Supplement to The Wageningen Lowland Runoff Simulator (WALRUS): a lumped rainfall-runoff model for catchments with shallow groundwater

C. C. Brauer, A. J. Teuling, P. J. J. F. Torfs, R. Uijlenhoet
Hydrology and Quantitative Water Management Group, Wageningen University, The Netherlands
claudia.brauer@wur.nl

1 Introduction

In this supplementary document we provide the main code for the Wageningen Lowland Runoff Simulator (WALRUS). The code is written in R and will be made available on the R CRAN website. WALRUS is licensed under the GPL v3 licence.

2 Script 1: loop over time steps

```
WALRUS_loop = function(p)
  {
  # compute number of o_steps
  L             = length(output_date)
  # make empty vectors for output states and fluxes
  o             = data.frame(matrix(nrow=L, ncol=11, dimnames=list(NULL,
                  c("ETact","Q","fGS","fQS","dV","dVeq","dG","hS","hQ","W","dt_ok"))))

  # look up soil type parameters
  p$b           = soil_char[["b"]]       [soil_char[["st"]]==p$st]
  p$psi_ae      = soil_char[["psi_ae"]]  [soil_char[["st"]]==p$st]
  p$theta_s     = soil_char[["theta_s"]][soil_char[["st"]]==p$st]
  p$aG          = 1-p$aS

  # INITIAL CONDITIONS
  # Q[1] is necessary for stepsize-check (if dQ too large)
  o$Q      [1]  = func_Qobs(output_date[2]) / (output_date[2]-output_date[1]) *3600
  # hS from first Q measurement and Qh-relation
  o$hS     [1]  = uniroot(f=function(x){return(
                  func_Q_hS(x,p,hSmin=func_hSmin(output_date[1]))-o$Q[1])},
                  lower=0, upper=p$cD)$root
  # dG and hQ
  if(is.null(p$dG0)==FALSE)                                 # if dG0 provided
  {
    o$dG   [1]  = p$dG0
    if((p$cD-o$dG[1])<o$hS[1])                              # if groundwater below surface water level
    {
      o$hQ  [1] = o$Q[1]*p$cQ                               # all Q from quickflow
    }else{                                                  # if groundwater above surface water level
      o$hQ  [1] = max(0,(o$Q[1]-(p$cD-o$dG[1]-o$hS[1])*(p$cD-o$dG[1])/p$cG) *p$cQ)
    }
  }else{                                                # if dG0 not provided
    if(is.null(p$Gfrac)==TRUE){p$Gfrac=1}               # if Gfrac also not provided, make Gfrac 1
    # if fGS not possible with current hS and cG, make Gfrac smaller
    while(((p$cD-o$hS[1])*p$cD/p$cG) < (p$Gfrac*o$Q[1])) {p$Gfrac = p$Gfrac/2}
    # compute dG leading to the right fGS
    o$dG   [1]  = uniroot(f=function(x){return((p$cD-x-o$hS[1])*(p$cD-x)/p$cG - o$Q[1]*p$Gfrac)},
                          lower=1, upper=(p$cD-o$hS[1]))$root
    o$hQ   [1]  = o$Q[1] *(1-p$Gfrac) *p$cQ
  }
  # dependent variables
  o$dVeq [1]  = func_dVeq_dG(o$dG[1], p)
  o$dV   [1]  = o$dVeq[1]
  o$W     [1]  = func_W_dV(o$dV[1], p)

  #
  o_step       = o[1,]
  i            = o[1,]

  # RUN FOR-LOOP OVER ALL TIME STEPS
  for (t in 2:L)
  {
    start_step     = output_date[t-1]                     # start at begin of output step
    end_step       = output_date[t]                       # first try whole output step
    sums_step      = rep(0,4)                             # to sum fluxes of substeps
    # as long as you're not at the end of the original time_step yet
    while(start_step < (output_date[t] - p_num$min_timestep))
    {
      o_step[1,]   = WALRUS_step(p=p, i=i, t1=start_step, t2=end_step)
      # if time step too large (and not very small)
      if((o_step$dt_ok == FALSE) & ((end_step-start_step) > p_num$min_timestep))
      {
        end_step   = (start_step + end_step)/2          # decrease step and run model
      }else{                                              # if one step completed (dt small enough)
        start_step = end_step                             # start of next step
        end_step   = output_date[t]                       # try to the end of the step
        sums_step  = sums_step + o_step[1:4]              # remember sums of fluxes
        i          = o_step                               # initial conditions for next step
      }
    }
    # final output of the step
    o[t,   ] = o_step                                     # keep states of last step
    o[t,1:4] = sums_step                                  # replace fluxes with sums of steps
  }

  # remove dt_ok column
  o = o[,1:10]

  return(o)
  } # end function

# compile to decrease runtime
WALRUS_loop  = cmpfun(WALRUS_loop)
```

