky view factor . . . 14
5.3.4 Input data management . . . 16
5.4 Output . . . 16
5.5 Settings . . . 16

6 How to make your own implementation **18**
6.1 Adding new input file types . . . 18
6.2 C++ part of Example 1 . . . 18
6.3 Fortran part of Example 1 . . . 20
6.3.1 Simulation structure . . . 20
6.4 Adding new input and output variables . . . 22

1 Introduction

RoadSurf is a Fortran library for predicting road conditions. The library does not contain all of the features included in the original model [1]. The included features are road surface temperature and storage term (water, ice, snow, deposit) calculation. Variables determined based on these values, such as friction and road condition, are excluded from the library. The library is designed to allow flexible implementation. It includes the subroutines for model initialization and calculation of the road surface temperature and storage terms, but the user can develop their own main program which implements them. The subroutines are separated to their own library to make it easier to keep the operational implementations up to date. Each user can have different kinds of input sources or even their own output variables. When the library is updated, it is easier to just update the library without the need to update the main program. One example of RoadSurf implementation is given in "examples" folder. See section 5 for more details. The example utilizes C++ interface. The input and output of the model are handled by the C++, whereas the actual simulation is done with the Fortran library. The code provided has also an example for running the simulations in parallel to multiple points at the same time.

2 Requirements & setup

RoadSurf library has been tested with GNU Fortran (GCC) 8.5.0. Functionality with older Fortran versions is not guaranteed. Building the RPM package requires `rpm-build`. Running the C++ code used in Example 1 requires C++11 and packages `pthread` and `jsoncpp`.

Simply compiling the library with `make` will create a `.so` file. If you wish to build a RPM package it can be done with command `make rpm`.

3 RoadSurf library

3.1 RoadSurfVariables

The library contains several derived types to store variables and parameters required used by the library subroutines. They are included in the module `RoadSurfVariables`. The derived types are:

- AtmVariables (`AtmVariables.f90.inc`): Atmospheric variables for the current simulation step
- CouplingVariables (`CouplingVariables.f90.inc`): Variables used in coupling
- InputPointers (`InputPointers.f90.inc`): Pointers to input data arrays given from the C++ interface
- GroundVariables (`GroundVariables.f90.inc`): Variables for ground properties
- InputArrays (`InputArrays.f90.inc`): Input data arrays
- InputSettings (`InputSettings.f90.inc`): Model settings given as input
- InputParameters (`InputParameters.f90.inc`): Parameters given as input
- InputRadiationCoefficient (`InputRadiationCoefficient.f90.inc`): Not used by the given example, but used when the radiation correction coefficient is predetermined instead of determining it with coupling method
- LocalParameters (`LocalParameters.f90.inc`): Point specific values given as input
- ModelSettings (`ModelSettings.f90.inc`): Model settings used in simulation
- OutputArrays (`OutputArrays.f90.inc`): Output data arrays
- OutputPointers (`OutputPointers.f90.inc`): Pointers to output data arrays for C++ interface
- PhysicalParameters (`PhysicalParameters.f90.inc`): Physical parameters used by simulation
- RoadCondParameters (`RoadCondParameters.f90.inc`): Parameters used in storage term (water, ice, snow, deposit) calculation
- SurfaceVariables (`SurfaceVariables.f90.inc`): Variables to describe and determine road surface state
- WearingFactors (`wearingFactors.f90.inc`): Variables to determine the wearing of the storage terms

3.2 RoadSurf module

RoadSurf module contains interfaces for submodules used in the road weather model simulation. Each of these submodules has an interface defined in the main module and actual procedure defined elsewhere. The submodules are:

- ConnectFortran2Carrays: Forward data given by C++ interface to Fortran arrays
- Initialization: Initialize various model parameters
- CheckValues: Check if there is abnormal values
- CouplingOperations1: Do coupling related checks and operations

- RelaxationOperations: Adjust air temperature, humidity and wind speed in forecast phase based on the latest observed values
- SetCurrentValues: Update some variables for current time step
- BalanceModelOneStep: Calculate ground temperature profile one time step forward
- SaveOutput: Save surface temperature and storage terms to model output arrays
- CheckEndCoupling: If coupling is in use and simulation is at the end of coupling phase, determine new radiation correction coefficients if necessary
- PrecipitationToStorage: Add precipitation either to snow or water storage or partially to both depending on the precipitation type
- ModRadiationBySurroundings: If sky view factor and local horizon angles are given, modify incoming radiation
- WearFactors: Determine wear factors (how fast storage terms decrease)
- RoadCond: Calculate storage terms for the next time step
- CalcAlbedo: Calculate surface albedo based on storage terms

An example implementation of road weather model simulation using these subroutines is provided in section 6.3

4 RoadSurf physics

4.1 Ground heat profile

The calculation of heat transfer between the layers is based on equation [2]:

$$\rho_g c_g \frac{\partial T(z,t)}{\partial t} = \frac{\partial}{\partial z} K \frac{\partial T(z,t)}{\partial z}, \tag{1}$$

where T is temperature, z is vertical distance in the ground, t is time, K is heat conductivity, ρ_g is density and c_g is specific heat capacity of the ground.

After integrating over the volume of the layer and time step, the equation is discretized and solved with forward difference explicit method. This results the following equation [3]:

$$T_i^{j+1} = T_i + \frac{1}{\rho_g c_g \frac{z_{i+1}-z_{i-1}}{2\Delta t}} \left(K_i \frac{T_{i+1}^j - T_i^j}{z_{i+1} - z_i} - K_{i-1} \frac{T_i^j - T_{i-1}^j}{z_i - z_{i-1}}\right). \tag{2}$$

The index j refers to time step and index i to the number of ground layer (The ground is separated to layers). The answer to the equation is calculated in multiple steps. First, some help variables are defined in function `ground_prop_init`, located in file `Initialization.f90`.

```
!Calculate depth difference between layers (mid point - mid point)
   DyC(1) = (ZDpth(2) - ZDpth(1))/2.0
   DO j = 2, Nlayers
         DyC(j) = (ZDpth(j + 1) - ZDpth(j - 1))/2.0
   END DO

   ConductivityLayers = NLayers

   !Calculate the thicknesses of individual layers
   Do j = 1, ConductivityLayers

      DyK(j) = ZDpth(j + 1) - ZDpth(j)

   END DO
```

`ZDpth` refers to layer depth and `NLayers` to the number of layers

These are used to calculate the following help variables (subroutine `calcCapDZCondDZ()` in file `BalanceModel.f90`):

```
    !Calculate help variables to use in the temperature profile calcualtion
   ground%condDZ(1) = -(ground%CC(1)/ground%DYK(1))
   ground%capDZ(1) = -(1/(ground%DYC(1)*ground%VSH(1)))

   DO j = 2, NLayerTemp

      ground%condDZ(j) = -(ground%CC(j)/ground%DYK(j))
      ground%capDZ(j) = -(1/(ground%DYC(j)*ground%VSH(j)))

   END DO
```

Where `ground` is derived type to store ground related values, `CC` is heat conductivity and `VSH` is volumetric heat capacity.

Heat flux between two layers is then calculated in subroutine `calcProfile()` (`BalanceModel.f90`):

```
  !Calculate heat flux for different layers
 do j = 1, NLayerTemp
    GFlux(j) = ground%condDZ(j)*(ground%Tmp(j + 1) - ground%Tmp(j))
 end do
 ground%GroundFlux = GFlux(3)
```

Where `Gflux` is heat flux between ground layers, and `Tmp` is ground temperature And finally the temperature for the next time step is calculated as (subroutine `calcProfile()`):

```
 !Calculate new temperatures
 do j = 1, NLayers
    ground%TmpNw(j) = ground%Tmp(j) + DTsecs*(ground%capDZ(j)*(GFlux(j) -&
     Gflux(j - 1)))
 end do
```

Where `DTSecs` is time step and `TmpNw` is ground temperature for the next time step

4.1.1 Volumetric heat capacity and heat conductivity

For temperatures above 0 °C, water density and specific heat capacity are taken to be temperature dependent according to equations (based on a tables in [4], p.149, and in [5], p.E-10) (subroutine `calcHcapHcond()` in `BalanceModel.f90`)

$$\rho_w(T) = -0.0050T^2 + 0.0079T + 1000.0028 \tag{3}$$

$$c_{pw}(T) = 1.02 \cdot 10^{-5}T^4 - 1.7169 \cdot 10^{-3}T^3 + 0.11516T^2 - 3.4739T + 4217.2, \tag{4}$$

where

ρ_w = water density (kg/m^3) ,
c_{pw} = specific heat capacity of water (kJ/kgK) ,
T = water temperature (°C) .

Below freezing temperature these have constant values ([6], p. 44)

ρ_w = 920 kg/m^3) ,
c_{pw} = 2100 (kJ/kgK) .

