Contents

1 Introduction

The SVE-R model couples the 2D Saint Venant Equations (SVE), also known as the shallow water equations, to a 1D Richards equation solver for infiltration. The SVE solver is adapted from the model presented in *Bradford and Katopodes* (1999), henceforth BK1999, which uses a finite volume discretization to solve the 2D SVE on an arbitrary mesh consisting of quadrilaterals. The SVE solver uses Hancock's predictor-corrector method to achieve 2nd order accuracy in time, and Roe's Approximate Riemann Solver along with a MUSCL reconstruction of primitive variables to obtain 2nd order accuracy in space. The model can handle a variably sloping bed, as well as wetting and drying in the domain.

The principal difference between BK1999's model and the SVE-R model documented here is that infiltration is represented by solving Richards equation in 1D at each grid cell and timestep. The 1D Richards equation was implemented using the algorithm described by *Celia et al.* (1990).

The core of the model is implemented in Fortran (`dryR.for`), and Python scripts are provided to write and read the Fortran files.

This document is divided into the following sections:

- Section 2: A brief summary of the Saint Venant and Richards Equations.
- Section 3: An overview the SVE-R numerical methods, and how the two model components are coupled.
- Section 4: A summary of the simulation domain (i.e. the specific scenario of rain driven overland flow on patchily vegetated hillslopes which the model is configured to solve).
- Section 5: An overview of the Python wrapper scripts, which write the Fortran input files, compile and execute the Fortran code, and read and visualize the Fortran outputs.

DISCLAIMER: The model is published with the permission of Bradford and Katopodes, and the documentation closely follows BK1999 and *Bradford and Katopodes* (2001), henceforth BK2001. For further information and detail, please refer to these papers. Parts of Sections 2 and 3 are reproduced verbatim from BK1999 and BK2001, because their wording is are clearer and more precise than ours would be. To be clear, we are not claiming that the SVE solver is original work. Also note that the original BK1999 model is not available open source.

2 Background: equations and numerical methods

2.1 The 2D Saint Venant Equations (SVE)

The SVE are written in integral form:

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{U}d\Omega+\oint_{\partial}(\mathbf{F}dy-\mathbf{G}dx)=\int_{\Omega}\mathbf{Q}d\Omega \tag{1}$$

ann where $\mathbf{U}^T=(h,hU,hV)$ is the vector of conservative variables.

$$\mathbf{F} = \begin{bmatrix} hU \\ hU^2 + \frac{1}{2}gh^2 \\ hUV \end{bmatrix}; \quad \mathbf{G} = \begin{bmatrix} hV \\ hUV \\ hV^2 + \frac{1}{2}gh^2 \end{bmatrix}$$

where h is the flow depth, z is the bed elevation, U and V are the vertically averaged velocities in the x and y directions, respectively. The source terms are defined as:

$$\mathbf{Q} = \begin{bmatrix} p - i \\ -gh\dfrac{\partial z}{\partial x} - ghS_{f,x} + \dfrac{u(p-i)}{2} \\ -gh\dfrac{\partial z}{\partial y} - ghS_{f,y} + \dfrac{v(p-i)}{2} \end{bmatrix}$$

where p is the rainfall rate, i is the infiltration rate of water into the bed, and $S_{f,x}$ and $S_{f,y}$ are the x and y components of the friction slope.

2.2 SVE numerical methods

The model is adapted from *Bradford and Katopodes* (1999), referred to here as BK1999, which uses predictor-corrector time-stepping to provide a second-order accurate solution.

A predictor is computed at time level $n + 1/2$ by solving the primitive equations in generalized coordinates, and a corrector is computed at the $n + 1$ time level by solving the integral equations.

Predictor Step

The predictor solution is computed by solving the equations in primitive form. In generalized coordinates:

$$\frac{\partial \mathbf{W}}{\partial t} + \mathbf{A}_W \frac{\partial \mathbf{W}}{\partial \xi} + \mathbf{B}_W \frac{\partial \mathbf{W}}{\partial \eta} = \mathbf{Q}_W \tag{2}$$

where ξ and η are in the directions of increasing j and k indices, respectively. The j,k indices indicate the column and row numbers of a given cell, respectively (see Figure 1 schematic). $\mathbf{W_T} = [h, U, V]$ is the array of primitive variables, and the matrices $\mathbf{A}_W$ and $\mathbf{B}_W$ are defined as:

$$\mathbf{A}_w = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} \quad \mathbf{B}_w = \begin{bmatrix} U_\eta & h\eta_x & h\eta_y \\ g\eta_x & U_\eta & 0 \\ g\eta_y & 0 & U_\eta \end{bmatrix}$$

where $U_\xi = U\xi_x + V\xi_y$ and $U_\eta = U\eta_x + V\eta_y$. ξ_x, ξ_y, η_x and η_y are the grid transformation metrics for mapping x and y to ξ and η. In Cartesian coordinates, the generalized coordinates simplify to: $\xi = x$ and $\eta = y$.

The predictor solution in cell j, k at $t + \Delta t/2$ is given as:

$$\mathbf{W}_{j,k}^{n+1/2} = \mathbf{W}_{j,k}^{n} - \frac{\Delta t}{2}(\mathbf{A}_W \overline{\Delta \mathbf{W}}_\xi + \mathbf{B}_W \overline{\Delta \mathbf{W}}_\eta - \mathbf{Q}_W)_{j,k}^{n+1/2} \tag{3}$$

where the overbar denotes a cell-average gradient of $\mathbf{W}$ in cell j, k, which is computed with a flux limiter (nonlinear average) in order to preserve solution monotonicity. Flux limiters become first-order accurate near discontinuities while remaining second-order accurate elsewhere, and several options are included in the code, described in Section 3.

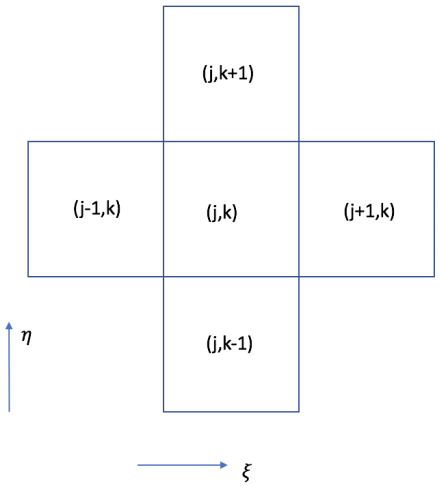


Figure 1: Sketch of a computational cell, after BK Figure 2.

Corrector Step

The corrector solution is obtained from the conservative form of the governing equations.

The predictor solutions are reconstructed to the left and right of each cell face using the monotone upstream scheme for conservation laws (MUSCL), which achieves second-order spatial accuracy. The reconstructed predictor values define a Reimann problem at each cell face, which are used to compute the interfacial fluxes:

$$\frac{\mathbf{U}_{j,k}^{n+1} - \mathbf{U}_{j,k}^{n}}{\Delta t} + \frac{1}{\Omega_{j,k}}\big[- \mathbf{F}_{\perp 1}^{n+1/2}\Delta s_1 + \mathbf{F}_{\perp 2}^{n+1/2}\Delta s_2 + \mathbf{F}_{\perp 3}^{n+1/2}\Delta s_3 - \mathbf{F}_{\perp 4}^{n+1/2}\Delta s_4\big] = \mathbf{Q}^{n+1/2}$$

where $\mathbf{Q}$ and $\mathbf{U}$ are the cell-center values in cell j, k with area Ω, $\mathbf{F}$ and $\mathbf{G}$ are the average boundary values on each cell face, and δs is the length of the cell faces. The indices 1 through 4 denote the four cell faces: index 1 corresponds to the bottom cell face, and the remaining cell faces are numbered in counter-clockwise order.

The flux **F** is normal to the cell boundary and positive in the direction of increasing cell coordinates. **F** is defined as:

$$\mathbf{F}_{\perp} = \begin{bmatrix} hu_{\perp} \\ huu_{\perp} + \frac{1}{2}gh^2 \cos\phi \\ hvu_{\perp} + \frac{1}{2}gh^2 \sin\phi \end{bmatrix}$$

where $u_{\perp}$ is the velocity perpendicular to the cell face, and ϕ is the angle between the face normal vector and the x-axis.

The fluxes are evaluated using a Godunov-type upwind scheme in which a Riemann problem is solved across each cell face, using the method of Roe (1981). Further details can be found in BK.

The source term

The source term **Q** contains the parameterization of the surface roughness (via the friction slope S_f and the lateral inputs (rainfall p and infiltration i). Infiltration is independently modeled at each timestep and grid cell with the 1D Richards equation, described in the next subsection.

