```
# -*- coding: cp1252 -*-
from pcraster import *
from pcraster.framework import *
from pcraster.collection import *
import time
from scipy import optimize
import math

class TACD2(DynamicModel):
  """Initialization of variables, in order to know what to complete"""
  Lat = None
  scalPar = None
  parameter = None
  nSubCatch = None # map of subcatchments
  DEM = None # digital elevation model (m a.s.l)
  LDD = None # LocalDrainDirection network, modified at the valley bottom
  LDDstream = None
  # Land classification
  nRGType = None # zones with different runoff generation, RGType under urban areas defined
  bUrban = None # cells with settlements
  bForest = None # forest
  # maps for leaf area index and vegetation coverage for each month
  cLai_jan, cLai_feb, cLai_mar, cLai_apr, cLai_may = None, None, \
                                                     None, None, None
  cLai_jun, cLai_jul, cLai_aug, cLai_sep, cLai_oct = None, None, \
                                                     None, None, None
  cLai_nov, cLai_dec = None, None
  cCov_jan, cCov_feb, cCov_mar, cCov_apr, cCov_may = None, None, \
                                                     None, None, None
  cCov_jun, cCov_jul, cCov_aug, cCov_sep, cCov_oct = None, None, \
                                                     None, None, None
  cCov_nov, cCov_dec = None, None
  bStream, sStreamWidth =None, None           # stream zones and width (m)
  sStreamLength   = None #is a factor of 1.0425 * pixel size
  cN              = None # Manning's roughness parameter (-)
  msPrec, msTemp = None, None
  msPotET           = None
  nRunoffStations = None # runoff gauging stations at outlet and subcatchment outlets
  nOut = None # additional subsurface groudwater outfow across the catchment boudary
  nWells = None
  nEndPit = None # catchment outlet of surface water (stream channel)
  sIniSoilMoisture = None # initial soil moisture (mm)
  sIniMTD_box      = None # initial content of MTD (mm)
  sIniUS_box       = None # initial content of upper storage (mm)
  sIniLS_box       = None # initial content of lower storage (mm)
  sIniGW_box       = None # initial content of groundwater storage (mm)
  sIntPrecIni      = None # initial content of interception storage (mm)
  # Output maps from Initialization run
  sIniSoilMoisture_new = None #"..\\OUT\\sm_ini.map"
```

```
50      sIniMTD_box_new      = None #"..\\OUT\\MTD_ini.map"
51      sIniUS_box_new       = None #"..\\OUT\\US_ini.map"
52      sIniLS_box_new       = None #"..\\OUT\\LS_ini_mod.map"
53      sIniGW_box_new       = None #"..\\OUT\\GW_ini_new.map"
54      sIntPrecIni_new      = None #"..\\OUT\\IntPrecIni_new.map"
55      sQmap                = None #"..\\OUT\\Q_debug.map" # debugging
56      sQmap3               = None #"..\\OUT\\Q_debug3.map" # debugging
57      sDmap                = None #"..\\OUT\\d_debug4.map" # debugging
58      SMmaps               = None #"..\\OUT\\SMmaps1.map" # debugging
59      SMtoQ                = None #"..\\OUT\\SMtoQ1.map" # debugging
60      # Climatic data
61      sCumPrec = None #"..\\OUT\\CumRain.map"           # cumulative precipitation over
    evaluation period
62      sMeanTemp = None #"..\\OUT\\MeanTemp.map"         # Map of mean temperature over
    evaluation period (°C)
63      sCumInterceptET = None #"..\\OUT\\CumInterceptET.map" # cumulated evaporation from
    interception storage
64      sCumActET = None #"..\\OUT\\CumActET.map"         # cumulative actual
    evapotranspiration (mm)
65      sCumToGW = None #"..\\OUT\\CumToGW.map"     # cumulated percolation to groundwater
    (mm/h)
66      sMeanVelocity   = None #"..\\OUT\\MeanVelo.map"       # mean water velocity during
    evaluation period (m/s)
67      IntoStream     = None #"..\\OUT\\IntoStream.map"      # map of stream input
68
69      def __init__(self, cloneMap, myinit = True):
70        DynamicModel.__init__(self)
71        setclone(cloneMap)
72        self.clone = readmap(cloneMap)
73        self.isInit = myinit
74
75      def initial(self):
76        #Latitude and amount of sun hours per day
77        self.Lat  = 4.4
78        self.SunHours = self.readmap("..\\MAPS\\SunHours")