Volumetric heat capacity (J/m^{3}K) for water can then be calculated as (subroutine `calcHcapHcond()` in `BalanceModel.f90`):

$$(\rho c)_w = \rho_w c_w. \tag{5}$$

Volumetric heat capacity for moist ground is calculated as a weighted average of the dry ground and water heat capacities (subroutine `calcHcapHcond()` in `BalanceModel.f90`):

$$(\rho c)_g = (1 - \phi)(\rho c)_s + \phi(\rho c)_w, \tag{6}$$

where

$(\rho c)_g$ = volumetric heat capacity of wet ground ,
$(\rho c)_s$ = volumetric heat capacity of dry ground ,
$(\rho c)_w$ = volumetric heat capacity of water ,
ϕ = porosity of the ground .

Porosity ϕ gives the fraction of void space within the ground. Typical values for ground (g) and asphalt ($asph$) are :

ϕ_g = 0.4 ,
ϕ_{asph} = 0.1 .

Heat conductivity of the ground is considered to be temperature dependent according to equation ([3], p. 32) (subroutine `calcCC()` in `BalanceModel.f90`):

$$\lambda = A + B\theta - (A - D)e^{-(C\theta)^E}, \tag{7}$$

where

θ = volumetric water content .

Coefficients A-E depend on the type of the ground. E.g., for deep ground, the following values are used in the model ([3], p. 32) (subroutine `HCapValues()` in `BalanceModel.f90`):

$$
\begin{aligned}
A &= 0.65 - 0.78\rho + 0.60\rho^2 && (8)\\
B &= 1.06\rho && (9)\\
C &= 1 + (2.6/\sqrt(m_c)) && (10)\\
D &= 0.03 + 0.1\rho^2 && (11)\\
E &= 4\ , && (12)
\end{aligned}
$$

where

m_c = clay fraction of the ground .

To be later used in the calculation of available heat for melting, volumetric heat capacity (J/m^3K) is converted to intensity units (W/m^2K) by multiplying it with layer thickness and dividing it with time step length:

$$HS_i = (\rho c)_g \frac{Z_{i+1} - Z_{i-1}}{2\Delta t} \quad (13)$$

4.2 Temperature field and layer height initialization

The vertical temperature distribution in the ground is calculated in the model by dividing it vertically into 15 layers. The uppermost layers are thinner than the ones in the bottom. Layer thickness is growing according to formula (`initDepth` in `Initialization.f90`)

$$
\begin{aligned}
Z_{i+1} &= Z_i + 0.01442(i-1) + Z_{Add} \quad , \quad i = 2, 3, \ldots, 15, && (14)\\
Z(1) &= 0
\end{aligned}
$$

where

Z_{i+1} = distance of layer i midpoint from ground surface ,
Z_{Add} = constant .

If there are no road surface temperature (RST) observations, the temperature of the uppermost layer and layers 2-4 is set to be same as the air temperature at the first time step.The temperature of the lowest layer is calculated with equation [7] (`initTemp` in `Initialization.f90`). :

$$T_{m+1} = Tc_{m+1} + Asin(\Omega J + \Omega Di - \frac{Z_{m+1}}{d}) \quad (15)$$

where A is the amplitude of variation during the year, Ω is frequency of the variation ($2\pi/365$), J is the Julian day, Tc is the climatological temperature average, Di is the displacement of the curve, m is the number of the ground layers and d is damping depth. In the model $Tc = 6.4$, $A = 0.6$, $Di = -170$ and $d = 2.7$.

Temperatures between the fourth layer and the lowest layer are calculated so that the change is linearly dependent of depth (`initTemp()` in `Initialization.f90`). If there are RST observations, the temperature of the layers 1-4 is set to be the observed RST but otherwise the initialization does not change.

4.3 Surface energy balance

Surface heat balance is calculated with equation [8] (subroutine `calcProfile()` in `BalanceModel.f90`):

$$G_0 = R_n - LE + H + Tr, \quad (16)$$

where

G = Heat flux to the ground,
R_n = Net radiation ,
LE = Latent heat flux ,
H = Sensible heat flux ,
Tr = Heating caused by traffic.

Next sections describe about these terms in the model in more detail.

4.4 Net radiation

Outgoing radiation LW_{up} [W/m^2] is calculated using black body emission (subroutine `calcRNet` in `BalanceModel.f90`):

$$LW_{up} = \epsilon\sigma_{SB}T^4_{surf}, \quad (17)$$

where

ϵ = emissivity factor ,
σ_{SB} = Stefan-Boltzmann constant $[5.67 * 10^8 W/(m^2K^4)]$,
T_{surf} = surface temperature $[K]$.

ϵ is given as input value to the model. Net radiation $[W/m^2]$ is calculated as [8]: (subroutine `calcRNet` in `BalanceModel.f90`):

$$R_n = SW_{down}(1 - \alpha_s) + LW_{down} - LW_{up}, \tag{18}$$

where
SW_{down} = incoming short wave radiation $[W/m^2]$,
α_s = surface albedo,
LW_{down}= incoming long wave radiation $[W/m^2]$

If sky view factor (SVF) and local horizon angles (LHA) are given, they are used to modify radiation fluxes [9] (subroutine `ModRadiationBySurroundings()` in `modRadiation.f90`). Incoming short wave radiation is calculated as the sum of diffuse solar radiation (SW_{dif}) and direct solar radiation (SW_{dir}). The model takes total incoming short wave radiation and direct solar radiation by input from the NWP model. The diffuse solar radiation is calculated as:

$$SW_{dif} = SW_{down} - SW_{dir} \tag{19}$$

Then the diffuse solar radiation is modified by using sky view factor [10]:

$$SW_{dif} = SVF * SW_{dif} + (1 - SVF)SW_{ref} \tag{20}$$

where SW_{ref} is reflected solar radiation from surroundings and is calculated as sum of reflected diffuse and direct radiation fluxes:

$$SW_{ref} = \alpha_{sur}SW_{dir} + \alpha_{sur}SW_{dif} \tag{21}$$

where α_{sur} is albedo of the surrounding environment.

Long wave radiation from surroundings is determined as the difference of net long wave radiation and incoming long wave radiation given as input to the model:

$$LW_{sur} = LW_{net} - LW_{down} \tag{22}$$

The incoming long wave radiation is modified by using following equation [10]:

$$LW_{down} = SVF * LW_{down} - (1 - SVF)LW_{sur} \tag{23}$$

RoadSurf uses given local horizon angels to determine whether the road point is in shadow. First, sun elevation and azimuth angles are calculated with `SunPosition` subroutine (`SunPosition.f90`). The calculation is based on book by Jean Meeus [11]. Then, the model checks whether the local horizon angle in the direction of the sun is greater than sun elevation angle. If that is the case, the direct solar radiation is set to zero.

4.5 Albedo

Surface albedo describes how much the surface reflects radiation. It is given to the RoadSurf as input. With dry surface, it has default value of 0.1. However, the albedo changes when the surface is covered by snow, ice or deposit. If there is snow on the surface and the snow storage is bigger than ice storage, albedo is set to the value for snow. Snow albedo is also given as input to RoadSurf, by default it is 0.6. Otherwise, if there is ice or deposit on the surface, albedo depends linearly of the total ice content. Total ice content is calculated as an average of the ice storage and the secondary ice storage added to the deposit storage. If combined ice storage is more than 1.5 mm, the albedo is set to be snow albedo. If combined ice storage is below 1.5 mm, then albedo is calculated with equation (subroutine `calcAlbedo` in `Cond.f90`):

$$\alpha_s = \alpha_{dry} + \frac{St_{sum,ice}/1.5mm}{\alpha_{snow} - \alpha_{dry}}, \tag{24}$$

where α_{dry} is dry surface albedo, α_{snow} is snow albedo and $St_{sum,ice}$ is the total ice content.

4.6 Sensible heat flux

To calculate sensible heat flux (H) we need to know the boundary layer conductance (BLC). This can be calculated using equation [3] [4](subroutine `calcBLCondAndLE()` in `BoundaryLayer.f90`):

$$BLC = \frac{c_a \rho_a k u^*}{ln(\frac{z_T - d + z_h}{z_h}) + \Psi_h}, \tag{25}$$

where

c_a = specific heat of air $[J/(kgK))]$,
ρ_a = density of air $[kg/m^3]$,
k = von Karman's constant (0.4) ,
u^* = friction velocity$[m/s]$,
z_T = height above the surface at which temperature is measured $[m]$,
d = zero plane displacement$[m]$,
z_h = surface roughness parameter for heat $[m]$,
Ψ_h = stability correction factor for heat .

H can then be calculated as:

$$H = BLC(T_{surf} - T_a), \tag{26}$$

where T_a is air temperature.

c_a is calculated as isobaric specific heat for dry air [12]:

$$c_a = 1005 + \frac{(T_a - 250)^2}{3364} \tag{27}$$

Air density can be solved from ideal gas law:

$$\rho_a = \frac{p}{R_d T_a}, \tag{28}$$

where p is pressure [Pa] and R_d is gas constant for dry air (287.05 J/(KgK)). Pressure is assumed to be 1000 hPa. Values for z_T, d and z_h can be given to the model as input data. Default values are $z_T = 2\ m$, $d = 0\ m$ and $z_h = 0.001\ m$. u^* is calculated from equation [3]:

$$u^* = \frac{ku}{ln(\frac{z_W - d + z_m}{z_m}) + \Psi_m}, \tag{29}$$

where z_W is wind speed measurement height, z_m is surface roughness parameter for momentum and Ψ_m is stability correction factor for momentum.