The friction slope is specified in the generalized form:

$$S_{f,x} = \left(\frac{\alpha U}{h^m}\right)^{1/\eta} \frac{|U|}{U}; \quad S_{f,y} = \left(\frac{\alpha V}{h^m}\right)^{1/\eta} \frac{|U|}{V}$$

where α is a roughness parameter, and m specifies the flow regime ($m = 2$ for laminar flow, 1/2 for turbulent flow). η=1/2 for most roughness schemes, with the exception of laminar flow, for which η=1. For Manning's equation, $\alpha = n$, $m = 2/3$ and $\eta = 1/2$:

$$S_{f,x} = \frac{n^2 U}{h^{4/3}}|U|; \quad S_{f,y} = \frac{n^2 V}{h^{4/3}}|U|$$

More generally, with $\eta = 1/2$:

$$S_{f,x} = \frac{\alpha^2}{h^m}U|U|; \quad S_{f,y} = \frac{\alpha^2}{h^m}V|U|$$

2.3 Richards equation

Richards equation is solved following the approach outlined by *Celia et al.* (1990), which involves a backward Euler approximation in time coupled with a simple Picard iteration scheme. The solver used the discrete approximation of the mixed H-θ form:

$$\frac{\partial \theta}{\partial t} - \nabla \cdot K \nabla H - \frac{\partial K}{\partial z} = 0 \tag{4}$$

where z denotes the vertical dimension (assumed positive upwards), θ is the soil moisture content, H is the matric potential and K is the unsaturated hydraulic conductivity. θ and matric potential H are related via the Van Genuchten water retention curve. The volumetric soil moisture content, θ, and effective saturation, S_e, are computed as:

$$\theta = \frac{\theta_S – \theta_R}{1 + (\alpha|H|)^n)^m} + \theta_R$$

$$S_e = \frac{\theta – \theta_R}{\theta_S – \theta_R}$$

where θ_S and θ_R are the saturated and residual soil moisture content; n is a measure of the pore size distribution; $m = (1 - 1/n)$; and α is related to the inverse of the air entry suction. The unsaturated hydraulic conductivity $K(\theta)$ is computed as:

$$K = K_s \sqrt{S_e}[1 – (1 – S_e^{1/m})^m]^2 \tag{5}$$

where K_s is the saturated hydraulic conductivity. The infiltration rate is solved with Darcy's law:

$$q = -K\left(\frac{\partial H}{\partial z} + 1\right) \tag{6}$$

where the 1 (second term) on the RHS of Equation 6 reflects the fact that H is the matric head, as opposed to the hydraulic head.

Coupling the SVE and Richards Equation models

The model components are coupled at each grid cell in two steps: the depth from the SVE solver provides the surface boundary condition to the Richards equation solver, and the infiltration rate from the Richards equation solver is used by the SVE source term. This requires that several cases be accounted for: (1) no rain and no ponding, (2) rain but no ponding, and (3) ponding (with or without rain). In case (1), a no flux boundary condition is applied at the surface. In case (2), the Richards equation solver computes a potential infiltration rate (PI), defined as the infiltration rate that would occur with $H = 0$ cm at the surface, and compares this value to the rainfall intensity, p. If p exceeds the potential infiltration rate, ponding begins and the boundary condition switches to case (3). Otherwise, the potential infiltration rate is greater than p, and $i = p$. Finally, in case (3), the upper boundary condition H is equal to the ponding depth h. These cases are schematically illustrated in Figure 2.

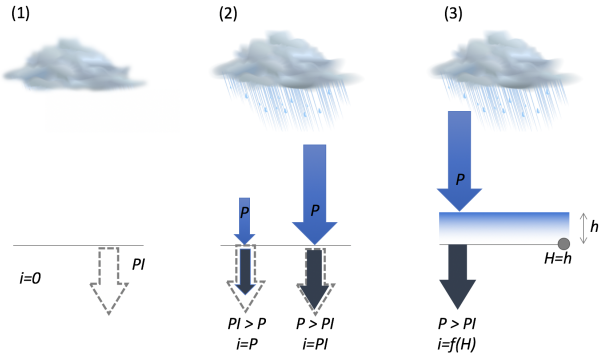


Figure 2: Schematic to illustrate the boundary condition cases for Richards equation. PI is the potential infiltration rate, which is computed to determine whether the rain intensity exceeds the infiltration capacity of the soil (in which case ponding occurs).

3 SVE-R Fortran code

This section outlines the structure of the Fortran code, how to compile and execute it, and the required input files. Further details on the the SVE and Richards equation solvers, particularly those relating the mathematical equations in Section 2 to the code in the Fortran subroutines, can be found in Appendix A.

3.1 SVE-R model structure

`dry.f` can be compiled and executed from an Mac Os terminal with:

```
gFortran -o ./sw -framework accelerate ./dry.for
.sw
```

where `.sw` is the name of the executable. `dry.f` interacts with a number of auxiliary files (see screenshot in Figure 3), including:

- `dry.inc`: specifies common variables used by `dry.f` (a number of variables are "common" variables, and not explicitly returned by the Fortran subroutines).
- `input`: files to initialize `dry.f` are located in this folder.
- `output`: `dry.f` saves the outputs to this folder.
- `params.json`: user-supplied parameter dictionary
- `.sw`: the compiled Fortran executable.
- `.ipynb` files: Jupyter notebooks to visualize the simulation outputs.

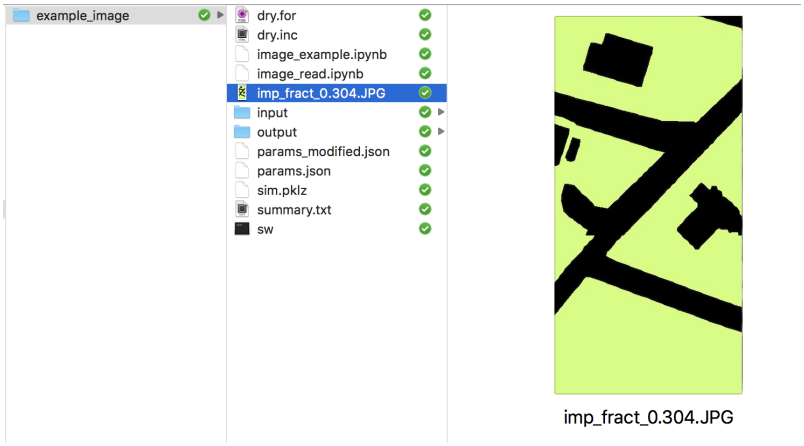


Figure 3: Auxiliary files associated with `dry.f`.

Figure 4 summarizes the organization of `dry.f`, which can be divided into main two steps: (1) initializing and (2) executing a time loop. The following subsections describe the model components in greater detail.

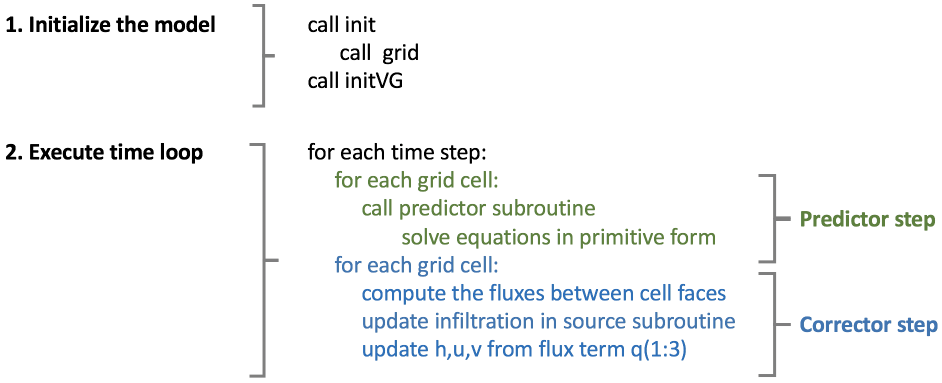


Figure 4: Summary of the SVE-R model structure in `dry.f`.

3.2 Model set-up

The `init` subroutine initializes the SVE component of the model, which involves reading a number of scalar parameters from `params.dat` and setting up the simulation grid. The infiltration parameters are initialized by the subroutine `initVG`. The following briefly describes the input files, and further details are included in Appendix B (including an illustrative example of a 2×2 grid).

Input files

The following is a brief list of required input files (see Appendix B for further details and examples).

- `params.dat`: specifies a number of scalar parameters (read by the `init` subroutine).
- `coords.dat`: contains the x,y,z coordinates at the cell nodes.
- `boundary.dat`: describes the boundary types and locations.
- `veg.dat`: contains the vegetation pattern.
- `nodes.dat`: contains a list of the node numbers surrounding each grid cell.
- `vanG.dat` : Van Genuchten parameters and initial H as a function of depth for vegetated and bare soil cells.

Grid variables

- `nn` : space allocated to the grid (`nn>np`).
- `np`: the number of cell nodes.
- `x(nn), y(nn), zz(nn)` : x, y, z coordinates of the cell nodes.
  `x,y,zz` are read as 1D arrays, and interpolated to a 2D grid with the help of the nodes file `nodes.dat`.
- `xc, yc, zc` : coordinates at the cell centers, with dimensions `(ncol, nrow)`
- `nop(nx,ny,4)`: node numbers defining each grid cell (see Figure 5).
- `inum`: number of boundary interfaces of each cell.
- `itype`: interface type of each cell boundary (1 for wall boundaries, 0 for open boundaries...).
- `ipos`: position of each cell boundary (1 for lower boundaries, 2 for right boundaries...).