79        # parameter for modification of temperature (SCHULLA Manual WASIM-ETH: usually less
    than 5°C) [°C]
80        self.scalPar = 3
81        self.fk = 1.0         # fk: Coast factor. set 1.0 for non-coastal basins
82        self.parameter = dict()
83        with open("..\\INI\\para_ini.tbl", "r") as f:
84          for line in f.readlines():
85            data=[x for x in line.strip().split()]
86            self.parameter[data[1]]=float(data[2])
87        ## OCIN: Index & VariableCollection not working properly in PCRaster V 4.1.0
88        ##       I had to implement it myself
89        """# Input of Parameters from Parameter table
90        self.aParameter = Index(["pStartYear", "pStartDay", "pStartPeriod", "pEndPeriod",\
91          "pTT_urban", "pTT", "pSFCF", "pCFMAX_urban", "pCFMAX",\
92          "pTT_melt", "pTT_melt_forest", "pTT_melt_urban", "pCWH", "pCFR",\
93          "pUrbanSplit", "pLP", "pFC1", "pFC2", "pFC3", "pFC4", "pFC5", "pFC6",\
94          "pBETA1", "pBETA2", "pBETA3", "pBETA4", "pBETA5", "pBETA6",\
95          "pAll_P", "pMTD", "pMTD_K", "pDH_K", "pDI_K_u", "pDI_K_l",\
96          "pDI_T", "pDI_H", "pFI_K_u", "pFI_K_l", "pFI_H", "pFI_T",\
```

```
97          "pFLI_K_u", "pFLI_K_l", "pFLI_H", "pFLI_T", "pEDI_K",\
98          "pDV_K_u", "pDV_K_l", "pDV_H", "pDV_T", "pGW_K", "pUS_H", "pGW_H", "pThres",\
99          "p_Exf", "p_Inf", "pPump_1", "pPump_2", "pPump_3", "pPump_4",\
100         "pPump_5", "pPump_6", "pPump_7", "pPump_8", "pPump_9", "pPump_10", "pPump_11",\
101         "pPump_12", "pPump_13", "pPump_14", "pPump_15", "pPump_16",\
102         "pBeta", "pTimeStep", "pQIni", "pWaterDepthIni",\
103         "pSnowETSep", "pSnowETOct", "pSnowETNovJan", "pSnowETDec", "pSnowETFeb",\
104         "pSnowETMarAprMay"])
105       # print(self.aParameter)
106       # substitution, argument has to be called with modell
107       self.parameter = VariableCollection([self.aParameter],
      value=ValueFromParameterTable("parameter", "para.tbl", dataType=Scalar))
108       """
109       self.nSubCatch = self.readmap("..\\MAPS\\clone")          # map of subcatchments
110       self.DEM = self.readmap("..\\MAPS\\dem")                  # digital elevation model
      (m a.s.l)
111       self.LDD             = self.readmap("..\\MAPS\\ldd01")    # LocalDrainDirection
      network, modified at the valley bottom
112       self.LDDstream       = self.readmap("..\\MAPS\\ldd01")    # original
      LocalDrainDirection network
113
114       # Land classification
115       self.nRGType         = self.readmap("..\\MAPS\\RG")       # zones with different
      runoff generation, RGType under urban areas defined
116       self.bUrban          = self.readmap("..\\MAPS\\urban")    # cells with settlements
117       self.bForest         = self.readmap("..\\MAPS\\forest")   # forest
118
119       # maps for leaf area index and vegetation coverage for each month
120       self.cLai_jan        = self.readmap("..\\MAPS\\LAI\\LAI01")
121       self.cLai_feb        = self.readmap("..\\MAPS\\LAI\\LAI02")
122       self.cLai_mar        = self.readmap("..\\MAPS\\LAI\\LAI03")
123       self.cLai_apr        = self.readmap("..\\MAPS\\LAI\\LAI04")
124       self.cLai_may        = self.readmap("..\\MAPS\\LAI\\LAI05")
125       self.cLai_jun        = self.readmap("..\\MAPS\\LAI\\LAI06")
126       self.cLai_jul        = self.readmap("..\\MAPS\\LAI\\LAI07")
127       self.cLai_aug        = self.readmap("..\\MAPS\\LAI\\LAI08")
128       self.cLai_sep        = self.readmap("..\\MAPS\\LAI\\LAI09")
129       self.cLai_oct        = self.readmap("..\\MAPS\\LAI\\LAI10")
130       self.cLai_nov        = self.readmap("..\\MAPS\\LAI\\LAI11")
131       self.cLai_dec        = self.readmap("..\\MAPS\\LAI\\LAI12")
132
133       self.cCov_jan        = self.readmap("..\\MAPS\\COV\\COV01")
134       self.cCov_feb        = self.readmap("..\\MAPS\\COV\\COV02")
135       self.cCov_mar        = self.readmap("..\\MAPS\\COV\\COV03")
136       self.cCov_apr        = self.readmap("..\\MAPS\\COV\\COV04")
137       self.cCov_may        = self.readmap("..\\MAPS\\COV\\COV05")
138       self.cCov_jun        = self.readmap("..\\MAPS\\COV\\COV06")
139       self.cCov_jul        = self.readmap("..\\MAPS\\COV\\COV07")