3 Script 2: one time step

```
WALRUS_step = function(p, i, t1, t2)
{
  ### FORCING
  # convert input to current stepsize [mm/timestep]
   P_t     = func_P     (t2)  - func_P     (t1)
   ETpot_t = func_ETpot(t2)  - func_ETpot(t1)
   fXG_t   = func_fXG   (t2)  - func_fXG   (t1)
   fXS_t   = func_fXS   (t2)  - func_fXS   (t1)
   hSmin_t = (func_hSmin(t2) + func_hSmin(t1))/2

  ### STEPSIZE
   dt      = (t2 - t1)/3600              # compute dt (in hours because parameters are in hours)
   dt_ok   = TRUE                        # stepsize small enough as default

  ### FLUXES (based on states from the start of this timestep [mm/timestep])
  PQ       = P_t * i$W                                        *p$aG
  PV       = P_t * (1-i$W)                                    *p$aG
  PS       = P_t                                              *p$aS
  ETV      = ETpot_t * func_beta_dV(i$dV)                     *p$aG
  ETS      = ETpot_t                                          *p$aS
   if(i$hS < p_num$min_h*1000){ETS = 0}        # no ET from empty channel
   ETact   = ETV + ETS
  fQS      = i$hQ                                             /p$cQ *dt
  fGS      = (p$cD - i$dG - i$hS) * max((p$cD - i$dG),i$hS) /p$cG *dt
  Q        = func_Q_hS(i$hS, p=p, hSmin=hSmin_t)                    *dt

  ### STATES (at the end of this time step / start of next time step) [mm])
  # note that fluxes are already for the whole time step (multiplied with dt)
  dV       = i$dV  - (fXG_t + PV - ETV - fGS           )      /p$aG
  hQ       = i$hQ  + (PQ - fQS                         )      /p$aG
  hS       = i$hS  + (fXS_t + PS - ETS + fGS + fQS - Q)      /p$aS
  dG       = i$dG  + (i$dV - i$dVeq) /p$cV *dt



  ### SPECIAL CASE: LARGE-SCALE PONDING AND FLOODING
   if((dV < 0) | (hS > p$cD))
   {
     if((dV < 0) & (hS <= p$cD))                  # if ponding and no flooding
     {
       hS  = hS + (-dV) *p$aG /p$aS               # all ponds to surface water
       dV = 0                                     # soil moisture deficit to surface
     }
     if((dV >= 0) & (hS > p$cD))                  # if no ponding and flooding
     {
       dV  = dV - (hS-p$cD) *p$aS /p$aG           # all floods into soil
       hS  = p$cD                                 # channel bankfull
     }
     if((dV <= 0) & (hS >= p$cD))                 # if ponding and flooding
     {
     dV  = dV*p$aG - (hS-p$cD)*p$aS               # compute total excess water
     hS  = p$cD - dV
```

```
    }
    if(dV < 0){dG = dV}                          # if ponding, groundwater to pond level
  }

  ### TEST IF STEP SIZE IS SMALL ENOUGH
  if(hS < -p_num$min_h)                          # if hS below channel bottom
  {
    dt_ok = FALSE
    hS = p_num$min_h*100
  }else if(hQ < -p_num$min_h)                    # if hQ below bottom Q-res.
  {
    dt_ok = FALSE
    hQ = p_num$min_h
  }else if(P_t > p_num$max_P_step)               # if too much rainfall added
  {
    dt_ok = FALSE
  }else if(abs(i$Q-Q) > p_num$max_dQ_step)       # if change in Q too big
  {
    dt_ok = FALSE
  }else if(abs(i$hS-hS) > p_num$max_h_change)    # if change in hS too big
  {
    dt_ok = FALSE
  }else if(abs(i$dG-dG) > p_num$max_h_change)    # if change in dG too big
  {
    dt_ok = FALSE
  }

  ### OUTPUT
  # compute dependent variables (at end of time step)
  W      = func_W_dV(dV,p)
  dVeq   = func_dVeq_dG(dG,p)

  # bind output together in a vector
  return(c(ETact, Q, fGS, fQS, dV, dVeq, dG, hS, hQ, W, dt_ok))

} # end function

# compile to decrease runtime
WALRUS_step  = cmpfun(WALRUS_step)
```