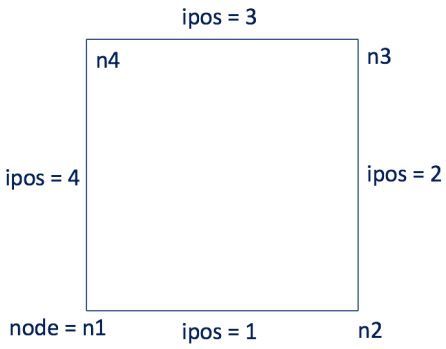


Figure 5: Schematic of the nodes and boundary naming convention.

3.3 Model output

`dry.f` saves the following files in the `output` subfolder. Most variables are saved at intervals of dt_p, to limit the size of the output files, with the exception of the hydrograph, which is saved at 1 s resolution.

- `dvol.out`: output file to check mass balance. Read by function `get_dvol` in `output_dry.py`.
- `fluxes1234.out`: lateral boundary fluxes, grouped by boundary position (i.e. boundary positions 1-4).
- `h.out`: primitive variables h,U, V, infiltration I and interfacial fluxes (spatially-distributed fields, saved at every grid-cell)
- `summary.out` : summarizes ponding time, runtime, final time, and if applicable, the reason for an early exit (i.e. no more water, AMAX too big)
- `hydro.out`: hydrograph at 1 second time resolution (m^3/s).
- `ptsTheta.out` : soil moisture profiles at two points, one vegetated and one bare.
- `time.out`: file containing time, and max CFL number at each timestep.

3.4 Principal model components (`dry.f` subroutines)

This section provides a high level summary of the main model components (i.e. `dry.f` subroutines) and how they connect. Further information and details can be found in Appendix A.

The predictor step

The `predict` subroutine is called to compute `hp`, `up`, `vp`,corresponding to $h^{n+1/2}$,$u^{n+1/2}$,$v^{n+1/2}$, at each cell. It modifies the common variables `hp`, `up`, `vp`, `dh`, `du`, `dv`, `qs`.

Corrector step

The corrector step computes the fluxes between the cell interfaces by calling the `fluxes` subroutine for each cell and for each interface.

The interfacial cell fluxes at a given time-step are stored in the common variable `f(0:nx,0:ny,1:3,1:2)`, where the first two indices of `f` contain the j, k coordinates, the third index correspond to the components of $\mathbf{F}_\perp$ (see Equation), and the final index denotes the cell face (1 for vertical cell faces and 2 for horizontal). `f(j,k,1:3,1)` represents the flux from cell $(j-1, k)$ to (j, k), and `f(j,k,1:3,2)` represents the flux from cell $(j, k-1)$ to (j, k) (see Figure **??**). Calling `fluxes(j-1,j,k,k,1)` modifies `f(j,k,1:3,1)` and calling `fluxes(j,j,k-1,k,2)` `f(j,k,1:3,2)`.

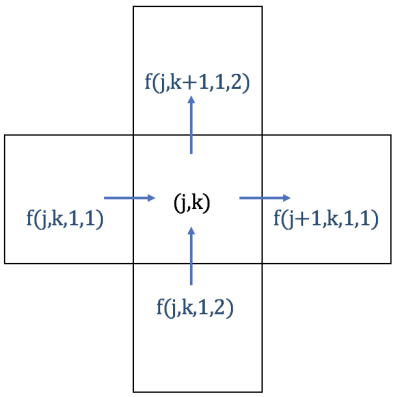


Figure 6: Definitional sketch for the interfacial fluxes, indicating how the fluxes into and out of cell (j,k) are stored in the array `f(0:nx,0:ny,1:3,1:2)`

.

Fluxes subroutine

Within the `fluxes` subroutine, `fluxes(jl,jr,kl,kr,i1)` computes the flux from cell `(jl,kl)` to `(jr,kr)`, modifying the common variable `f(jr,kr,1:3,i1)`. `i1 = 1` indicates that the flux is between vertical cell faces, i.e. from cell `(j-1,k)` to `(j,k)`. Similarly, `i1 = 2` indicates that the flux is between horizontal cell faces, i.e. from cell `(j,k-1)` to `(j,k)`.

`fluxes` first reconstructs the predicted values to the left and right of each face (`hl,hr,ul, ur,vl,vr`) using the monotone upstream scheme for conservation laws (MUSCL). These reconstructed predictor values define a Riemann problem at each cell face, and `fluxes` calls the `solver` subroutine to compute $\mathbf{F}_{\perp\mathbf{I}}$ from Equation **??**, reproduced below:

$$\mathbf{F}_{\perp\mathbf{I}} = \frac{1}{2}(\mathbf{F}_{\perp\mathbf{L}} + \mathbf{F}_{\perp\mathbf{R}} - \hat{\mathbf{R}}|\hat{\mathbf{\Lambda}}|\hat{\mathbf{\Delta V}})$$

The `source` subroutine

The source term is computed with the subroutine `source`, which modifies the common variable `qs` (or $Q(j,k)$). `source` is called by both the predictor and corrector steps; however, Richards equation is only solved during corrector step. The input arguments are the cell indices `j,k`, the primitive variables, and an indicator variable `lstep` for the step type (`lstep = 0` for the predictor step and 1 for the corrector step.

In the predictor step, `source` is called for each cell before updating the predictor values (`hp,up,vp`). `source` is similarly called in the corrector step before updating the fluxes.

Infiltration details (corrector-step)

In the corrector step, `source` calls the infiltration-specific subroutines (`timestep` and `potential` to update the infiltration rate i and the surface boundary conditions at cell `j,k`. `timestep` solves Richards equation at cell `(j,k)` and returns the updated H, θ and K, which `source` uses to compute the infiltration rate i. `potential` is used to prescribe the boundary conditions.

The Richards equation solver is implemented separately for each grid cell, and the 3-dimensional H, θ and K fields are saved as the common variables `r8H` (cm), `r8Theta`, and `r8K` (cm/s), respectively, with dimensions (`nrow` $\times$ `ncol` $\times$ `nz`).

The `potential` subroutine computes the infiltration that would occur with $H = 0$ at the surface (PI/`PI`), and is required for Richards case 2 in Figure 4.

From Darcy's law, the infiltration rate i (cm/s) is computed as:

$$i = K\left(\frac{\partial H}{\partial z} + 1\right)$$

which corresponds to `r8kt*((hdum(nz)-hdum(nz-1))/dz+1)` in `source`.

As an intermediate step, the depth is updated with:

```
znew= zold + prate*100*dt - r8kt*( (hdum(nz) - hdum(nz-1))/dz + 1)*dt
```

where `prate` rainfall has been converted to cm/s. `winflt` $= p - i$ is then computed as:

```
winflt = (znew - zold)/dt/100.
```

Richards equation is solved by the `timestep` subroutine, which is called from `source`. Richards equation is solved every `iscale` timesteps. For all other SVE time steps, the infiltration rate is estimated as:

```
r8kt*((hdum(nz) - hdum(nz-1))/dz + 1.d0)*dt
```

if the depth is greater than zero. Ponding does not occur until the `potential` subroutine determines that the rainfall exceeds to potential infiltration rate ($p > PI$).

When there is rain but no ponding (`depth=0` and `prate > 0`), the `potential` subroutine is called to estimate a potential infiltration rate PI, defined as the infiltration that would occur with $H = 0$ at the surface (in cm/s).
If the potential infiltration rate is less than the rainfall rate (`(PI .lt. prate*100)`), ponding begins. In this case, the Richards boundary condition is switched to fixed H.

4 The Simulated Domain

To simulate runoff/runon environments on patchily-vegetated hillslopes, several aspects of the BK model have been modified (in addition to coupling to a 1D Richards solver). This section describes the simulated domain, as illustrated in Figure 7.

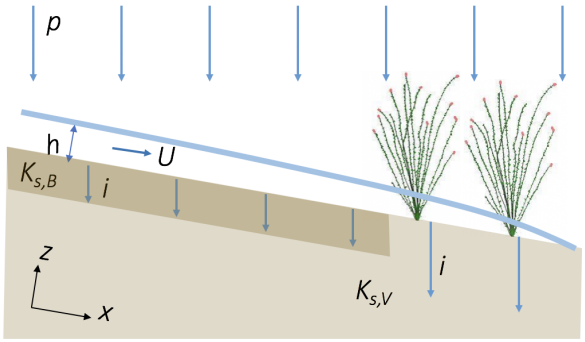


Figure 7: Schematic of the model domain. The darker shading under the non-vegetated regions represents the surface crust. $K_{s,V}$ and $K_{s,B}$ represent the saturated hydraulic conductivities of undisturbed soil and surface crust, respectively.

First, roughness parameters are specified separately for vegetated and bare cells. The parameters for the resistance formula for vegetated cells are specified as `alpha`, `eta` and `m` (α, η and m), and for bare soil cells as `alpha_B`, `eta_B` and `m_B` (α_B, η_B and m_B). The default soil parameters are Manning's equation with `alpha = 0.1`, and `alpha_B = 0.03`.

The model is set up to specify rain-driven overland flow on a planar hillslope, for which the lateral boundaries are closed with the exception of the open boundary at the bottom of the hillslope. Limited support is provided for a fixed flux (subcritical) upslope boundary condition, where the inflow boundary dimensions are equal to the entire upslope area. Code for other boundary conditions is not provided, but can be implemented with modification to `input_coords.py` (or directly to the Fortran input parameter functions). For a fixed flux upslope boundary, `tr` also specifies that the time as which the inflow stops (i.e. the boundary changes from fixed flux to a wall boundary). In the case of rain AND upslope inflow, for example, to simulate rain on a longer hillslope domain, the rain and inflow will end at the same time. The code would need to be modified with the addition of a new variable (e.g. time of inflow end) to change this, which would be straightforward.