140       self.cCov_aug        = self.readmap("..\\MAPS\\COV\\COV08")
141       self.cCov_sep        = self.readmap("..\\MAPS\\COV\\COV09")
142       self.cCov_oct        = self.readmap("..\\MAPS\\COV\\COV10")
143       self.cCov_nov        = self.readmap("..\\MAPS\\COV\\COV11")
144       self.cCov_dec        = self.readmap("..\\MAPS\\COV\\COV12")
145
```

```
146     # Streams
147     self.bStream        = self.readmap("..\\MAPS\\Streams")     # stream network (streams=1/ rest=0)
148     self.sStreamWidth   = self.readmap("..\\MAPS\\bWidth")      # stream width (m)
149
150     #sStreamLength  = scalar(0.00029);                          # average stream length through cell (m)?? what is the stream length
151     self.sStreamLength  = pcraster.celllength() * 1.0425        # is a factor of 1.0425 * pixel size of 0.00027777777778-> 32.1856
152     self.cN             = self.readmap("..\\MAPS\\Manning")     # Manning's roughness parameter (-)
153
154     # Climatic data
155     ## OCIN: this data will have a basename width of 8
156     self.msPrec            = "..\\MAPS\\P\\P000"
157     self.msTemp           = "..\\MAPS\\T\\T2M0"
158     self.msPotET           = "..\\MAPS\\E\\E000"
159     # Gauging stations or control points
160     self.nRunoffStations = self.readmap("..\\MAPS\\RunoffSt")   # runoff gauging stations at outlet and subcatchment outlets
161     self.nOut            = self.readmap("..\\MAPS\\nOut")       # additional subsurface groudwater outfow across the catchment boudary
162     #nWells          = self.readmap("..\\gisdaten\\well200")    # cells with pumping stations of groundwater
163     self.nWells           = nominal(1);
164     self.nEndPit         = self.readmap("..\\MAPS\\EndPit")     # catchment outlet of surface water (stream channel)
165     #tsQ_gauged    = self.readmap("..\climatic_data\runoff_cal.tss"                  # Gauged runoff for calculating the model efficiency(m³/s)
166     # Inititial values for soil and storages (maps are produced by initialization run)
167     self.sIniSoilMoisture = self.readmap("..\\INI\\SM_ini")     # initial soil moisture (mm)
168     self.sIniMTD_box      = self.readmap("..\\INI\\MTD_ini")    # initial content of MTD (mm)
169     self.sIniUS_box       = self.readmap("..\\INI\\US_ini")     # initial content of upper storage (mm)
170     self.sIniLS_box       = self.readmap("..\\INI\\LS_ini")     # initial content of lower storage (mm)
171     self.sIniGW_box       = self.readmap("..\\INI\\GW_ini")     # initial content of groundwater storage (mm)
172     self.sIntPrecIni      = self.readmap("..\\INI\\IntPrec_ini")# initial content of interception storage (mm)
173
174     #.........................
175     # OUTPUT-Files from MODEL
176     #.........................
177     #tsIntoStream_konz      = self.readmap("..\\OUT\\trans\IntoStream_konz.tss"  # concentration of stream input
178     #tsIntoStream_O18   = self.readmap("..\\OUT\\trans\IntoStream_O18.tss"
179     self.tsIntPrec         = "..\\OUT\\IntPrec.tss"              # timeserie of intercepted water
180
181     # Output maps from Initialization run
182     self.sIniSoilMoisture_new = "..\\OUT\\INI\\SM_ini.map"
```

```
183         self.sIniMTD_box_new      = "..\\OUT\\INI\\MTD_ini.map"
184         self.sIniUS_box_new       = "..\\OUT\\INI\\US_ini.map"
185         self.sIniLS_box_new       = "..\\OUT\\INI\\LS_ini.map"
186         self.sIniGW_box_new       = "..\\OUT\\INI\\GW_ini.map"
187         self.sIntPrecIni_new      = "..\\OUT\\INI\\IntPrec_ini.map"
188         self.sQmap                = "..\\OUT\\INI\\Q_debug.map" # debugging
189         self.sQmap3               = "..\\OUT\\INI\\Q_debug3.map" # debugging
190         self.sDmap                = "..\\OUT\\INI\\d_debug4.map" # debugging
191         self.SMmaps               = "..\\OUT\\INI\\SMmaps1.map" # debugging
192         self.SMtoQ                = "..\\OUT\\INI\\SMtoQ1.map" # debugging
193         # Climatic data
194         self.tsMeanPrec             = "..\\OUT\\MeanRain.tss"      # mean daily
    precipitation (mm/d)
195         self.tsPrecSubCatch           = "..\\OUT\\PrecCatch.tss"  # mean daily
    precipitation in the subcatchments (mm/d)