Ψ_m and Ψ_h are used to correct the boundary layer conductance for the buoyancy effects. If boundary layer is stable, mixing is smaller than in unstable conditions. To describe stability we define factor which tells the relative importance of thermal and mechanical turbulence in boundary layer transport [3]. This factor can be calculated as

$$\zeta = -\frac{k z_T g H}{c_a \rho_a T_a {u^*}^3}, \tag{30}$$

where g is gravitational constant (9.81 m/s^2). Now we can write Ψ_h and Ψ_m as function of ζ [3]. In stable conditions ζ is positive and

$$\Psi_h = \Psi_m = 4.7\zeta. \tag{31}$$

In unstable conditions ζ is negative and

$$\Psi_h = -2ln(\frac{1 + \sqrt[2]{1 - 16\zeta}}{2}) \tag{32}$$

and

$$\Psi_m = 0.6\Psi_h. \tag{33}$$

Because ζ is function of H, which we are trying to solve, we must use iterative process to obtain the value. In the first iteration round we start from values $\Psi_h = 0$ and $\Psi_m = 0$. Then we calculate BLC, H and ζ. We then calculate Ψ_h and Ψ_m with equation 31 or with equations 32 and 33 depending on stability. On the next iteration round we use these values of Ψ_h and Ψ_m to calculate BLC, H and ζ and so on until the absolute difference of BLC between subsequent iteration rounds is smaller than 0.001 or the maximum number of iterations is reached. If maximum number of iterations is reached but difference is still bigger than 0.01 an error message is given. Default value for maximum iterations is 40.

4.7 Latent heat flux

Latent heat flux can be calculated as [13] (subroutine `calcLE()` in `BoundaryLayer.f90`):

$$LE = \frac{\rho_m c_a}{\gamma} \frac{e_s - e_a}{r_o} \tag{34}$$

where

ρ_m = density of moist air $[kg/m^3]$,
γ = psychrometric constant $[kPa/K]$,
e_s = vapour pressure of the surface $[kPa]$,
e_a = vapour pressure of the air $[kPa]$,
r_o = aerodynamic resistance. $[s/m]$

In the model ρ_m is approximated as ρ_a. γ is calculated using equation:

$$\gamma = 0.1 * (0.00063 T_a + 0.47496), \tag{35}$$

which has been developed using values in the book of Calder (1990) [13] for γ in different temperatures. Water vapour pressure is assumed to be same as saturated water vapour pressure at the surface. This can be calculated over water as [13]:

$$e_s = 0.61078 e^{17.269 T_{surf}/(T_{surf}+237.3)} \tag{36}$$

and over ice as

$$e_s = 0.61078 e^{21.875 T_{surf}/(T_{surf}+265.5)} \tag{37}$$

Water vapour pressure in the air is calculated from saturation water vapour pressure using relative humidity:

$$e_a = \frac{RH}{100} * e_s, \tag{38}$$

where e is saturation water vapour pressure and RH is relative humidity. e_s can be calculated using equations 36 and 37. If relative humidity is greater than 100 % it is decreased to 100 %. According to Tourula ja Heikinheimo (1998) [14] last term in the equation 34, r_o, can be calculated in unstable conditions as (subroutine `calcRAero` in `BoundaryLayer.f90`):

$$r_o = \frac{(ln(\frac{z-d}{z_m}) + \Psi)(ln(\frac{z-d}{z_h}) + \Psi)}{k^2 u}. \tag{39}$$

Here Ψ_m andΨ_h are considered to be equal and marked as Ψ. In RoadSurf Ψ_m and Ψ_h are used separately in this equation. Also zero plane displacement is assumed to be small, but surface roughness is added to the logarithm. Maximum value for aerodynami resistance is set to 30 s/m in RoadSurf. Now we have all parameters to calculate latent heat flux from equation 34.

With latent heat flux we can also calculate the amount of evaporated water in time step with equation:

$$EV_w = \frac{LE}{E_{m^3}} \partial t. \tag{40}$$

Here E_{m^3} $[J/m^3]$ means the amount of energy needed to evaporate one cubic meter of water. It can be calculated as:

$$E_{m^3} = L_{wat} \rho_w, \tag{41}$$

where L_{wat} is latent heat of water vaporization $[J/kg]$ and ρ_w is the density of water.

If temperature is below zero we can calculate E_{m^3} as

$$E_{m^3} = L_{fus} \rho_w, \tag{42}$$

where L_{fus} means the latent heat of sublimation $[J/kg]$.

If there is not water on the surface, latent heat flux and evaporated amount of water are set to zero.

4.8 Relaxation

Relaxation means that the forecasted air temperature, relative humidity and wind speed values are modified in the forecast phase according to the bias of these variables at the time of the last observations . This is done using the equation (subroutine `RelaxationOperations()` in Relaxation.f90):

$$X(t) = X_F(t) - (X_{FO}(t) - X_O)e^{-\frac{t}{t_c}}, \tag{43}$$

where $X(t)$ is the forced value as time t, X_{FO} is the forecasted value at the time of the last observation and X_O is the last observed value. t_c is same scaling constant as in equation 44. It is possible to turn relaxation on or off.

4.9 Coupling

Forecasted radiation can be corrected using so called coupling method [15] [16]. Generally one run using RoadSurf has two phases: initialization phase and the forecast phase. In the initialization phase the model is run using observations as input, and in the forecast phase the input is obtained from Numerical Weather Prediction model (NWP model). However, the radiation is not measured at road weather stations, so it is taken from the nearest forecast also in the initialization phase. When coupling is used, an additional phase is added between the initialization and the forecast phase. This phase is called coupling phase and it uses both observations and forecast. Usually it takes a few hours for NWP to produce the forecast, during which new observations are made. Consequently, there is usually period when both observations and forecast are available.

In coupling phase, the road surface temperature (RST) observations are used to correct the forecasted radiation. The correction is done by a radiation correction coefficient. When the coefficient is 1.0, the radiation has the original forecasted value. When the coefficient is smaller than 1.0, the radiation is reduced, and when it is greater than 1.0, the radiation is increased. The correction coefficient is given either to the incoming long wave or short wave radiation depending on which one has higher intensity at the end of the coupling phase. If sky view factors and local horizon angles are used, the correction coefficient is always given to long wave radiation. The determination of the size of the correcting coefficient is done using the following procedure:

1. Run model normally several hours
2. Compare the forecasted RST to the observed RST

3a. If forecasted RST is too high, reduce the correcting coefficient
3b. If forecasted RST is too low, increase the correcting coefficient

4. Go back to the start of the coupling phase and start again from 1 using new coefficient for radiation.

This procedure is repeated until the forecasted RST is within 0.1 K from the observed RST. The determination of the coefficient is explained in more detail at the end of this section. After the determination of the coefficient, the model continues to the actual forecast phase and uses the obtained coefficient for the selected radiation variable (incoming long wave or short wave radiation). However, the coefficient is forced gradually towards 1.0 as the forecast continues using the following equation:

$$C_f(t) = 1.0 + C_R e^{-\frac{t}{t_c}}, \tag{44}$$

where C_f is the forced correcting coefficient at time t, C_R is the original coefficient, and $t_c = 4h$ is a scaling constant.

Next the calculation of the correcting coefficient is explained in detail. The procedure originates from open source METRo road weather model [15], but heavy modifications have been made. First we introduce the used parameters:

k = iteration number,
C_R = Final coefficient used in the forecast,
$C_{R,k}$ = coefficient of the current iteration ,
$C_{R,k+1}$ = coefficient for the next iteration ,
ΔT_{above} = saved value of the forecasted RST from the iteration when the it has been closest to observed value from above (The forecasted value has been too high) ,
ΔT_{below} = saved value of the forecasted RST from the iteration when the it has been closest to observed value from below (The forecasted value has been too low),
C_{above} = coefficient from the iteration in which ΔT_{above} was obtained ,
C_{below} = coefficient from the iteration in which ΔT_{below} was obtained ,

The procedure to determine the coefficient is described below (subroutine `Coupling_control()` in Coupling.f90). The options are exclusive and are checked in this order.