The bare soil areas are represented by a two-layer model, where the upper, less-permeable layer represents a surface crust. With the exception of K_s, the van Genuchten parameters are the same between vegetated and bare cells. In the bare soil areas, K_s of the lower soil layer is equal to that in the vegetated patches (See schematic). The van Genuchten parameters and vertical structure of the soil can be modified in `input_phi.py`.

5 Python wrapper scripts

- `write_nodes(path, ncol, nrow)` : writes the cell node indices to `input/nodes.dat`. This function assumes that the grid is rectangular, and would need to be modified for a non-rectangular grid. It saves `nop`, which contains the indices of the nodes surrounding each cell face.
- `build_coords(params)` : constructs the `x,y,z` coordinates fields.

Example Python scripts are provided to generate the Fortran input files, execute `dry.f` and read the outputs. `call_dry.py` is a control script which executes each of these components, as illustrated in the Figure 8 schematic.

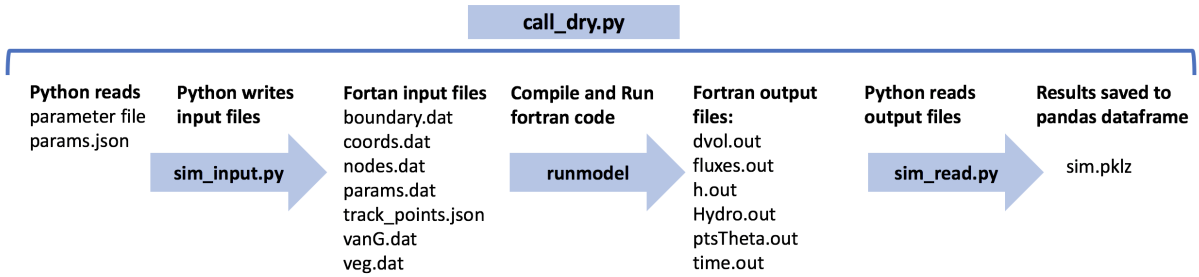


Figure 8: Schematic illustrating the python wrapper script workflow. `call_dry` takes as input `sim_name`, the name of a subfolder containing the parameter file `params.json`.

`output_dry.py` contains three main sub-functions, listed below:

- `call_dry.py`
  - `sim_input.py`
  - `runmodel`
  - `sim_read.py`

`call_dry.py` is a control script which writes the Fortan input files, compiles and executes the Fortran code dry.for, and reads the output files.

5.1 `sim_input.py`: writes the Fortran input files

`sim_input.py` takes the input arguement `sim_path`, the path to the simulation directory containing a parameter file `params.json`, and sets up the file structure required by `dry.f`:

- copies `dry.f` and `dry.inc` to `sim_path`
- makes `sim_path/input` and `sim_path/output` subdirectories
- reads and interprets the parameter file `params.json`

`sim_input.py` calls the following functions to write the input files (located in `sim_name/input`):

- `write_param`: writes the general parameter file `params.dat`.
- `input_veg.py`: contains functions to write the input vegetation file `veg.dat`.
- `input_coords.py`: contains functions to write the grid and topography files `coords.dat`, `nodes.dat`.
- `input_boundary.py` contains functions to write the boundary file `boundary.dat`.

See appendix B for examples of the Fortran input files.

5.1.1 `params.json`: specifies the parameters

SVE parameters include:

- `dx` : grid discretization (default = 1.)
- `nrow` : number of grid cells in the along-slope direction.

  hillslope length `Ly` = `nrow * dx`
- `ncol` : number of grid cells in the across-slope direction (default = 10).

  hillslope width `Lx` = `ncol * dx`
- `topo` : topography type (default = "plane", specifying a planar hillslope).
- `vegtype` : specifies the vegetation field type, with options:
  - `randv` : randomly-generated vegetation field, constructed with parameters f_V (vegetation fraction) and σ_x, σ_y, (across- and along-slope patch length-scales).
  - `image` : read vegetation field from an image file (e.g. jpeg or png). In this case, the image file name must be included under the field `image_name`.
- `alpha` / α_V : generalized roughness parameters for vegetated / permeable areas. If Manning's equation is used, then $\alpha = n$.
- `alpha_B` / α_B : equivalent to `alpha` / α for bare soil areas.
- `epsh` / ϵ: depth tolerance for dry cells. The momentum equations are not solved if `h< epsh` ($h < \epsilon$).
- `beta` / β : a constant in limiter subroutine if ilim = 5, beta family (default = 1).

Infiltration parameters include:

- `Ks` : infiltration rate in vegetated/permeable areas, in cm/hr.
- `KsB` : infiltration rate in bare soil/impermeable areas, in cm/hr.
- `stop_tol` : error tolerance for Richards equation convergence, default = 0.01
- `tsat_min`: default = 600. Specified in seconds. Minimum time until the soil is allowed to saturate (i is set to K_{sat}, bypassing the Richards equation solver
- `H_i` : initial H at the surface (the soil is initialized to an equilibrium profile).
- `tr` : storm duration (min)
- `p` : rain intensity (cm/hr)

5.2 `runmodel`: compiles and execute `dry.f`

The function `runmodel`, located in `call_dry.py`, uses the Python os library to compile and execute `dry.f` with command line arguments:

```
os.system("gfortran -o {0}/sw  -framework accelerate {0}/dry.for".format(sim_path))
os.system("cd {0} \n ./sw  \n cd {1}".format(sim_path, current_dir))
```

5.3 `read_sim`: reads the Fortran output files

The output files are read by `sim_read.py`, which reads the output files and saves the results as a dictionary `sim.pklz`. The variables to be saved are provided in the list `fortran_outvars`.

which calls the following functions to read the output files:

- `read_time`: reads `time.out`
- `read_hydro` : reads `hydro.out` and returns `t_h` and `hydro`. The hydrograph is normalized by the hillslope dimensions to convert m^3/s to cm/s.
- `get_h` : reads `h.out`, which contains the primitive variables (`h,u,v`), infiltration (`inflVmap`), and the inter-cell fluxes `xflux0,yflux0,xflux1,yflux1`, all with dimensions: `nprt x ncol x nrow` (where `nprt` is the number of timesteps).
- `get_dvol.py`: reads `dvol.out`, containing volume tracking variables `vol`, `flux` and `infl`, representing the total change in volume, lateral boundary fluxes and infiltration since the previous print timestep (i.e. `dt_p`). `get_dvol.py` normalizes by the domain area and converts to cm.

6 Python examples and Jupyter notebooks

The following examples are included to demonstrate the model functionality:

- `example_2by2`: illustrate the grid set-up for a 2×2 domain (see appendix B).
- `example_image`: rain driven overland flow on a planar hillslope with the vegetation field read from an image file.
- `example_inflow` illustrates a subcritical inflow boundary at the top of the hillslope, with rain = 0 and $K_s = 0$.
- `example_rain`: rain driven overland flow on a planar hillslope with the randomly-generated vegetation field.

Jupyter notebook files within each of these folders are included to visualize the simulated results.

Additionally, several template notebooks are included to process the SVE-R simulation results:

- `example_mass_balance.ipynb`: Check mass balance in the SVE-R model.
-

7 Feature code

Code to generate the features used to train the random forests is provided in `feature_functions.py`, with the Jupyter notebook `example_features.ipynb` provided to interface with the results. Within `RF_patterns.py`, the function `get_feature_matrix` takes an input binary (im)permeability pattern and returns a matrix of features, where each row corresponds to a grid cell in the input pattern.

References

[1] Bradford, S. F., & Katopodes, N. D. (1999). Hydrodynamics of turbid underflows. I: Formulation and numerical analysis. Journal of hydraulic engineering, 125(10), 1006-1015.

[2] Bradford, S. F., & Katopodes, N. D. (2001). Finite volume model for nonlevel basin irrigation. Journal of irrigation and drainage engineering, 127(4), 216-223. https://doi.org/10.1061/(ASCE)0733-9437(2001)127:4(216)

[3] Celia, M. A., Bouloutas, E. T., & Zarba, R. L. (1990). A general mass-conservative numerical solution for the unsaturated flow equation. Water resources research, 26(7), 1483-1496. https://doi.org/10.1029/WR026i007p01483

[4] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.

8 Appendix A: Relating `dry.f` code to SVE-R equations

The following is a partial list of grid correspondences:

- $U_\xi =$ `uxi`
- $U_\eta =$ `ueta`
- $d\xi =$ `dxi`
  - $\xi_x =$ `dxi(1)`
  - $\xi_y =$ `dxi(2)`
- $d\eta =$ `deta`
  - $\eta_x =$ `deta(1)`
  - $\eta_y =$ `deta(2)`
- $\frac{dz}{dx} =$ `sx`
- $\frac{dz}{dy} =$ `sy`

8.1 The predictor step

The `predict` subroutine is called for each cell to obtain `hp`, `up`, `vp` (corresponding to $h^{n+1/2}$,$u^{n+1/2}$,$v^{n+1/2}$). `predict` modifies the common variables `hp`, `up`, `vp`, `dh`, `du`, `dv`, `qs`.