196         self.sCumPrec           = "..\\OUT\\CumRain.map"          # cumulative
    precipitation over evaluation period
197
198         self.tsMeanTemp             = "..\\OUT\\MeanTemp.tss"      # mean hourly
    temperature (°C)
199         self.sMeanTemp              = "..\\OUT\\MeanTemp.map"      # Map of mean
    temperature over evaluation period (°C)
200
201         self.tsPotET                = "..\\OUT\\potET.tss"              # mean daily
    potential evapotranspiration (mm/d)
202         self.tsPotETSubCatch          = "..\\OUT\\potETCatch.tss"      # mean daily
    potential evapotranspiration in the subcatchments(mm/d)
203         self.tsInterceptET          = "..\\OUT\\InterceptET.tss"       # mean daily
    interception (mm/d)
204         self.tsInterceptETCatch     = "..\\OUT\\InterceptETCatch.tss"   # mean daily
    interception in the subcatchments (mm/d)
205
206         self.sCumInterceptET        = "..\\OUT\\CumInterceptET.map"     # cumulated
    evaporation from interception storage
207         self.tsActET                = "..\\OUT\\actET.tss"              # mean daily
    evapotranspiration (mm/d)
208         self.tsActETSubCatch          = "..\\OUT\\actETCatch.tss"       # mean daily
    evapotranspiration in the subcatchments (mm/d)
209         self.sCumActET          = "..\\OUT\\CumActET.map"              # cumulative
    actual evapotranspiration (mm)
210         self.tsSnowPack           = "..\\OUT\\snowpack.tss"
                                        # mean w.e. of snow pack (mm)
211
212         # Direct runoff
213         self.tsQ_DirectStream       = "..\\OUT\\Q_direct.tss"         # direct runoff into
    streams (mm/d)
214         self.tsQ_DirectUrban        = "..\\OUT\\Q_urban.tss"          # direct urban runoff
    (mm/d)
215
216         # SOF-zone
217         self.tsQ_SOF            = "..\\OUT\\Q_SOF.tss"            # saturated overlandflow
    (mm/d)
218         self.tsMTD_box      = "..\\OUT\\MTD_box.tss"          # micro-topographic
    depression storage (mm)
```

```
219         self.tsSOF_IntoStream   = "..\\OUT\\SOF_IntoStream.tss"   # stream input from
    MTD_storages in SOF-areas (mm/d)
220
221         # Soil routine
222         self.tsSoilMoisture     = "..\\OUT\\SoilMoisture.tss"     # soil moisture content
    of all zones (-)
223         #self.tsToRunoffGeneration  = "..\\OUT\\ToRG.tss"            # water flow to
    runoff generation (mm/d)
224
225         # Runoff generation routine
226         self.tsQ_US           =  "..\\OUT\\Q_US.tss"                # runoff out off
    upper storages (mm/d)
227         self.tsUS_box   = "..\\OUT\\US_box.tss"                     # content of upper
    storage (mm)
228         self.tsUS_IntoStream  = "..\\OUT\\US_IntoStream.tss"        # stream input from
    upper storages (mm/d)
229
230         self.tsQ_LS           = "..\\OUT\\Q_LS.tss"                 # runoff out off
    lower storages (mm/d)
231         self.tsLS_box   = "..\\OUT\\LS_box.tss"                     # content of lower
    storage (mm)
232         self.tsLS_IntoStream  = "..\\OUT\\LS_IntoStream.tss"        # stream input from
    lower storages (mm/d)
233
234         self.tsQ_       = "..\\OUT\\Q_all.tss"                     # lateral flow out of
    all storages(US, LS, SOF)(mm/d)
235         self.tsAll_box  = "..\\OUT\\all_box.tss"                    # content of all
    storages (mm)
236
237         # Groundwater
238         self.sCumToGW   = "..\\OUT\\CumToGW.map"                    # cumulated
    percolation to groundwater (mm/d)
239         self.tsToGW     = "..\\OUT\\ToGW.tss"                       # mean percolation to
    groundwater (mm/d)
240         self.tsQ_GW           = "..\\OUT\\Q_GW.tss"                 #lateral outflow out
    of groundwater storage (mm/d)
241         self.tsGW_box   = "..\\OUT\\GW_box.tss"                     # content of
    groundwater storage (mm)
242
243         self.tsPump     = "..\\OUT\\Pump.tss"                       # content of water in
    lower storage at pumping stations
244         self.tsPumpRate     = "..\\OUT\\PumpRate.tss"               # pump rate
245
246         # Flow into streams and channel routing