1. If k=25, set $C_R = 1.0$ and end coupling (too much iterations)
2. If the surface temperature observation is missing or the value is abnormal at the end of the coupling period, set $C_R = 1.0$ and end the coupling
3. If surface temperature has abnormal value, et $C_R = 1.0$ and end the coupling

4. If the forecasted RST is not within 0.1 K from the observed RST and there has been an iteration where forecasted RST was too high as well as an iteration when it was too low (There exists saved values for ΔT_{above} and ΔT_{below}):

$$C_{R,k+1} = C_{above} - \frac{\Delta T_{above}}{\Delta T_{above} - \Delta T_{below}} (C_{above} - C_{below}) \tag{45}$$

5. If the forecasted RST is more than 0.1 K higher than the observed RST: $C_{R,k+1} = 0.5 C_{R,k}$
6. If the forecasted RST is more than 0.1 K below the observed RST: $C_{R,k+1} = 2C_{R,k}$
7. If the forecasted RST is within 0.1 K from the observed RST, set $C_R = C_{R,k}$

4.10 Storage terms

The RoadSurf model describes the amount of water, snow, ice and deposit on the surface with storage terms. All storage terms tell the equivalent water content in mm. Different processes can change the amount of substance in the storage. For example precipitation increases water or snow storage depending on precipitation phase. Wear due to traffic decreases all storages. Storage terms also interact with each other, for example ice can melt to water , which increase water storage and decrease ice storage. Storages are calculated for driving tracks.

4.10.1 Precipitation type

Precipitation is given to the model in units mm/h. To calculate the change of storage terms it is transferred to units $mm/\partial t$, which means millimetres in time step. Precipitation is divided to rainfall and snowfall depending on its type. Model can either interpret the type by itself or use values given from weather model. These operations are done in subroutine `CalcPrecType()` in `Cond.f90`. RoadSurf uses three precipitation types in calculation: water, snow and sleet.

If values are given from the NWP model, the RoadSurf decodes types as:

0	=	Drizzle ,
1	=	Rain ,
2	=	Sleet ,
3	=	Snow ,
4	=	Freezing drizzle ,
5	=	Freezing rain ,
6	=	Hail .

Drizzle, rain, freezing drizzle and freezing rain are classified as water. In addition to snow, hail is also classified as snow. Sleet is naturally classified sleet. If precipitation type is water, it is considered as rainfall and if type is snow, it is considered as snowfall. If precipitation type is sleet, half of precipitation is considered as snowfall and the other half as rainfall.

If neither precipitation type is not given, model interprets the type using function [17]

$$P_{rain} = \frac{1}{1 + e^{P_{exp}}}, \tag{46}$$

where $P_{exp} = 22 - 2.7T_a - 0.2RH$. T_a means air temperature and RH relative humidity. If $P_{rain} < 0.3$ precipitation type is snow. With $P_{rain} > 0.7$ precipitation type is water and if $0.3 < P_{rain} < 0.7$ precipitation type is sleet. Precipitation is divided to snowfall and rainfall in the same way as in case when precipitation type was given to the model.

4.10.2 Wear

Traffic wear is included to the model by using wear factors. Wear can decrease the amount of snow, ice, deposit and water and transform snow to ice. Wear factor tells how much storage is reduced $[mm/h]$.

The wear factors are calculated as (subroutine `WearFactors()` in `Cond.90`):

$$Wf_{snow} = 0.45 * St_{snow}, \tag{47}$$

$$Wf_{ice} = 0.319 * St_{ice}, \tag{48}$$

$$Wf_{deposit} = 1.16 * St_{deposit}, \tag{49}$$

$$Wf_{water} = 0.145 * St_{water}. \tag{50}$$

Wf_{snow}, Wf_{ice}, $Wf_{deposit}$ and Wf_{water} mean wear factors for snow, ice, deposit and water and St_{snow}, St_{ice}, $St_{deposit}$ and St_{water} mean storage terms. If snow storage is smaller than 0.2 mm, snow wear is multiplied by 3. Ice has also secondary storage term St_{ice2}, which means secondary ice storage. It wears out faster than primary ice storage and describes amount of ice on busy lanes. It's wear factor is calculated as:

$$Wf_{ice2} = 2.552 * St_{ice2}. \tag{51}$$

wear factors for snow, ice and deposit has minimum value of 0.01 mm and wear factor for water has minimum value of 0.06 mm. Deposit wear is included only in snow free conditions. If $St_{snow} > 0$, then $Wf_{deposit} = 0$.

To get wear as millimetres in one time step ($W_{storage}$), wear factors [mm/h] are divided by 3600 and multiplied by the length of the time step [s].

4.10.3 Changes in storage terms

Changes in storage terms are caused by rainfall, snowfall, wear and transfer from one storage to another. We can write following equations for changes in storage terms during one time step (subroutines `WaterStorage()`, `SnowStorage()`, `IceStorage()`, `DepositStorage()` in `Stroage.f90`):

$$\begin{aligned}\Delta St_{water} = RF - Ev_{water} + Me_{snow1} + Me_{snow2} + Me_{ice} + Me_{deposit} \\ -Fe_{water1} - Fe_{water2} - \omega W_{water} + St_{deposit,over},\end{aligned} \tag{52}$$

$$\Delta St_{snow} = SF - Me_{snow1} - Me_{snow2} - W_{snow} - STI, \tag{53}$$

$$\Delta St_{ice} = Fe_{water1} + Fe_{water2} - Me_{ice} - W_{ice} + \beta W_{snow} + STI + DTI, \tag{54}$$

$$\Delta St_{ice2} = Fe_{water1} + Fe_{water2} - Me_{ice} - W_{ice2} + \beta W_{snow} + STI, \tag{55}$$

$$\Delta St_{deposit} = -Ev_{deposit} - Me_{deposit} - W_{deposit} - DTI, \tag{56}$$

where

ΔSt_{water}	=	Change in water storage in one timestep ,
ΔSt_{snow}	=	Change in snow storage in one timestep ,
ΔSt_{ice}	=	Change in ice storage in one timestep ,
ΔSt_{ice2}	=	Change in secondary ice storage in one timestep ,
$\Delta St_{deposit}$	=	Change in deposit storage in one timestep ,
RF	=	Rainfall ,
SF	=	Snowfall ,
Ev_{water}	=	Evaporation/ condensation of water ,
$Ev_{deposit}$	=	Evaporation/ condensation of deposit ,
Me_{snow1}	=	Normal melting of snow ,
Me_{snow2}	=	Melting of wet snow ,
Me_{ice}	=	Melting of ice ,
$Me_{deposit}$	=	Melting of deposit ,
Fe_{water1}	=	Normal freezing of water ,
Fe_{water2}	=	Freezing of water in case of wet snow ,
W_{snow}	=	Snow wear ,
W_{ice}	=	Ice wear,
W_{ice2}	=	Secondary ice storage wear ,
$W_{deposit}$	=	Deposit wear ,
W_{water}	=	Water wear ,
β	=	Snow packing to ice factor ,
ω	=	Water wear parameter ,
$St_{deposit,over}$	=	Overflowing deposit ,
STI	=	Snow transfer to ice ,
DTI	=	Deposit transfer to ice .

All units are mm. The amount of rainfall and snowfall was handled in section 4.10.1 and wear in section 4.10.2.

For Ev_{water} to be other than zero surface needs to fulfil certain conditions. Ice, snow and deposit storages must be zero and and the average surface temperature ($T_{surf,av}$) must be over 0.25 °C. If these conditions do not occur, $Ev_{water} = 0$. Average surface temperature denotes the average temperature of the two uppermost ground layers.

The amount of evaporated/condensated water in time step (EV_w) was already calculated in section 4.7. However, we can use this only if there is water on the surface. If the water is only in the pores of the road, we need to multiply EV_w by pore resistance factor for evaporation. The water is considered to be only in pores if water storage is smaller than 1 mm. If water storage is greater, 1 mm of water is considered to be in pores ($St_{water,pore}$) and the rest on the surface ($St_{water,surf}$). However, in the current version of the model pore resistance factor does not make difference, because of the value of the factor in the model is 1. So in the end $Ev_{water} = EV_w$.

The RoadSurf does not take evaporation from icy surface into account, so $Ev_{deposit}$ can get only negative values and mean condensation to the surface. If EV_W is negative, $Ev_{deposit} = EV_w$, otherwise $Ev_{deposit} = 0$.

To calculate the melting terms we need to know the amount of available energy. This is the heat which is released when the surface cools to melting temperature. When melting occurs, surface temperature is set to melting temperature in the model. To calculate available energy we use HS_1 which was calculated in section 4.1.1. Here we use power units for calculation purposes. We get the available energy in units $[W/m^2]$ just by multiplying HS_1 with the change in temperature (subroutine `melting()` in `Storage.f90`):

$$Q_{av} = HS_1(T_{surf} - T_{melt}), \tag{57}$$

where T_{melt} =0.25 °C is the melting temperature used in the model. Basically Q_{av} means the power with which surface could cool to the melting temperature in one time step. The power used to melt the snow or ice can't be greater than this power. If available power isn't enough to melt the full snow layer in one, surface remains partly snow covered. The same is true for ice, but here we consider only snow for simplicity. If the available power is more than needed to melt the full layer, the rest of the energy is used to warm up the surface from the melting temperature. New surface temperature can be solved from equation 57 when putting the remaining power (Q_{re}) in place of Q_{av}. The result is:

$$T_{surf} = T_{melt} + \frac{Q_{re}}{HS_1}. \tag{58}$$

The power needed to melt the full snow layer, in other words the snow storage, in area A in one time step can be calculated as (subroutine `NewMeltFreezeHeat()` in `Storage.f90`):

$$Q_{all} = \frac{L_{melt}M_{snow}}{\partial tA} = \frac{L_{melt}V_{snow}\rho_w}{\partial tA} = \frac{L_{melt}Ah_{snow}\rho_{snow}}{\partial tA} = \frac{L_{melt}\rho_w\frac{St_{snow}}{1000}}{\partial t}, \tag{59}$$

where

L_{melt}	=	Latent heat of melting $[J/kg]$,
M_{snow}	=	Mass of snow $[kg]$,
V_{snow}	=	Volume of snow (changed to water) $[m^3]$,
h_{snow}	=	Height of snow layer $[m]$.