The following provides an (incomplete) correspondence between BK equations and Fortran code, with the mathematical notation in *italics* and the equivalent code in `typewriter`. For example, for the array of primitive variables:

$$\mathbf{W_T} = \begin{bmatrix} h \\ U \\ V \end{bmatrix} = \begin{bmatrix} \texttt{h} \\ \texttt{u} \\ \texttt{v} \end{bmatrix}$$

The matrices $\mathbf{A}_W$ and $\mathbf{B}_W$ are defined as:

$$\mathbf{A}_w = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} = \begin{bmatrix} \texttt{uxi} & \texttt{h dxi(1)} & \texttt{h*dxi(2)} \\ \texttt{g dxi(1)} & \texttt{uxi} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{uxi} \end{bmatrix}$$

$$\mathbf{B}_w = \begin{bmatrix} U_\eta & h\eta_x & h\eta_y \\ g\eta_x & U_\eta & 0 \\ g\eta_y & 0 & U_\eta \end{bmatrix} = \begin{bmatrix} \texttt{ueta} & \texttt{h*deta(1)} & \texttt{h*deta(2)} \\ \texttt{g dxi(1)} & \texttt{ueta} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{ueta} \end{bmatrix}$$

$$\partial_\xi \mathbf{W} = \begin{bmatrix} \partial_\xi h \\ \partial_\xi u \\ \partial_\xi v \end{bmatrix} = \begin{bmatrix} \texttt{dh(1)} \\ \texttt{du(1)} \\ \texttt{dv(1)} \end{bmatrix}; \qquad \partial_\eta \mathbf{W} = \begin{bmatrix} \partial_\eta h \\ \partial_\eta u \\ \partial_\eta v \end{bmatrix} = \begin{bmatrix} \texttt{dh(2)} \\ \texttt{du(2)} \\ \texttt{dv(2)} \end{bmatrix}$$

where `dh(1)` implies `dh(j,k,1)`, and similarly for `du(1)` and `dv(1)`.

$\mathbf{A}_w \partial_\xi \mathbf{W}$ is computed as:

$$\mathbf{A}_w \partial_\xi \mathbf{W} = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} \begin{bmatrix} \partial_\xi h \\ \partial_\xi u \\ \partial_\xi v \end{bmatrix} = \begin{bmatrix} U_\xi \partial_\xi h + h\xi_x \partial_\xi u + h\xi_y \partial_\xi v \\ g\xi_x \partial_\xi h + U_\xi \partial_\xi u \\ g\xi_y \partial_\xi h + U_\xi \partial_\xi v \end{bmatrix}$$

Equivalently in code:

$$\mathbf{A}_w \partial_\xi \mathbf{W} = \begin{bmatrix} \texttt{uxi} & \texttt{h dxi(1)} & \texttt{h dxi(2)} \\ \texttt{g dxi(1)} & \texttt{uxi} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{uxi} \end{bmatrix} \begin{bmatrix} \texttt{dh(1)} \\ \texttt{du(1)} \\ \texttt{dv(1)} \end{bmatrix} =$$

$$\begin{bmatrix} \texttt{uxi*dh(1) + h*(dxi(1)*du(1) + dxi(2)*dv(1))} \\ \texttt{g*dxi(1)*dh(1) + uxi*du(1)} \\ \texttt{g*dxi(2)*dh(1) + uxi*du(1)} \end{bmatrix}$$

The `limitr` subroutine contains the flux limiter that the predictor step uses to compute the cell-average spatial gradients (which preserves solution monotonicity by becoming first-order accurate near

discontinuities, yet remains second-order accurate elsewhere). Various choices are included, with the parameter `ilim` specifying the averaging type. `ilim`=5 instructs the code to use the β family of averages, and is given by:

$$\overline{\mathbf{\Delta W}} = \text{sign}(a)\text{min}[\text{max}(|a|,|b|), \beta(|a|,|b|)] \text{ if } ab > 0$$
$$0 \text{ if } ab \leq 0$$

$\beta = 1$ yields the relatively more dissipative Minmod average, and $\beta = 2$ yields the less dissipative Superbee average.

8.2 The corrector step

The corrector step computes the fluxes between the cell interfaces by calling the `fluxes` subroutine for each cell `j,k` and for each interface.

The interfacial cell fluxes at a given time-step are stored in the common variable `f(0:nx,0:ny,1:3,1:2)`, where the first two indices of `f` contain the j, k coordinates, the third index correspond to the components of $\mathbf{F}_\perp$ (see Equation), and the final index denotes the cell face (1 for vertical cell faces and 2 for horizontal). `f(j,k,1:3,1)` represents the flux from cell $(j-1, k)$ to (j, k), and `f(j,k,1:3,2)` represents the flux from cell $(j, k-1)$ to (j, k) (see Figure **??**). Calling `fluxes(j-1,j,k,k,1)` modifies `f(j,k,1:3,1)` and calling `fluxes(j,j,k-1,k,2)` `f(j,k,1:3,2)`.

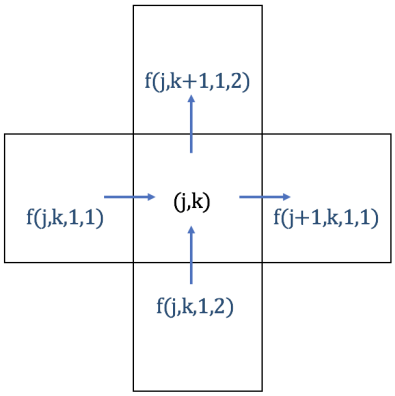


Figure 9: Definitional sketch for the interfacial fluxes, indicating how the fluxes into and out of cell (j,k) are stored in the array `f(0:nx,0:ny,1:3,1:2)`

.

Fluxes subroutine

Within the `fluxes` subroutine, `fluxes(jl,jr,kl,kr,i1)` computes the flux from cell `(jl,kl)` to `(jr,kr)`, modifying the common variable `f(jr,kr,1:3,i1)`. `i1 = 1` indicates that the flux is between

vertical cell faces, i.e. from cell `(j-1,k)` to `(j,k)`. Similarly, `i1 = 2` indicates that the flux is between horizontal cell faces, i.e. from cell `(j,k-1)` to `(j,k)`.

`fluxes` first reconstructs the predicted values to the left and right of each face (`hl,hr,ul, ur,vl,vr`) using the monotone upstream scheme for conservation laws (MUSCL). These reconstructed predictor values define a Riemann problem at each cell face, and `fluxes` calls the `solver` subroutine to compute $\mathbf{F}_{\perp\mathbf{I}}$ from Equation **??**, reproduced below:

$$\mathbf{F}_{\perp\mathbf{I}} = \frac{1}{2}(\mathbf{F}_{\perp\mathbf{L}} + \mathbf{F}_{\perp\mathbf{R}} - \hat{\mathbf{R}}|\hat{\mathbf{\Lambda}}|\hat{\mathbf{\Delta V}})$$

The solver subroutine

This subroutine uses the monotone upstream scheme for conservation laws (MUSCL) to compute $\mathbf{F}_{\perp}$ at each cell face. The code in `solver` corresponds to the Equation **??** as:

- $\hat{\mathbf{R}}$: `e(3,3)`
- $|\hat{\mathbf{\Lambda}}|$: `a(3) x`
- $\hat{\mathbf{\Delta V}}$: `ws(3)`

$$\hat{\mathbf{R}} = \begin{bmatrix} 1 & 0 & 1 \\ \hat{u} - \hat{a}\cos\phi & -\sin\phi & \hat{u} + \hat{a}\cos\phi \\ \hat{v} - \hat{a}\sin\phi & \cos\phi & \hat{v} + \hat{a}\sin\phi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ \texttt{uhat-chat*cndum} & \texttt{-sndum} & \texttt{uhat+chat*cndum} \\ \texttt{vhat-chat*sndum} & \texttt{cndum} & \texttt{vhat+chat*sndum} \end{bmatrix}$$

$$|\hat{\mathbf{\Lambda}}| = \begin{bmatrix} |\hat{u}_\perp - \hat{a}| & & \\ & |\hat{u}_\perp| & \\ & & |\hat{u}_\perp + \hat{a}| \end{bmatrix} = \begin{bmatrix} \texttt{|uperp - chat|} & & \\ & \texttt{|uperp |} & \\ & & \texttt{|uperp + chat|} \end{bmatrix}$$

$$\mathbf{\Delta}\hat{\mathbf{V}} = \begin{bmatrix} \frac{1}{2}\left(\Delta h - \frac{\hat{h}\Delta u_\perp}{\hat{a}} + \frac{\hat{h}\Delta c_T}{2c_T}\right) \\ \hat{h}\Delta u_\parallel \\ \frac{1}{2}\left(\Delta h + \frac{\hat{h}\Delta u_\perp}{\hat{a}} + \frac{\hat{h}\Delta c_T}{2c_T}\right) \end{bmatrix} = \begin{bmatrix} \texttt{0.5*(dhdum - hhat*duperp/chat)} \\ \texttt{hhat*dupar} \\ \texttt{0.5*(dhdum - hhat*duperp/chat)} \end{bmatrix}$$

- Δh : `dhdum = hr - hl`
- Left interface: `hl = hp(jl,kl) + 0.5*dh(i1)`
- Right interface: `hr = hp(jl,kl) - 0.5*dh(i1)`

8.3 The source subroutine

The source term is computed with the subroutine `source`, which modifies the common variable `qs` (or $Q(j,k)$). `source` is called by both the predictor and corrector steps; however, Richards equation is only solved during corrector step. The input arguments are the cell indices `j,k`, the primitive variables, and an indicator variable `lstep` for the step type (`lstep = 0` for the predictor step and 1 for the corrector step.