247         self.tsQ_Exf         = "..\\OUT\\Exf.tss"                   # exfiltration out of
    groundwater into stream (mm/h)
248         self.tsQ_Inf         = "..\\OUT\\Inf.tss"                   # infiltration out of
    stream into groundwater (mm/h)
249         self.tsIntoStream    = "..\\OUT\\IntoStream.tss"            # stream input (mm/h)
250         self.tsTotIntoStream = "..\\OUT\\TotIntoStream"             # total steam input
    in catchment (mm/h)
251         self.tsGW_IntoStream = "..\\OUT\\GW_IntoStream.tss"         # stream input from
    groundwater (mm/h)
252         self.tsQ_IntoStream  = "..\\OUT\\Q_IntoStream.tss"
```

```
253         self.tsWaterDepth    = "..\\OUT\\Waterdepth.tss"              # mean water depth at
    certain points (m)
254         self.tsVelocity      = "..\\OUT\\Velocity.tss"                # water velocity at
    certain points(m/s)
255         self.sMeanVelocity   = "..\\OUT\\MeanVelo.map"                # mean water velocity
    during evaluation period (m/s)
256         self.IntoStream    = "..\\OUT\\IntoStream.map"                # map of stream input
257
258         # Runoff at outlet of (sub)catchment(s)
259         self.tsQ_sim     = "..\\OUT\\Q_sim.tss"                       # runoff at gauging
    stations (m³/s)
260         self.tsQ_out          = "..\\OUT\\runoff_out.tss"             # underground outflow
    from the catchment (mm/h)
261
262         # Budget check
263         self.tsBalance        = "..\\OUT\\Check.tss"                   # water balance check
264         self.tsBalanceRouting = "..\\OUT\\CheckRouting.tss"           # water balance check
    for routing-routine
265         self.tsRoutingPercent = "..\\OUT\\Routing_P.tss"              # percetage of
    balancing error for balancing routine
266         self.tsStartStorage = "..\\OUT\\StartStorage.tss"             # water in storages
    at begin of simulation
267         self.tsEndStorage = "..\\OUT\\EndStorage.tss"                 # water in storages
    at end of simulation
268         self.tsEndPrec = "..\\OUT\\EndPrec.tss"                       # cumulated
    precipitation input
269         self.tsEndInterceptET = "..\\OUT\\EndInterceptET.tss"         # cumulated
    evaporation out of interception routine
270         self.tsEndActualET = "..\\OUT\\EndActualET.tss"               # cumulated
    evapotranspiration out of all other routines
271         self.tsEndIntoStream = "..\\OUT\\EndIntoStream.tss"           # cumulated total
    stream input
272         self.tsEndPumpRate = "..\\OUT\\EndPumpRate.tss"               # cumulated
    withdrawal by poumping
273         self.tsEndQ_out = "..\\OUT\\EndQ_out.tss"                     # cumulated
    subsurface outfow
274
275         # Evaluation
276         self.sMeanQ_sim = "..\\OUT\\MeanQ_sim.map"                    # map of mean
    simulated runoff (m³/s)
277         self.sMeanRunoff = "..\\OUT\\MeanRunoff.map"                  # map of mean
    measured runoff (m³/s)
278         self.slogMeanRunoff = "..\\OUT\\logMeanRunoff.map"            # map of logarithmic
    mean measured runoff (m³/s)
279
280         #.........................
281         # MISCELLANEOUS
282         #.........................
283
284         # Array for loop construction of kinematic function
285         self.aHour = [i for i in range(1)]
286         # Number of timesteps using for kinematic function
287         self.cNrSteps = len(self.aHour)
288
```

```
289         ## Here starts the "initial" routine of the original TAC_D
290         #.........................
291         # ASSIGNMENT of PARAMETERS
292         #.........................
293         # General model settings
294         self.cStartDay     = self.parameter["pStartDay"]      # Start day (Julian date)
295         self.cStartPeriod  = self.parameter["pStartPeriod"]   # timestep marks beginn of
    period for which reports are made; start of evaluation period
296         self.cEndPeriod    = self.parameter["pEndPeriod"]     # timestep marks end of
    period (usally equals last timestep); end of evaluation period
297
298         # Snow parameters
299         self.cTT = ifthenelse(self.bUrban, scalar(self.parameter["pTT_urban"]), scalar(self.