Melting can occur if average surface temperature is over melting temperature and $Q_{av} > 0$. If $Q_{av} < Q_{all}$, power used in melting (Q_{melt}) is just Q_{av}. If $Q_{av} > Q_{all}$, power used in melting is Q_{all}. The amount of melted snow in millimetres can be solved from equation 59 when putting Me_{snow1} in place of St_{snow} and Q_{melt} in place of Q_{all}. The result is (subroutines `SnowStorage()` and `WaterStorage` in `Storage.f90`):

$$Me_{snow1} = 1000\frac{Q_{melt}\partial t}{L_{melt}\rho_w}. \tag{60}$$

The same procedure can be done when calculating the melting of ice.

There is also an other condition in which snow can melt in RoadSurf. If snow is wet enough, it is automatically transferred to water, $Me_{snow2} = St_{snow}$. This happens when water to snow ratio is bigger than 0.6. Water to snow ratio is calculated as:

$$WS_{rat} = St_{water,surf}/(St_{water,surf} + St_{snow}), \tag{61}$$

where $St_{water,surf}$ is water on the surface.

Melting of deposit is calculated in simpler way. If the average surface temperature is above the melting temperature, all deposit is automatically melted, $Me_{deposit} = St_{deposit}$

Freezing of water (Fe) is also simplified in the model. If average surface temperature is below −0.25 °C, the whole water storage freezes: $Fe = St_{water}$.

The change from snow to ice can occur when snow is considered wet. To determine this model uses also water to snow ratio, but the limit is smaller than when calculating melting. If snow is wet and average surface temperature is below freezing temperature the whole snow storage transforms to ice storage, $STI = St_{snow}$. Snow transforms to ice also due to wear of traffic. This is taken into account in term βW_{snow} in equation 54.

The value of water reduction due to traffic wear (ωW_{water}) depends on the amount of water in the road. If St_{water} is smaller than 0.1 mm, there is no reduction due to traffic wear ($\omega = 0$). If St_{water} is greater than 0.9 mm, reduction is equal to W_{water} ($\omega = 1$). If St_{water} is between 0.1 and 0.9, reduction is halved ($\omega = 0.5$)

If deposit and snow occur at the same time, model transfers deposit to ice storage. Then $DTI = St_{deposit}$.

Although changes in storage terms are represented here as equations which consists of many terms, in reality model calculates the changes one at a time. Model takes care that storage terms will not get negative by checking values. There are also maximum limits for each storage. Maximum limit for water storage is 2 mm, for snow storage 100 mm, for ice storage 50 mm and for deposit storage 2 mm. If storage terms are exceed these values, they are set to maximum values. Snow storage is an exception, because as a crude approximation of ploughing it is reduced to half if it goes over maximum limit. In case of snow, ice and water, the overflowing part is thought to be blown off the road and just disappears, but overflowing deposit storage ($St_{deposit,over}$)is transferred to water storage.

There are also minimum limits for storage terms. Value of storage term is set to zero if it goes below this value. Minimum limit for water storage is 0.05 mm, for snow storage 0.1 mm,for ice storage 0.05 mm and for deposit storage 0.01 mm. Minimum and maximum limits prevents the model going unrealistic.

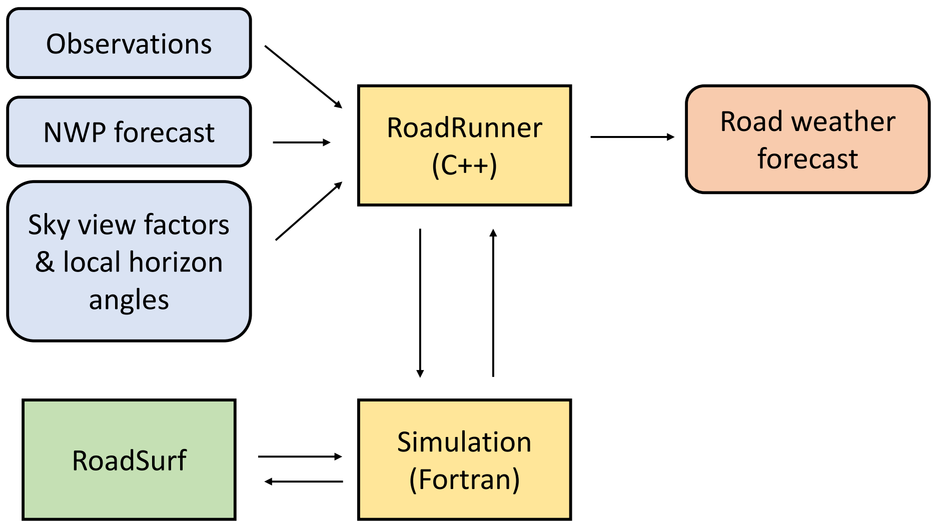

Figure 1: Data flow in the example implementation

5 Example implementation

5.1 Running

Folder `example/example1` includes input files `example_observations.json` and `example_forecast.json`. `example_config.json` is a configuration file containing information for the model run, like list of input data, initialization length, forecast length, model time step and other settings. The folder has also a Makefile for automatic compilation. The code is located in folder `src`. The code should be compiled with command `make`. After that, the example simulation can be run with command:

```
./roadrunner -t 20191202T0000 example_config.json
```

where the forecast start time is given by option `-t`. If the RoasSurf library is not found, you might need to set the library path first:

```
LD_LIBRARY_PATH="insert RoadSurf folder"
export LD_LIBRARY_PATH
```

Note that `-t` does not set the simulation start time, but the forecast start time. The simulation start time is forecast start time reduced by the initialization length. Current time is used is forecast start time is not specified. Other available options are:

- `-h` print help message
- `-j` number of parallel processes
- `-c` configuration file

The configuration file is read by default as the last argument. To run the model with for example eight parallel processes, use command

```
./roadrunner -t 20191202T0000 -j 8 example_config.json
```

The output printed to screen should look like this:

```
Reading data source named 'MEPS_000'
Reading data source named 'RWS_obs'
Number of jobs = 8
 coupling coefficient too small, coupling failed
```

`coupling coefficient too small, coupling failed` indicates that coupling failed at one simulation point. This is normal and is not a cause of concern. Coupling method tries to modify radiation so that the simulated surface temperature fits to the last observed surface temperature. If the coefficient modifying radiation becomes too large or small the coupling is terminated.

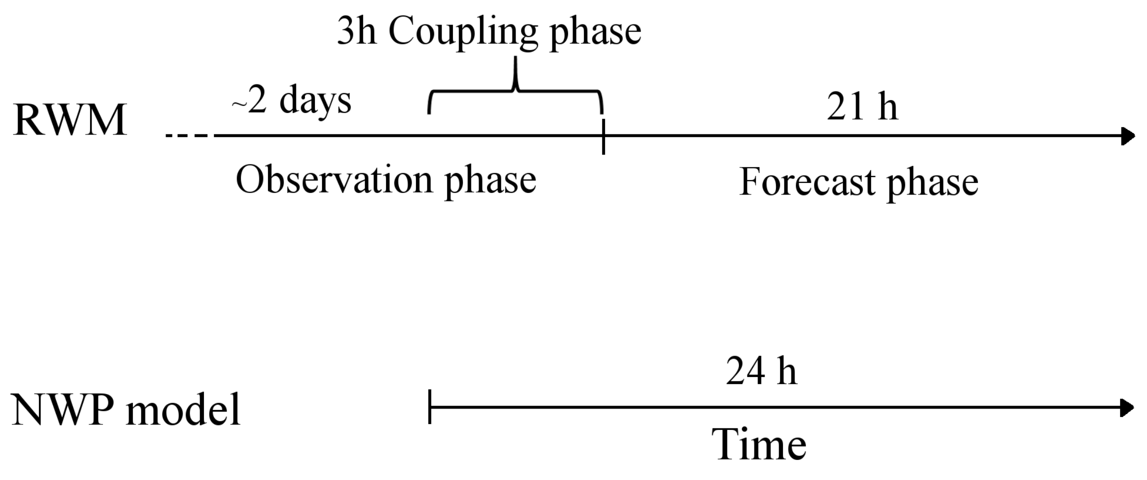

Figure 2: Phases in the road weather model and the relation of the model time line to a forecast from 3D numerical weather prediction (NWP) model. The duration of the model phases and the NWP forecast length may vary.