In the predictor step, `source` is called for each cell before updating the predictor values (`hp,up,vp`):

```
call source(j, k, h(j,k), u(j,k), v(j,k), 0)
```

`source` is similarly called in the corrector step before updating the fluxes:

```
call source(j, k, hp(j,k), up(j,k), v[(j,k), 0)
```

Inside `source`, the primitive variables are labeled `depth,udum,vdum`. The source term in Fortran code is given as:

$$\mathbf{Q} = \begin{bmatrix} \texttt{winflt} \\ \texttt{-grav*depth*sx(j,k)-grav*depth*fricSx+0.5*udum*winflt} \\ \texttt{-grav*depth*sy(j,k)-grav*depth*fricSy+0.5*vdum*winflt} \end{bmatrix}$$

where $S_{f,x} =$ `fricSx` and $S_{f,x} =$ `fricSy`, and $p - i =$ `winflt`.

Infiltration details (corrector-step)

In the corrector step, `source` calls the infiltration-specific subroutines (`timestep` and `potential` to update the infiltration rate i and the surface boundary conditions at cell `j,k`. `timestep` solves Richards equation at cell `(j,k)` and returns the updated H, θ and K, which `source` uses to compute the infiltration rate i. `potential` is used to prescribe the boundary conditions.

The Richards equation solver is implemented separately for each grid cell, and the 3-dimensional H, θ and K fields are saved as the common variables `r8H` (cm), `r8Theta`, and `r8K` (cm/s), respectively, with dimensions (`nrow` $\times$ `ncol` $\times$ `nz`).

From Darcy's law, the infiltration rate i (cm/s) is computed as:

$$i = K\left(\frac{\partial H}{\partial z} + 1\right)$$

which corresponds to `r8kt*((hdum(nz)-hdum(nz-1))/dz+1)` in `source`.

As an intermediate step, the depth is updated with:

```
znew= zold + prate*100*dt - r8kt*( (hdum(nz) - hdum(nz-1))/dz + 1)*dt
```

where `prate` rainfall has been converted to cm/s. `winflt` $= p - i$ is then computed as:

```
winflt = (znew - zold)/dt/100.
```

8.4 timestep: the Richards equation subroutine

Richards equation is solved by the `timestep` subroutine, which is called from `source`.

timestep inputs and outputs

The inputs are `hnp1m, thetan`, which are the initial conditions to the Richards eqn solver (`hdum,thetadum` in the source subroutine, which calls `timestep`). The outputs are `hnp1mp1,r8thetanp1m,r8knp1mp1`, which are H, θ and K at the following timestep (after dt_r time elapsed).

When the soil is flagged as saturated, the surface flux is set to K at the surface, which should be very close to saturated (`flux = - r8knp1m(nz)`). This is achieved by adjusting the value of H at node (nz-1):

```
hnp1mp1(nz-1) = hnp1mp1(nz) + dz + flux*dz/r8knp1m(nz-1)
```

Note: this approach should be treated with caution. Is was designed for the use case of a fixed intensity rain storm, in which a uniform wetting front would arise. It is not meant for variable rainfall cases.

Stop tolerance

`stop_tol0` is the stop tolerance specified in the input `params.dat`, with default value `stop_tol0 = .01`. `stop_tol` is reset to the original `stop_tol0` at the beginning of each Richards solver timestep. The convergence tolerance is relaxed if the Richards solver fails to converge within a specified number of iterations (default = 100).

Potential infiltration

The subroutine `potential` is used to compute the potential infiltration rate PI, defined as the infiltration that would be observed for a surface boundary condition of $H = 0$. Potential infiltration is computed when there is rain but no ponding, and returns the potential infiltration rate, defined as the infiltration rate that would occur with $H = 0$ at the surface.

9 Appendix B: Input and output files, illustrated with a 2×2 grid example

To illustrate the grid set-up, example input files are included here for the 2x2 grid example. Note that while the SVE-R model Fortran code does not assume a rectangular grid, the provided Python code and sample input files do assume a rectangular grid. Figure 10 shows the cell center and node indices for the example grid, and Figure 11 shows the boundaries, which are a fixed-flux subcritical boundary at the top of the hill, closed/wall lateral boundaries, and an open boundary at the bottom of the hill.

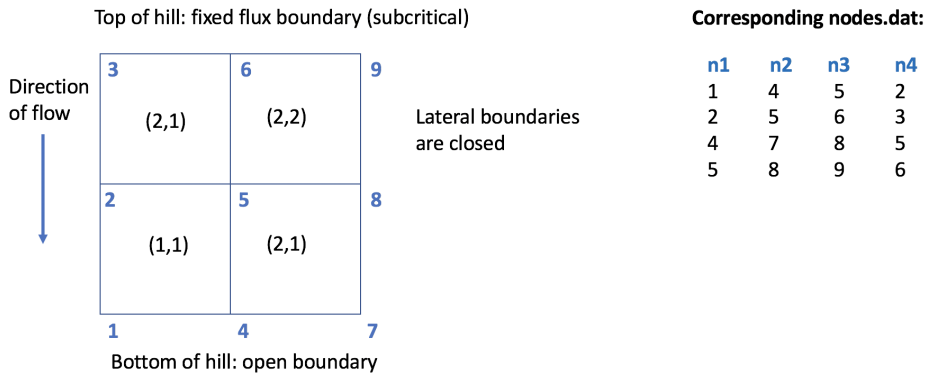
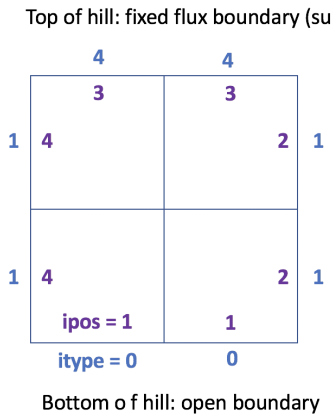


Figure 10: Example 2×2 grid to illustrate how cell centers and nodes are assigned. (j,k) coordinates are shown in the cell centers, and the node indices are at the cell corners.

9.1 params.dat

Sample `params.dat` files can be found in the `input` directories of the examples. The parameters specified in `params.dat` are:

- `grav` : acceleration due to gravity.
- `dt` : SVE timestep (`dt` is labeled `dt_sw` in the Python code).
- `tmax` : maximum time until the simulation ends.
- `tr` : rain duration in seconds (`tr` is labeled `t_rain` in the Python code).
- `prate` : rain intensity in m/s.
- `nt` : number of time steps (`tmax/dt`)
- `epsh` : depth threshold to solve the SVE momentum equations.
- `beta` : value of beta in limitr subroutine.
- `nprt` : frequency f which the SVE-R output fields are saved: `nprt= dt_p`. Note: the hydrograph is saved every second.
- `iscale` : ratio of SVE to Richards equation timestep durations.
- `stop_tol` : convergence criteria for Richards equation solver (default = 0.01).
- `h0, u0, v0` : initial depth, `u`, and `v` (defaults = 0).
- `r8mV,r8etaV, r8alphaV` : roughness parameters m, η and α for vegetated cells.
- `r8mB,r8etaB, r8alphaB` : roughness parameters m, η and α for bare cells.
- `jveg, kveg` : `j, k` indices of a vegetated cell for which the soil profile (H and θ) is saved every `nprt` timesteps.



Closed lateral boundaries

Corresponding summary of boundary cells

number of boundary cell
4

j	k	inum	itype		ipos	
1	1	2	1	0	4	1
1	2	2	1	4	4	3
2	2	2	4	1	3	2
2	1	2	1	0	2	1

Figure 11: Boundaries for the 2×2 grid in Figure 11. The boundary types are labeled in blue outside the grid, with 0 for the open boundary, 1 for the closed boundaries and 4 for the fixed flux boundary.

- `jbare, kbare` : `j, k` indices of a bare soil cell for which the soil profile (H and θ) is saved every `nprt` timesteps.
- `tsat_min` : Minimum time until the soil is allowed to "sarturate" (default =0).

9.2 boundary.dat

`boundary.dat` specifies the boundary cell locations, types and orientations. For each boundary cell location (`j,k` indices), the number of boundary faces (`inum`), types (`itype`) and orientations (`ipos`) must be specified. `boundary.dat` also lists any fixed boundary condition cells (`fix`) and the relevant primitive variables (h, U and/or V, depending on the boundary type).