    parameter["pTT"]))    # freezing temperature (°C) for snow fall
300         self.cSFCF = self.parameter["pSFCF"]    # snowfall correction factor
301         # threshold temperature (°C) for snow melting (degree-hour-methode)
302         self.cTT_melt = ifthenelse(self.bForest,\
303                               scalar(self.parameter["pTT_melt_forest"]),\
304                               ifthenelse(self.bUrban, scalar(self.parameter["pTT_melt_urban"
    ]),\
305                                                       scalar(self.parameter["pTT_melt"])))
306         self.cCFMAX = ifthenelse(self.bUrban, scalar(self.parameter["pCFMAX_urban"]),\
307                                         scalar(self.parameter["pCFMAX"]))        # degree
    hour factor (mm/°C*h) this is hour do i need to change
308         self.cCWH = self.parameter["pCWH"]                # coefficient of water holding
    capacity
309         self.cCFR = self.parameter["pCFR"]                # refreezing coefficent
310
311         # Urban parameter
312         self.cUrbanSplit = scalar(self.parameter["pUrbanSplit"])  # fraction of sealed area
    vgl. DYCK/PESCHKE (1995)
313
314         # Soil parameters
315         self.cLP = scalar(self.parameter["pLP"])    # reduction parameter for actual
    evapotranspiration, parameter LP for all zones[-]
316
317         # field capacity for each zone (m)
318         ## self.cFieldCapacity = ifthen(self.nRGType < 7,
    scalar(self.parameter["pFC"+self.nRGType]))
319         self.cFieldCapacity = ifthenelse(self.nRGType == 1, self.parameter["pFC1"],\
320                                  ifthenelse(self.nRGType == 2, self.parameter["pFC2"],\
321                                  ifthenelse(self.nRGType == 3, self.parameter["pFC3"],\
322                                  ifthenelse(self.nRGType == 4, self.parameter["pFC4"],\
323                                  ifthenelse(self.nRGType == 5, self.parameter["pFC5"],\
324                                  ifthen(self.nRGType == 6, scalar(self.parameter["pFC6"])))))))
325         # parameter BETA for each zone (m)
326         self.cBETA = ifthenelse(self.nRGType == 1, self.parameter["pBETA1"],\
327                       ifthenelse(self.nRGType == 2, self.parameter["pBETA2"],\
328                       ifthenelse(self.nRGType == 3, self.parameter["pBETA3"],\
329                       ifthenelse(self.nRGType == 4, self.parameter["pBETA4"],\
330                       ifthenelse(self.nRGType == 5, self.parameter["pBETA5"],\
331                       ifthen(self.nRGType == 6, scalar(self.parameter["pBETA6"])))))))
332
333         # Runoff generation parameters
```

```
334     self.cAll_P = ifthenelse((self.nRGType == 7) | (self.nRGType == 6), scalar(0), self.
    parameter["pAll_P"])# percolation to deeper groundwater (mm/h) in  all RG-zones  ??hour
    change to day except valley bottom and saturated overland flow
335     self.cMTD = self.parameter["pMTD"]           # max. height of "micro-topographic
    depression" (m)
336     self.cMTD_K = self.parameter["pMTD_K"]       # storage coefficient (h-1)
337     self.cDH_K = self.parameter["pDH_K"]         # storage coefficient (h-1) deep
    percolation in high areas
338     self.cDI_K_u = self.parameter["pDI_K_u"]     # upper storage coefficient (h-1)
    "delayed Interflow"
339     self.cDI_K_l = self.parameter["pDI_K_l"]     # lower storage coefficient (h-1)
340     self.cDI_H = self.parameter["pDI_H"]         # limit of lower storage (mm)
341     self.cDI_T = self.parameter["pDI_T"]         # percolation to lower storage (mm/h)
    hour to day ??
342     self.cFI_K_u = self.parameter["pFI_K_u"]     # upper storage coefficient (h-1)
    "fast Interflow"
343     self.cFI_K_l = self.parameter["pFI_K_l"]     # lower storage coefficient (h-1)
344     self.cFI_H = self.parameter["pFI_H"]         # limit of lower storage (mm)
345     self.cFI_T = self.parameter["pFI_T"]         # percolation to lower storage (mm/h)
    hour to day??
346     self.cFLI_K_u = self.parameter["pFLI_K_u"]   # upper storage coefficient (h-1)
    "fast, lateral Interflow, piston flow"
347     self.cFLI_K_l = self.parameter["pFLI_K_l"]   # lower storage coefficient (h-1)
348     self.cFLI_H = self.parameter["pFLI_H"]       # limit of lower storage (mm)
349     self.cFLI_T = self.parameter["pFLI_T"]       # percolation to lower storage (mm/h)
    hour to day ????
350     self.cEDI_K = self.parameter["pEDI_K"]       # storage coefficient (h-1) "extrem
    delayed Interflow"
351     self.cDV_K_u = self.parameter["pDV_K_u"]     # storage coefficient (h-1) upper
    storage in the valley bottom
352     self.cDV_K_l = self.parameter["pDV_K_l"]     # storage coefficient lower storage
353     self.cDV_H   = self.parameter["pDV_H"]       # limit of lower storage
354     self.cDV_T   = self.parameter["pDV_T"]       # percolation to lower storage
355     # limit of upper storage for all RG-Types (mm)
356     self.cUS_H = ifthenelse(self.nRGType == 6, self.parameter["pDV_H"],\
357                     ifthenelse(self.nRGType == 7, scalar(0), self.parameter["pUS_H"]))
358     # limit for groundwater storage (mm)
359     self.cGW_H = ifthen(self.nRGType != 6, scalar(self.parameter["pGW_H"]))
360
361     # Groundwater parameters  # storage coefficient (h-1)
362     self.cGW_K = ifthenelse(self.nRGType != 6, self.parameter["pGW_K"], scalar(0))
363
364     # Parameter for interaction stream-groundwater
365     self.cThres = self.parameter["pThres"]      # threshold limit of upper storage for
    infiltration and exfiltration (m)
366     self.c_Exf = self.parameter["p_Exf"]        # exfiltration rate over the whole
    valley bottom (m/d)
367     self.c_Inf = self.parameter["p_Inf"]        # infiltration rate over the whole
    valley bottom (m/d)
368
369     # groundwater uptake parameters # pump rate for each pump (m³/d)
370     self.cPump = ifthenelse(self.nWells == 1, scalar(self.parameter["pPump_1"]),\
371                   ifthenelse(self.nWells == 2, scalar(self.parameter["pPump_2"]),\
372                   ifthen(self.nWells == 3, scalar(self.parameter["pPump_3"]))))
```

```
373
374         # Channel routing parameters
375         self.cBeta                = self.parameter["pBeta"]           # beta coefficient
    for kinematic wave equation (-)
376         self.sQIni                = self.parameter["pQIni"]           # initial streamflow
    (m³/s)
377         self.sWaterDepthIni        = self.parameter["pWaterDepthIni"] # initial water depth
    (m)
378
379         # Snow evaporation (mm/d)                    # values according to RACHNER (1999)
380         self.cSnowETSep       = self.parameter["pSnowETSep"]
381         self.cSnowETOct       = self.parameter["pSnowETOct"]
382         self.cSnowETNovJan    = self.parameter["pSnowETNovJan"]
383         self.cSnowETDec       = self.parameter["pSnowETDec"]
384         self.cSnowETFeb       = self.parameter["pSnowETFeb"]
385         self.cSnowETMarAprMay = self.parameter["pSnowETMarAprMay"]
386
387         #.................................