The simulation produces output file `example_output.json`

5.2 Data flow

The data flow in the example implementation is defined in Figure 1. C++ part of the program reads in observed and forecasted atmospheric values along with sky view factors and local horizon angles. Then it calls Fortran subroutine to run the simulation for each point. Fortran part uses subroutines in RoadSurf library to run the simulation. After simulations for each point are done, the C++ part writes the output file.

5.3 Input

5.3.1 Required variables

Running the model simulation requires following variables as input:

- Air temperature (°C)
- Dew point temperature (°C) or Humidity (%)
- Wind speed (m/s)
- Precipitation (mm/h)
- Incoming long wave radiation (W/m^2)
- Incoming short wave radiation (W/m^2)

In addition, Road surface temperature (°C) observations should be used in the model initialization when available but the model works without them. Precipitation phase is also an optional variable.

The model run consists of two phases. First there is a initialization phase where the atmospheric values should consists of observations (Figure 2). After that, there is a forecast phase where the atmospheric values should consist of forecasted values. Usually the forecast is based on data from some numerical weather prediction model (NWP). Between initialization and forecast phases is a coupling phase, where the model sets a coefficient for incoming radiation to modify it so that the latest simulated road surface temperature fits to the observed surface temperature [16]. It is explained in more detail in section 4.9.

5.3.2 Required parameters

There are many parameters that are not defined in the RoadSurf library but are given as input. Tables 5.3.2 and 5.3.2 list the parameters in the derived type `InputParameters` and some default values.

5.3.3 Input for shadowing algorithm and sky view factor

The model can also use sky view factor and local horizon angles to modify incoming radiation. Section 4.4 gives more details about how they affect the radiation calculation. Local horizon angles should be given in an array with 360 values, where each value represents a local horizon angle in one degree-wide horizon slice. The first value should represent north, 90th value east, 180th value south and 270th value west. By using local horizon angles, model can determine when the road point is in shadow. In the example, files `example_skyview.txt` and `example_local_horizons.txt` show how sky view factor and local horizon angles can be given to the model, but all the sky view factors are 1.0 and local horizon angles are 0.0 so they do not represent real values. In addition to the sky view factor and local horizon angels, radiation modification requires following additional input variables:

- Net long wave radiation at the surface (W/m^2)

NightOn	19.0	Beginning hour of night traffic (UTC)
NightOff	4.0	Ending hour of night traffic (UTC)
CalmLimDay	1.5	Minimum wind speed day (m/s)
CalmLimNgt	0.4	Minimum wind speed night (m/s)
TrfFricNgt	5.0	Traffic induced friction night (W/m^2)
TrFfricDay	10.0	Traffic induced friction day (W/m^2)
Grav	9.81	Gravitational acceleration (m/s^2)
SB_Const	5.67e-8	Stefan-boltzman constant (W/m^2K^4)
VK_Const	0.4	Von Karman's konstant
LVap	2.452E6	Latent heat of water vaporisation (J/kg)
LFus	0.334E6	Latent heat of fusion(J/kg)
WatDens	999.87	Density of water at 0 C (kg/m^3)
SnowDens	100.0	Density of snow (kg/m^3) [6])
IceDens	920.0	Density of ice (kg/m^3) [6])
DepDens	920.0	Density of deposit (kg/m^3)
WatMHeat	333000.0	Latent heat of melting (J/kg)
PorEvaF	1.0	Pore resistance factor for evaporation
ZRefW	10.0	Wind reference height (m)
ZRefT	2.0	Wind reference height (m)
ZeroDisp	0.0	Zero displacement height (m)
ZMom	0.4000	Roughness factor for momentum (m)
ZHeat	0.0010	Roughness factor for heat (m)
Emiss	0.95	Emissivity constant of the surface
Albedo	0.10	Dry ground albedo
Albedo_surroundings	0.15	Albedo of surroundings
MaxPormms	1.0	Maximum porosity
TClimG	6.4	Climatological temperature at the bottom layer (°C)
DampDpth	2.7	Damping depth (m)
Omega	2*pi/365	constant to calculate bottom layer temperature
AZ	0.6	Constant to calculate bottom layer temperature
DampWearF	0.5	Damp surface wear factor
AlbDry	0.1	Asphalth albedo
AlbSnow	0.6	Snow albedo
vsh1	1.94e+06	Volumetric eat capacity of surface layers (dry) (J/m^3K)
vsh2	1.28e+06	Volumetric heat capacity of deep ground layers (J/m^3K)(dry)
Poro1	0.1	Porosity for surface layers
Poro2	0.4	Porosity for deep ground layers
RhoB1	2.11	Bulk density for surface layers (kg/m^3)
RhoB2	1.6	Bulk density for ground layers (kg/m^3)
Silt1	0.1	Clay fraction for surface layers
Silt2	0.8	Clay fraction for deep ground layers
freezing_limit_normal	-0.25	Freezing limit (°C)
snow_melting_limit_normal	0.25	Melting limit for snow (°C)
ice_melting_limit_normal	0.25	Melting limit for ice (°C)
frost_melting_limit_normal	1.25	Melting limit for deposit (°C)
frost_formation_limit_normal	0.25	Dew/deposit formation limit (°C)
T4Melt_normal	0.25	Normal melting limit (°C)
TLimColdH	-19.0	Higher limit for cold ground temperature (°C)
TLimColdL	-21.0	Lower limit for cold ground temperature (°C)
WetSnowFormR	0.1	Water to snow ratio for wet snow formation
WetSnowMeltR	0.6	Water to snow ratio for wet snow melting
PLimSnow	0.3	Precipitation interpretation : snow limit (mm)
PLimRain	0.7	Precipitation interpretation : rain limit (mm)
MaxSnowmms	100.0	MAX snow storage : mm (plowed away if above)
MaxDepmms	2.0	MAX deposit storage (mm)
MaxIcemms	50.0	MAX ice storage (mm)
MaxExtmms	1.0	MAX extra (surface) water content (mm)
Snow2IceFac	0.5	Snow to ice transition factor

Table 1: Parameters given as input to RoadSurf with some defalut values, part 1

MinPrecmm	$0.05 * \delta t/3600.0$	MIN precipitation
MinWatmms	$0.01 * \delta t/3600.0$	MIN water storage
MinSnowmms	$0.1 * \delta t/3600.0$	MIN snow storage
MaxWatmms	MaxPormms+ MaxExtmms	MAX water storate (mm)
WDampLim	0.1 * MaxPormms	Dry/Damp limit for water
WWetLim	0.9 * MaxPormm	Damp/Wet limit for water
WWearLim	0.1 * MaxPormms	Wear limit for water
MinDepmms	$0.01 * \delta t/3600.0$	MIN deposit storage
MinIcemms	$0.05 * \delta t/3600.0$	MIN ice storage

Table 2: Parameters given as input to RoadSurf with some defalut values, part 2. δt refers to model time step.

- Direct solar radiation (W/m^2)

The details of radiation modification are explained by Karsisto & Horttanainen, 2023 [9].

5.3.4 Input data management

The example RoadSurf implementation can read multiple input files. They are determined in the configuration file. The data files can have data for different times and for several different points. The data reading script interpolates the data individually for each station for each simulation time step. If the files contain overlapping data, the data in the latest file in the list replaces the previously read data. This example has forecast data in `example_forecast.json`. It contains the MEPS [18] forecast started at 2th December 2019 00 UTC, and a combination of 00, 06, 12, and 18 UTC made forecasts from previous three days. The data is created so that first six hours of each forecast are included in the file, expect for the last day for which the first 34 forecast hours are included. The forecast start time is 2th December 2019 00 UTC, and the data before that is for model initialization. The observation data is given in file `example_observations.json`. It contains road weather station observations from about three days before the forecast start time. The data is provided by Fintraffic. The atmospheric values read in from observation data replaces the values read from forecast data.

In the Example 1, the required input parameters are defined in `InputParameters.h`. Values can be also given in the configuration while, in which case they replace the default values. See section 5.5 for details.

5.4 Output

As output, the model gives following variables in file `example_output.json`:

- Road surface temperature (°C)
- Water layer thickness (mm)
- Snow layer thickness (water equivalent mm)
- Ice layer thickness (water equivalent mm)
- Deposit (frost) layer thickness (water equivalent mm)

5.5 Settings

Settings for the model run are given in the configuration file `example_config.json`. Forecast length and initialization (analysis) length are given under object ”time”:

```
"time":
  {
 "analysis": 48, // hours
 "forecast": 26 // hours
  },
```

One other available option is `coupling_minutes`, that determines the length of the coupling period in minutes.