Provided boundary types are:

- `itype` = 1 for a closed boundary
- `itype` = 0 for an open boundary
- `itype` = 4 for a subcritical inflow boundary
- `itype` = 5 for a supercritical influx boundary

For a subcritical inflow boundary, the normal depth is computed with Manning's equation with $n = 0.1$. The input parameter `influx` is specified in unis m^2/s, which can be related to the rainfall rate (in cm/hr) as: $p = q \cdot 3.6e5/L_y$ (cm/hr), or `influx` $q = p \cdot L_y/3.6e5$ (m^2/s).

Below is a sample `boundary.dat` for the 2x2 grid example, for which every cell is a boundary cell. The first two rows contain the number of boundary cells, defined as a grid cell with one or more boundaries. The array on lines 3-7 show the (j,k) indices for each boundary cell, the number of boundaries associated with that cell, and boundary type of each boundary cell.

```
number of boundary cell
  4
 j    k          inum    itype             ipos
1     1             2       1        0         4        1
1     2             2       1        4         4        3
2     2             2       4        1         3        2
2     1             2       1        0         2        1
ncol
2
nrow
2
j     kbeg          kend
    1      1             2
    2      1             2
number of fixed bc cells, ndir
 2
 j     k    fix h    fix u    fix v
1     2     0.0     0.0     -2e-05
2     2     0.0     0.0     -2e-05
```

9.3 coords.dat

In the two rows specify the number of nodes (`npt`) and the number of cells (`ne`). The remaining rows list the x,y,z coordinates at the nodes, in order of node number. Below is an example `coords.dat` file for the 2×2 exampleg rid.

```
npt             ne
9               4
x             y             z
0.00          0.00          0.00
0.00          1.00          0.02
0.00          2.00          0.04
1.00          0.00          0.00
1.00          1.00          0.02
1.00          2.00          0.04
2.00          0.00          0.00
2.00          1.00          0.02
2.00          2.00          0.04
```

9.4 vanG.dat

`vanG.dat` specifies the soil dimensions, van Genuchten parameters and initial H profile, independently for vegetated and bare soil cells (Note: all vegetated cells have the same parameters and ICs, and likewise for the bare soil cells). See example below from the 2×2 grid example. The first two rows specify the soil depth (`zmax`) and soil layer thickness (`dz`), and the second two rows specify the number

of soil grid points (soil layers + 1). Following this, the soil parameters and initial H for the vegetated cells are listed for each soil layer in order of decreasing soil depth (the first row is the lowest layer of the soil column). This format is repeated for the bare soil cells. From left to right, the columns are: α,θ_S,θ_R, λ, K_s, H_i, where $\lambda = n - 1$ in Equation 5 and H_i is initialize with an equilibrium soil moisture profile.

```
dz            zmax
1.0            20
nz
21
vegetated cells
alpha    theta_S   theta_R    lambda    Ksat                    h_init
0.0096   0.472     0.0378     0.47      0.000555555555556       -320.0
0.0096   0.472     0.0378     0.47      0.000555555555556       -321.0
0.0096   0.472     0.0378     0.47      0.000555555555556       -322.0
0.0096   0.472     0.0378     0.47      0.000555555555556       -323.0
....
 bare soil cells
alpha    theta_S   theta_R    lambda    Ksat                    h_init
0.0096   0.472     0.0378     0.47      0.000555555555556       -320.0
0.0096   0.472     0.0378     0.47      0.000555555555556       -321.0
0.0096   0.472     0.0378     0.47      0.000555555555556       -322.0
```

9.5 h.dat

`h.out`: columns are `j,k,h,u,v,zinflmap2, xflux0,yflux0,xflux1,yflux1`. These variables are written for every grid cell, every `nprt` timesteps (i.e. at time intervals of `dt_p`). Following this, the print iteration `itp` and current time t is written on a new line. An example for a 2×2 grid is shown in Figure 12.

```
j  k  h              u              v              zinflmap2      xflux0         yflux0         xflux1         yflux1
1  1  0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
1  2  0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
2  1  0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
2  2  0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00   0.000000E+00
0   0.00
1  1  0.976960E-05   0.000000E+00   0.000000E+00  -0.292207E-04   0.000000E+00  -0.973063E-07   0.000000E+00  -0.390816E-04
1  2  0.268793E-03   0.000000E+00  -0.597791E-02  -0.292207E-04   0.000000E+00  -0.390876E-04   0.000000E+00  -0.337101E-03
2  1  0.976960E-05   0.000000E+00   0.000000E+00  -0.292207E-04   0.000000E+00  -0.973063E-07   0.000000E+00  -0.390816E-04
2  2  0.268793E-03   0.000000E+00  -0.597791E-02  -0.292207E-04   0.000000E+00  -0.390876E-04   0.000000E+00  -0.337101E-03
1   10.00
1  1  0.370174E-04   0.000000E+00   0.000000E+00  -0.389400E-04   0.000000E+00  -0.161459E-05   0.000000E+00  -0.678008E-04
1  2  0.335986E-03   0.000000E+00  -0.690895E-02  -0.389400E-04   0.000000E+00  -0.678023E-04   0.000000E+00  -0.173935E-03
2  1  0.370174E-04   0.000000E+00   0.000000E+00  -0.389400E-04   0.000000E+00  -0.161459E-05   0.000000E+00  -0.678008E-04
2  2  0.335986E-03   0.000000E+00  -0.690895E-02  -0.389400E-04   0.000000E+00  -0.678023E-04   0.000000E+00  -0.173935E-03
2   20.00
1  1  0.683697E-04   0.000000E+00   0.000000E+00  -0.418626E-04   0.000000E+00  -0.554588E-05   0.000000E+00  -0.787600E-04
1  2  0.380046E-03   0.000000E+00  -0.748920E-02  -0.418626E-04   0.000000E+00  -0.787608E-04   0.000000E+00  -0.164684E-03
2  1  0.683697E-04   0.000000E+00   0.000000E+00  -0.418626E-04   0.000000E+00  -0.554588E-05   0.000000E+00  -0.787600E-04
2  2  0.380046E-03   0.000000E+00  -0.748920E-02  -0.418626E-04   0.000000E+00  -0.787608E-04   0.000000E+00  -0.164684E-03
3   30.00
```

Figure 12: Screenshot of a sample codeh.out file.

9.6 `dvol.dat`

Contains output from Fortran mass balance tracking. Columns are `dvol, flux, infl` containing the change in surface volume, horizontal fluxes out of the domain, infiltration and rain volumes, respectively, in units m^3.

hydro.out

10 Appendix C : `dry.f` pseudocode

10.1 Model overview/summary

```
call init ! grid subroutine called from the input subroutine
call myoutput ! write initial conditions to file


do it = 0,nt-1 ! loop over time steps (it = time iteration number)
   t = t+dt  ! increment time (t = current time)

   ! Predictor step
   loop over grid cells:
      call bconds(j, k, h, u, v) ! update h,u,v values in ghost cells
      call predict(j, k)         ! get predictor variables hp, up, vp

    ! Corrector step
   ! Corrector part 1: compute the fluxes between cell faces
   loop over grid cells:
    call bconds(j,k,hp,up,vp) ! update hp,up,vp in the ghost cells
    ! compute interfacial fluxes.
    call fluxes(j-1,j,k,k,1)   ! vertical faces.
    call fluxes(j,j,k-1,k,2)   ! horizontal faces.
   loop over boundaries:
      compute boundary fluxes

   ! Corrector part 2:
   loop over cells:
      call source ! update the source term qs
      compute q from qs and f ! q = (h,uh,vh)
      ! update h,u,v from q. check
       if q > 0:         ! check for negative depth.
         h = q(1)         ! if positive depth, update h
       else:
         q = 0           ! if negative depth, reset h and q(1) to zero
         h = 0

       if h < epsh:    !  neglect momentum in nearly dry cells
```

```
        u, v = 0.     !  set u and v to 0 in nearly dry cell
        q(2:3) = 0    !  set momentum fluxes to 0

       elif h >= epsh:   !  store u and v values
        u, v = q(2)/h, q(3)/h

    ! Exit when the ponded water is less than min_vol
    if volume < min_vol
      call gracefulExit

    ! Exit when the CFL number is too big
    if(CFL > 10) then
       call gracefulExit

   ! Exit if time runs out
    if(t > tmax) then
        call gracefulExit

\end{verbatim}



\subsection{Grid pseudocode}
Pseudocode for grid setup:

\begin{lstlisting}
subroutine grid
include "dry.inc" ! include common variables
read np, ne   ! np: number of grid points, ne : number of cells
read x, y, zz ! from coords.dat, coordinates of the cell nodes
read veg      ! from veg.dat, vegetation at the cell nodes
read nop   ! from nodes.dat, the node numbers surrounding each cell.

! Compute grid metrics.
for each cell (j,k):
   n1, n2, n3, n4 = nop(j,k,1:4) ! get the node numbers
   ! compute xc, yc, zc as the average of x,y, zz at the surrounding nodes
   xc(j,k) = 0.25*(x(n1) + x(n2) + x(n3) + x(n4))

   compute dxi, deta, area ! compute grid metrics
   compute sx, sy, dz, ds, sn, cn

loop over cells: ! Set values in the ghost cells
   loop over faces:
      call findbc(i,j,k,jj,kk,j2,k2) ! get ghost cell indices (jj, kk)
      set sx, sy, dxi, deta in ghost cell equal to values in boundary cell (j,k)
```