388         # General
389         #.................................
390
391         # Julian date
392         self.sDay = self.cStartDay
393         self.sHour = 0
394         self.sHourStep = 24                      # New Hour = Old Hour + HourStep
395         self.sDayStep  = 1                       # New Day = Old Day + DayStep
396         self.sYearStep = 1                       # New Year = Old Year + YearStep
397         self.sStartYear = self.parameter["pStartYear"]  # Year at the beginning of model run
398         #sStartYear = pStartYear
399         self.sYear = self.sStartYear
400
401         # Length of evaluation period (timesteps)
402         self.cLengthPeriod = self.nrTimeSteps()
403
404         # Conversion factor for (mm/h) into (m³/s)
405         #self.cMmToCubM = (0.001 * cellarea())/86400
406         self.cMmToCubM = (0.001 * cellarea()) #daily rainfall in meters, convert first to mm
407         # Conversion factor from (kg/(timestep) into (kg/s)
408         self.ckg_trans = 1.0/86400
409
410         #.........................................................
411         #Constants for conversion from delta 18O to molecules per m³
412         #.........................................................
413         self.Rstand = 0.0020052    # Ratio of 18O/16O in standard (SMOW)
414         self.Density = 0.9997      # Density of water at 10°C
415         self.Avo = 6.022E+23       # Avogadro-number
416
417         #.................................
418         # Calculate additional catchment data
419         #.................................
420
421         # Slope
422         #--------
423         # Calculate slope and mean slope of each RGType zone (dz/dx)
```

```
424         # values between 0 and 1
425         self.sSlope = slope(self.DEM)
426         self.sSlopePerRGT = areaaverage(self.sSlope, pcraster.nominal(self.nRGType))
427 
428         # Calculate slope factor for storage runoff (= (tanß/ average tanß))
429         self.cSlopeFactor = (self.sSlope / self.sSlopePerRGT)
430         # slope factor is limited to 0.3 so that the overflow is not always activated in
    flat areas
431         self.cSlopeFactor = ifthenelse(self.cSlopeFactor < 0.3, 0.3, self.cSlopeFactor)
432         # kinematic function divides by root of slope ( => not 0)
433         self.sSlope = pcraster.max(0.0001, self.sSlope)
434 
435         self.SlopMap = scalar(pcraster.atan(slope(self.DEM)))
436         self.SlopMap = ifthenelse(self.SlopMap == 0, scalar(0.001), self.SlopMap)
437         self.AspMap  = scalar(aspect(self.DEM))                      # aspect [deg]
438         self.AspMap  = ifthenelse(self.AspMap <= 0, scalar(0.001), self.AspMap)
439 
440         # Seepage of streams #not necessary for Dreisam catchment
441         #--------------------
442         # Define LDD for streams
443         self.StreamLDD = ifthen(self.bStream, pcraster.nominal(self.LDDstream))
444         # bStream is the stream order 0 is no channel.
445         # here using LDD stream is the original one of LDD
446         # on the condition of bstream, if the cell value of bstream is true, then the
    result is LDD
447         # if the cell value of bstream is false, then the result is MV
448 
449         #if condition has a cell value 1 (TRUE) the value on expression1 is assigned Result
450         #if condition has a cell value 0 (FALSE) the value on expression2 is assigned to
    Result.