Several model run related settings can be given under object ”model”. If `use_coupling` is set to 1, the model uses coupling method to fit the simulated surface temperature to the latest observation (see section 4.9 for details). If `use_relaxation` is given value of 1, relaxation is used. It means that when simulation moves from initialization phase to the forecast phase, air temperature, humidity and wind speed values are modified so that the transition is smooth. Otherwise there can be abrupt change in these values between observation and forecast phase (see section 4.8 for details). `DTsecs` means simulation time step in seconds.

```
    "model":
    {
        "use_coupling": 1, //1 if coupling is used
        "use_relaxation": 1, // 1 if relaxation is used
```

```
      "DTSecs": 30.0 // model time step in seconds
    },
```

Other available options are:

- `tsurfOutputDepth`: Depth from which the output surface temperature is interpolated. By default the model gives the average of the first two layers
- `couplingEffectReduction`: Value that determines how fast the coupling effect is reduced. Default value is 4.0*3600

Output settings are given in object "Output":

```
 "output":
    {
   "start": 0,      // minutes from now, default is zero
   "step":  60,     // minutes, default is 60
   "filename": "example_output.json"
    },
```

`start` means the output start time in minutes added to the forecast start time. Value 0 means that the output is started at the forecast start time, and for example -180 means that the output is started 3 hours before forecast start time. `step` means the output time step in minutes. The output file name is given in the `filename` variable.

Sky view factor and local horizon angle input file are given under object "parameters":

```
 "parameters":
    {
     "sky_view_file":"example_skyview.txt",
     "local_horizon_file":"example_local_horizons.txt"
    },
```

Several model parameters can also be given under this object to override the default values. The full list can be found in `InputParameters.h` and `InputParameters.cpp`.

Input data files are given as list under object "input". Each should contain keys `name`, `path`, `type`, `source` and `params`.

```
 // Data source names in the order they will be used

    "input":
    [
   {
       "name": "MEPS_000",
            "path": "example_forecast.json",
            "type": "json",
            "source": "forecast",
            "params":["Temperature 2m","Humidity","WindSpeed","PrecipitationForm",
                     "Precipitation","RadiationNetSurfaceLW","RadiationLW","RadiationGlobal",
                     "RadiationDirectSW"]
   },
   {
       "name": "RWS_obs",
            "path": "example_observations.json",
            "type": "json",
            "source": "observations",
            "params":["Temperature 2m","Humidity","WindSpeed","Precipitation"]
   }
    ] // sources
```

`name` should be unique for each input source. `path` is path to the input file. `type` defines the input file type. The example supports only json at the moment, but other input types can be easily added. `source` should be either "forecast" or "observations", depending if the input data contains forecast data or observation data. `params` contains variables available in the input data. The possible variable names are:

- "Temperature 2m"
- "Humidity"
- "DewPoint"
- "WindSpeed"
- "PrecipitationForm"

- ”Precipitation”
- ”RadiationNetSurfaceLW”
- ”RadiationLW”
- ”RadiationGlobal”
- ”RadiationDirectSW”
- ”RoadTemperature”

The order of the input data sources is important, as the previously read data is replaced with new data if there is overlapping

6 How to make your own implementation

6.1 Adding new input file types

Example 1 is programmed so that adding new input file types is easy. `GenericSource` provides generic class for data sources. `JsonSource` is an implementation of the `GenericSource`. To make a new data source, just make a new implementation of the GenericSource that contains the required functions:

- Constructor that reads in the location data and the weather data and interpolates the weather data to each model time step
- `void GetWeahter(&InputData pData, int pontID)`: function to retrieve input data for the given point
- `std::vector<LocalParameters> GetLocations()`: function to return vector of LocalParameters, a class that contains point coordinates
- `std::vector<int> GetPointIDs()`: function to return vector of pointIDs
- `int GetLatestObsIndex(int pointID)`: function to get index of latest observation value for the given point

After creating new implementation of generic source, the constructor should be added to `GenericSourceFactory.cpp` following the example of the json source:

```
if (type == "json")
    return new JsonSource(pWallClock, pStartTime, pEndTime, pSource, pConfig,SimLen,DTSecs);
```

The type should be defined in the configuration file, so the right class can be used for reading the data.

6.2 C++ part of Example 1

Example 1 can be modified for various needs in addition to adding new input data formats. This section gives step by step explanation of how C++ side of the Example 1 works, so that modifications can be made at a right place. The main program is located in `roadrunner.cpp`:

```
// ----------------------------------------------------------------------
/*!
 * \brief Main program
 */
// ----------------------------------------------------------------------
int main(int argc, char* argv[])
try  // NOLINT(cppcheck-syntaxError)
{
  if (!parse_options(argc, argv))
    return 0;

  // Read the JSON configurationi
  const Json::Value nulljson;
  auto json = read_json(options.configfile);
  // Initialize settings to default values
  InputSettings settings{json, options};
  //Class for handling input data
  DataHandler datahandler;

  // Start the data handler
  datahandler.init(settings.forecast_time, settings.start_time, settings.end_time,
                   json,settings.SimLen,settings.DTSecs);
  //run the model
  run(json, settings, datahandler);

  return 0;
}
catch (const std::exception& e)
```

```
{
  std::cerr << "Error: " << e.what() << std::endl;
  return 1;
}
```

It does the following steps:

1. Read command line input (`parse_options(argc, argv)`)
2. Read configuration file (`auto json = read_json(options.configfile);`)
3. Initialize input model settings using the options given in configuration file and command line (`InputSettings settings{json, options}`)
4. Create new instance of `DataHandler` class to do input data management (`DataHandler datahandler;`)
5. Read in input data with `datahandler` (`datahandler.init(...)`). It utilizes `GenericSourceFactory` mentioned earlier.
6. Run the simulation for all given points (`run(json, settings, datahandler)`)

`run` chooses one of two different functions to do the simulation depending whether parallel computation is used. The following is the code of the function without parallel computation:

```
// ----------------------------------------------------------------------
/*!
 * \brief Run the points one by one
 */
// ----------------------------------------------------------------------

void run_locations_sync(const Json::Value& pJson,
                               InputSettings& pSettings,
                               const DataHandler& pDataHandler)
{
  // Check that output file is given
  auto jfilename = Json::Path(".output.filename").resolve(pJson);
  if (jfilename.isNull() && options.outfile.empty())
    throw std::runtime_error("Output filename not set");
  std::string outputFileName=jfilename.asString();
  Json::Value nulljson;
  //Set parameters
  InputParameters parameters{pSettings, pJson.get("parameters", nulljson)};
  //Read sky view factors and local horizon angels
  SkyView skyview(pJson.get("parameters", nulljson));
  //Get simulation times
  const auto times = get_times(pSettings);
  //Get vector of location information
  auto locations=pDataHandler.GetLocations();
  //Get vector of point ids
  auto pointIDs=pDataHandler.GetPointIDs();
  //Variable to determine wheter input data was succesfully read
  bool success=false;
   // Number of forecast locations
  int nlocations = static_cast<int>(locations.size());
  //Json object to store output forecast
  Json::Value forecast;

  //Go trough all forecast locations
  for (int loc_index = 0; loc_index < nlocations; ++loc_index)
  {
     //Point location
     LocalParameters lParameters=locations[loc_index];
     success=false;
     //Get input data for point
     const auto input =
        read_input( pSettings, times, pDataHandler, parameters, lParameters,
                  skyview, pointIDs[loc_index], success);
     //If input data was succesfully obtained
     if (success)
     {
       //Report progress every 1000 points
       if (loc_index % 1000 == 0)
          std::cout << "\t" << loc_index << " / " << nlocations << " points" << std::endl;
        // initialize output vectors with missing values
       OutputData output(pSettings.SimLen);
        // Pass the data to Fortran via pointers
       auto outPointers = output.pointers();
       auto inPointers = input.pointers();
```

```
      //Run the road weather model
      runsimulation(
          &outPointers, &inPointers, &pSettings,
          &parameters, &lParameters);
      //Save output to json object
      save_output(output, input, lParameters,pointIDs[loc_index], pSettings,times,
              loc_index,forecast);

    }
  }
  //write output to file
  write_output(outputFileName,forecast);
}
```

The code does the following steps:

1. Check that output file name is given and read it to string
2. Initialize `InputParameters` class with default parameters and override if values are given in the configuration file (`InputParameters parameters{pSettings, pJson.get("parameters", nulljson)}`)
3. Read in sky view factors and local horizon angles (`SkyView skyview(pJson.get("parameters", nulljson));`)
4. Get time array of simulation times (`const auto times = get_times(pSettings);`)
5. Get vector of `LocalParamters` objects, which contain coordinates for each point (`auto locations=pDataHandler.GetLocations();`)
6. Get vector of point ID numbers (`auto pointIDs=pDataHandler.GetPointIDs();`)
7. Get number of forecast locations (`int nlocations = static_cast<int>(locations.size());`)
8. Initialize `forecast` object to store forecasts (`Json::Value forecast;`)
9. Go trough forecast locations:
   (a) Get location information for point (`LocalParameters lParameters=locations[loc_index];`)
   (b) Get input data for point (`const auto input = read_input(...)`)
   (c) If reading input was success:
      i. Print progress to screen every 1000 points
      ii. Initialize output data class (`outputData output(pSettings.SimLen);`)
      iii. Get pointers to arrays in input and output data classes
      iv. Run the simulation for the point (`runsimulation(...)`)
      v. Save output to `forecast` class (`save_output(...)`
10. Write output file (`write_output(outputFileName,forecast);`)

6.3 Fortran part of Example 1

The main road weather model simulation subroutine in Example 1 is in `Simulation.f90`. When adding new features, this subroutine can be easily modified to implement them without touching the RoadSurf library subroutines.