10.2 Source pseudocode

```
! select the soil profiles from the r8H, r8Theta, r8K
hdum = r8H(j,k,1:nz)
thetadum = r8THETA(j,k,1:nz)
r8kdum = r8K(j,k,1:nz)

! convert the ponded depth to cm to use with as the Richards equation BC
zold = depth*100.d0 ! input depth in cm
znew = zold   ! initialize depth after the infiltration step in cm
PI = 0.d0    ! Initialize the potential infiltration as 0.
iskip = 0    ! Initialize the infiltration indicator (iskip = 1 if Richards solver is
    skipped, in which case r8H, r8Theta and r8K matrices are not updated).

! determine whether the cell is vegetated or not, and define the roughness parameters
    accordingly:
if vegetated cell:
    isveg = 1
    fm,falpha,feta = r8mV, r8alphaV, r8etaV
else: ! bare cell
   isveg = 2
    fm,falpha,feta = r8mB, r8alphaB, r8etaB

! only update depth in the corrector step
if lstep = 1 then

! Before the Richards equation solver is called, several cases need to be handled.
! only solve Richards equation every iscale timesteps
   if mod(it, iscale) != 0 then
     if there is ponding, set the the surface flux / infiltration rate using K from the
         previous timestep
! Case 2: rain and no ponding
   else (if prate > 0 and zold = 0) then
     call potential(hdum, thetadum, PI) ! determine the potential infiltration rate PI. Note:
         potential is a subroutine that takes hdum and thetadum as inputs and returns PI as
         the output.
!    Handle two cases: PI > prate (no ponding) and PI < prate (ponding starts)
     if  prate > PI then
         record the time of ponding, tp
        isetflux = 0  ! set the Richards boundary type to fixed H (Dirichlet), with htop=0 at
            the surface.
        call Richards equation with the updated boundary conditions.
     else   ! no ponding
       isetflux = 1
       flux = - prate*100. ! set the infiltration  rate to the rainfall rate
        winflt = (znew - zold)/dt/100.d0
! Case 3: no ponding and no rain
    else if (zold = 0 and prate = 0) then
       isetflux = 1 ! set the Richards surface boundary condition to fixed flux
```

```
      flux = 0    ! set flux to zero
      call Richards equation solver to update the soil moisture profile
      winflt = 0
! Case 4: ponding (with or without rain)
   else if zold > 0 then
       isetflux = 0 ! set the Richards surface boundary condition to fixed H
      htop = zold    ! set surface BC to ponded depth
      call Richards equation solver with updated boundary conditions.

       update depth (znew) using the infiltration rate from the updated K
       winflt = (znew - zold)/dt/100.d0
! End of Richards equation solver cases.

   if iskip = 0 ! if Richards equation was called by timestep
         ! update r8THETA, r8H, r8K (3D soil matrices) with 1D solver results
         r8THETA(j,k,1:nz) = thetap1dum
        r8H(j,k,1:nz) = hp1dum
        r8K(j,k,1:nz) = r8kp1dum

         compute r8fluxin (surface flux)

         if ipass = 0 ! if Richards equation was not passed (due to saturation flag)
         compute r8fluxout ! drainage flux
      else ! saturation flag went off
          r8fluxout = r8fluxin ! assume soil flux out = flux in
      compute r8newmass ! change in soil moisture since the previous timestep

      ! update oldTHETA (soil moisture content from the previous timestep)
      oldTHETA(j,k,1:nz) = r8THETA(j,k,1:nz)
else  ! in the predictor step, there is no infiltration or precipitation
   winflt = 0.d0

if (depth > epsh) then ! depth greater than threshhold
       ! compute magnitude
       vmag = dsqrt(udum*udum + vdum*vdum)

       if (feta .eq. 0.5) then ! non-laminar schemes

          fricSx = (falpha/depth**fm)**(2.d0)*udum*vmag
          fricSy = (falpha/depth**fm)**(2.d0)*vdum*vmag


       elseif (feta .eq. 1) then ! special case for laminar

          fricSx = falpha*udum/depth**fm
          fricSy = falpha*vdum/depth**fm

          ! include a catch for high Re cases: use DW ff with f = 0.5
          Rel = vmag*depth/1.e-6
```

```
        if (Rel .gt. 500) then
            ffact = 0.5 ! okay for smooth surfaces (following Kirstetter)
            fricSx = ffact*udum*vmag/8./grav/depth
            fricSy = ffact*vdum*vmag/8./grav/depth
        endif

      endif

     ! update the source terms
      qs(1) = winflt

      qs(2) = 0.5D0*udum*winflt - grav*depth*fricSx -
     &         grav*depth*sx(j,k) ! sx(j,k) = x-dir bed slope
      qs(3) = 0.5D0*vdum*winflt - grav*depth*fricSy -
     &         grav*depth*sy(j,k) ! sx(j,k) = y-dir bed slope
     else

      qs(1) = winflt
      qs(2) = 0.d0
      qs(3) = 0.d0

     endif
```

10.3 `timestep` pseudocode

Pseudocode for subroutine `timestep`:

```
 subroutine timestep(hnp1m,thetan,hnp1mp1,r8thetanp1m,r8knp1mp1)
!
!   Input:
!            hnp1m, thetan (real, kind = 8) - initial
!   Output:
!            hnp1mp1,r8thetanp1m,r8knp1mp1 - h, theta and k at time m+1
!
!  Comments: uses common variables nz, stop_tol, htop as surface h
!            modifies common variable ipass
  include 'dry.inc'


  declare variables hnp1m(nz), thetan(nz), r8cnp1m(nz), r8knp1m(nz) ... ! input and output
      arrays, and arrays used by the Richards eqn solver
  istop_flag = 0 ! indicator variable switches to 1 when convergence criteria is met
  niter = 0 ! number of iterations
  stop_tol = stop_tol0 ! reset stop tolerance to input (in case condition was relaxed)
  ipass = 0 ! indicator variable, set to 1 if soil is 'saturated'

  do while stop_flag = 0:
     ! r8cnp1m,r8knp1m,r8thetanp1m given hnp1m
```

```
  call vanGenuchten(k,hnp1m(k),
     r8cnp1m(k),r8knp1m(k),r8thetanp1m(k), isveg)

Do some linear algebra (see Celia et al. (1990)

! Compute deltam, the increment in iteration for iteration m+1
deltam = matmul(Ainv, R_MPFD)

increment niter (number of iterations)
niter = niter + 1

if niter > 100
   stop_tol = stop_tol*10 !relax stop tolerance
   if stop_tol > 10:
      quit and return error

! Handle saturation cases
compute t2b_theta = theta(top) - theta(bottom) ! the soil moisture difference between the
    top and bottom of the soil profile.

! Test whether the soil has saturated
! After time tsat_min, if there's ponded water at the surface:
! If the soil moisture difference between the surface and bottom is very small, then:
if t > tsat_min and depth > 0
  if t2b_theta < 0.005, then
    hnp1mp1, thetanp1m = hnp1m, thetan ! update h and theta to (n+1, m+1)
    flux = - r8knp1m(nz)  ! set flux to K(surface)
    ! adjust hnp1mp1(nz-1) to obtain this flux
    hnp1mp1(nz-1) = hnp1mp1(nz) + dz + flux*dz/r8knp1m(nz-1)

    ipass = 1   ! flag the soil as saturated
      istop_flag = 1 ! exit Richards solver - don't wait for convergence
! Give the soil a chance to unsaturate after the rain, and the water has drained
! Test whether a saturated soil has unsaturated
! After the storm, if there's no ponding
if t > tr and depth = 0
  if t2b_theta < 0.005, then !if the soil moisture difference between the surface and
      bottom is very small, then the soil was flagged as saturated.

   hnp1mp1, thetanp1m = hnp1m, thetan ! update h and theta to (n+1, m+1)

   flux = 0   ! set surface flux to zero

   ! set hnp1mp1(nz-1) to achieve zero flux BC at the surface
   hnp1mp1(nz-1) = hnp1mp1(nz) + dz
   ipass = 0  ! unfreeze the soil
   isetflux = 1 ! set a fixed, zero flux boundary condition


if (ipass eq 0) then ! if the soil is not saturated then...
```

```
if max(deltam) < stop_tol then ! convergence criteria has been met
   istop_flag = 1

      hnp1mp1 = hnp1m + deltam ! update h(n+1,m) to (n+1,m+1)

      ! apply surface boundary conditions
      if (isetflux .eq. 0) then ! fixed H
        hnp1mp1(nz) = htop
      elseif (isetflux .eq. 1) then ! fixed flux
        r8gkt = r8knp1m(nz-1)
        hnp1mp1(nz) =  hnp1mp1(nz-1) - dz - flux*dz/r8gkt
      endif
      ! apply free drainage BC at the lower boundary
      hnp1mp1(1) = hnp1mp1(2)

      call vanGenuchten

   else  ! update h and keep iterating

      hnp1mp1 = hnp1m + deltam ! update h(n+1,m) to (n+1,m+1)
      hnp1m = hnp1mp1 ! update the old h(n+1,m)

   ! apply surface and lower boundary conditions
```