6.3.1 Simulation structure

`runsimulation` subroutine uses subroutines provided by the RoadSurf library to run the simulation:

```
!---------INITIALIZE----------------------------------------------------------
   call ConnectFortran2Carrays(inPointers,modelInput,&
           outPointers,modelOutput)

   call Initialization(modelInput, inSettings, settings, &
                       modelOutput, atm, surf, inputParam, localParam,&
                       coupling, phy, ground, condParam)
!---------START SIMULATION---------------------------------
   !Start temperature profile simulation, go trough input data values
   i = 1
   Do while (i < settings%SimLen .and. (settings%Simulation_Failed .eqv. .false.))
      Call CheckValues(modelInput, i, settings, surf,localParam)

      !Check if coupling is on
      if (settings%use_coupling) Then
         !Check if in coupling phase, save variables if at the start of the
         ! coupling phase,
        !Load saved variables if coupling is started again.
        !After coupling, calculate radiation coefficients
        call CouplingOperations1(i, coupling, surf, settings, ground, modelInput,&
                                 CondParam,localParam)

      end if

      !set current values to atm%Tair etc
      call SetCurrentValues(i, ground%Tmp, modelInput, atm, settings, surf, coupling,&
         ground)

      !If relaxation is used
      if (settings%use_relaxation) Then
        !Smooth t2m, rh and wind values when moving from initialization phase
        !to forecasting phase
        call RelaxationOperations(i, atm, settings,ground%Tmp)

      end if
      !Calculate temperature profile and storage values one timestep forward
      call roadModelOneStep(i, phy, ground, surf, atm,&
                            settings, coupling, modelInput, condParam,&
                            inputParam, localParam)

      !Save output
      call SaveOutput(modelOutput, i, surf)

      !Coupling control if at the end of the coupling period
      call CheckEndCoupling(i, settings, coupling, surf)

      i = i + 1
    end do

    !If simulation is not failed, make calculations for last value
    !and save output for last time step if index matches wanted
    !output time step
    if (settings%Simulation_Failed .eqv. .false.) Then

      !Make still calculation for i=SimLen (the last value)

      !set last input values as interpolated values
      call lastValues(modelInput, atm, settings, ground, surf)

      !Calculate temperature profile and storage values one time step forward
      call roadModelOneStep(settings%SimLen, phy, ground, surf, atm,&
                            settings, coupling, modelInput, condParam,&
                            inputParam,localParam)

      !Save output
      call SaveOutput(modelOutput, i, surf)
!--------SIMULATION END---------------------------
```

It uses subroutine `roadModelOneStep` to calculated road surface temperature and storage terms one step forward:

```
    !Determine wheter the precipitation is rain or snow and add to storage
    call PrecipitationToStorage(settings,condParam,modelInput%PrecPhase(input_idxI),&
                                atm,surf)
    !Make radiation corrections based on sky view and local horizon angles
    if (localParam%sky_view<1.0 .and. localParam%sky_view>-0.01)then
      call ModRadiationBySurroundings(modelInput,inputParam,localParam,input_idxI)
    end if
    !Calculate temperature profile one time step forward
    !Checks also for melting (can affect temperature)
    call BalanceModelOneStep(modelInput%SW(input_idxI), &
                             modelInput%LW(input_idxI), &
                             phy, ground, surf, atm, settings, coupling, modelInput,&
                             input_idxI,condParam)
   ! ************* WEAR FACTORS
    call WearFactors(condParam%Snow2IceFac, settings%Tph, surf, wearF)
    !Calculate storage terms
    call RoadCond(phy%MaxPormms, surf, atm, settings, &
                  condParam,wearF)

    ! ************* ALBEDO
    call CalcAlbedo(ground%Albedo, surf, condParam)
```

The given code has following steps:

1. Forward data given by C++ interface to Fortran arrays. (`ConnectFortran2Carrays`)
2. Initialize various model parameters (`Initialization`)
3. Begin the main simulation loop. The loop uses `int i` to proceed trough data points in input data arrays.
    - (a) Check if there is abnormal values (`CheckValues`)
    - (b) If coupling is in use, do coupling related checks and operations (`CouplingOperations1`)
    - (c) Update some variables for current time step (`SetCurrentValues`)
    - (d) If relaxation is in use, modify air temperature, humidity and wind speed in forecast phase (`RelaxationOperations`)
    - (e) Calculate temperature profile and storage terms for one time step forward (subroutine `roadModelOneStep`):
        - i. Add precipitation either to snow or water storage or partially to both depending on the precipitation type (`PrecipitationToStorage`)
        - ii. If sky view factor and local horizon angles are given, modify incoming radiation (`ModRadiationBySurroundings`)
        - iii. Calculate ground temperature profile one time step forward (`BalanceModelOneStep`)
        - iv. Determine wear factors (how fast storage terms decrease) (`WearFactors`)
        - v. Calculate storage terms for the next time step (`RoadCond`)
        - vi. Calculate surface albedo based on storage terms (`CalcAlbedo`)
    - (f) Save surface temperature and storage terms to model output arrays (`SaveOutput`)
    - (g) If coupling is in use and simulation is at the end of coupling phase, determine new radiation correction coefficients if necessary (`CheckEndCoupling`)
4. Update some variables for the last simulation step (`lastValues`)
5. Calculate temperature profile and storage terms for one time step forward (repeat step (e))
6. Save road surface temperature and storage terms from final step to output arrays (`SaveOutput`)

If one does not want to use C++ interface, one can replace step 1 with a subroutine that reads the data directly to `InputArrays` class and initializes `OutputArrays` class. In that case, values should be provided also for `InputSettings`, `InputParameters` and `LocalParameters`.

Depending on the user needs, each step can be replaced with another subroutine or new subroutines can be added in between.

6.4 Adding new input and output variables

`InputArrays` and `OutputArrays` are integrated to the RoadSurf library, so it is not recommended to modify them unless one wants to make new features to the library itself. To add new output or input variables for local implementation, one needs to pass them to the simulation separately. One option is to make a copy of `InputArrays` and `OutputArrays` with a different name and implement them in a similar way as the original ones.

References

[1] M. Kangas, M. Heikinheimo, and M. Hippi. "RoadSurf: a modelling system for predicting road weather and road surface conditions". In: *Meteorological applications* 22.3 (2015), pp. 544–553.

[2] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* McGraw-Hill, 1980, p. 197.

[3] G.S. Campbell. *Soil Physics with BASIC.* Elsevier, 1985, p. 150.

[4] G.S. Campbell. *An Introduction to Environmental Biophysics.* Springer-Verlag, 1986, p. 159.

[5] *CRC Handbook of Chemistry and Physics.* CRC Press, 1975.

[6] T.R. Oke. *Boundary Layer Climates.* Methuen, New York, 1987.

[7] J. L. Monteith. *Principles of Environmental Physics.* Edward Arnold, London, 1975.

[8] W. Brutsaert. *Evaporation into the Atmosphere.* D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984, p. 299.

[9] Virve Karsisto and Matti Horttanainen. "Sky View Factor and Screening Impacts on the Forecast Accuracy of Road Surface Temperatures in Finland". In: *Journal of Applied Meteorology and Climatology* 62.2 (2023), pp. 121–138.

[10] A.V. Senkova, L. Rontu, and H. Savijärvi. "Parametrization of orographic effects on surface radiation in HIRLAM". In: *Tellus A: Dynamic Meteorology and Oceanography* 59.3 (2007), pp. 279–291.

[11] Jean Meeus. *Astronomical algorithms.* Richmond, 1991.

[12] J.R Garratt. *The atmospheric boundary layer.* Cambridge University Press, 1992.

[13] G.S. Calder. *Evaporation in the Uplands.* John Wiley and Sons, 1990, p. 148.

[14] Tourula T. and M. Heikinheimo. “Modelling evapotranspiration from a barley field over the growing season”. In: *Agricultural and Forest Meteorology* 91 (1998), pp. 237–250.

[15] L.P. Crevier and Y. Delage. “METRo: A new model for road-condition forecasting in Canada”. In: *Journal of Applied Meteorology* 40.11 (2001), pp. 2026–2037.

[16] Virve Karsisto et al. “Improving road weather model forecasts by adjusting the radiation input”. In: *Meteorological Applications* 23.3 (2016), pp. 503–513.

[17] J. Koistinen and E Saltikoff. “Experience of customer products of accumulated snow, sleet and rain”. In: *COST75, Advanced weather radar systems, International seminar*. Locarno, 1999, pp. 397–406.

[18] Inger-Lise Frogner et al. “HarmonEPS—the HARMONIE ensemble prediction system”. In: *Weather and Forecasting* 34.6 (2019), pp. 1909–1937.