451         # Modify StreamLDD for kinematic function --> ending streams have to be pits (ldd =
    5)
452         self.StreamLDD = lddrepair(pcraster.ldd(self.StreamLDD))
453 
454         # Determine last cells of ending streams
455         self.sLastStreamCell = cover(ifthenelse(self.StreamLDD == 5, scalar(1), scalar (0)),
     scalar(0))
456         #
457 
458         # Determine cells downstream of last stream cells and assign "true"
459         ## make comments of the sSeepage to avoid using LDD and
460         self.sSeepage = upstream(self.LDD, self.sLastStreamCell)
461         self.bSeepage = self.sSeepage >= 1
462 
463         # Runoff generation type of stream cells
464         self.nRGTypeStream = ifthen(self.bStream, self.nRGType)
465         # Calculate part of RGType of stream cells
466         self.sRGTypeStreamPart = areaarea(pcraster.nominal(self.nRGTypeStream)) / areaarea(
    self.bStream)
467 
468         #.....................................................................
469         # Creation of basic maps with only one zone, rest of area as missing value
470         #.....................................................................
471         self.bSOF_zone    = self.nRGType == 7        # map defined for only "saturated
```

```
overlandflow"
472     self.nSoilType    = ifthenelse(self.nRGType != 7, self.nRGType, 0)  # map with all
zones that go through the soil routine
473     self.sValleyBottom = self.nRGType == 6   # map with valley bottom only
474     self.sStreamValley = ifthenelse(self.nRGType == 6, self.bStream, 0) # map of stream
in the valley bottom
475
476     #.....................................................
477     # Assigning initial values and parameters to variables and defining variables
478     #.....................................................
479
480     # Climatic input
481     self.sCumPrec     = scalar(0)
482     self.sCumTemp     = scalar(0)
483     self.sCumInterceptET = scalar(0)
484     self.sCumActET    = scalar(0)
485
486     # Snow routine
487     self.sSnowPack         = scalar(0)
488     self.sWaterContent      = scalar(0)
489
490     # Urban runoff
491     self.sUrbanSplit = ifthenelse(self.bUrban, self.cUrbanSplit, 0)
492
493     # Runoff generation
494     #-------------------
495     self.sCumQ_SOF = 0              # cumulative saturated overland flow (mm/h)
496     self.sCumQ_US  = 0              # cumulative runoff out off upper storage (mm/h)
497     self.sCumQ_LS  = 0              # cumulative runoff out off lower storage (mm/h)
498     self.sCumQ_out = 0
499     self.sCumToGW  = 0
500     self.sAll_box  = 0
501     self.sLS_box_prePump = 0
502
503     # Assign storage coefficents K to upper storage (-h)
504     ## cDH_K,           # high areas
505     ## cDI_K_u,         # delayed interflow
506     ## cFI_K_u,         # fast interflow
507     ## cFLI_K_u,        # accumulation areas at hill foots, fast lateral interflow,
piston flow
508     ## cEDI_K,          # extrem delayed interflow
509     ## cDV_K_u          # valley bottom
510     self.cUS_K = ifthenelse(self.nRGType == 1, self.cDH_K,\
511                  ifthenelse(self.nRGType == 2, self.cDI_K_u,\
512                  ifthenelse(self.nRGType == 3, self.cFI_K_u,\
513                  ifthenelse(self.nRGType == 4, self.cFLI_K_u,\
514                  ifthenelse(self.nRGType == 5, self.cEDI_K,\
515                  ifthen(self.nRGType == 6, scalar(self.cDV_K_u)))))))
516
517     # Assign percolation parameter (upper --> lower storage)
518     ## cDI_T,       # delayed interflow
519     ## cFI_T,       # fast interflow
520     ## cFLI_T,      # accumulation areas at hill foots, fast, lateral interflow, piston
flow
```

```
521       ## cDV_T        # valley bottom
522       self.cUS_T = ifthenelse(self.nRGType == 2, self.cDI_T,\
523                    ifthenelse(self.nRGType == 3, self.cFI_T,\
524                    ifthenelse(self.nRGType == 4, self.cFLI_T,\
525                    ifthen(self.nRGType == 6, scalar(self.cDV_T)))))
526
527       # Assign storage coefficents K to lower storage (-h)
528       ## cDI_K_l,     # delayed interflow
529       ## cFI_K_l,     # fast interflow
530       ## cFLI_K_l,    # accumulation areas at hill foots, fast, lateral interflow, piston
      flow
531       ## cDV_K_l      # valley bottom
532       self.cLS_K = ifthenelse(self.nRGType == 2, self.cDI_K_l,\
533                    ifthenelse(self.nRGType == 3, self.cFI_K_l,\
534                    ifthenelse(self.nRGType == 4, self.cFLI_K_l,\
535                    ifthen(self.nRGType == 6, scalar(self.cDV_K_l)))))
536
537       # Assign storage limit H to lower storage (mm)
538       ## cDI_H,       # delayed interflow
539       ## cFI_H,       # fast interflow
540       ## cFLI_H,      # accumulation areas at hill foots, fast, lateral interflow, piston
      flow
541       ## cDV_H        # valley bottom
542       self.cLS_H = ifthenelse(self.nRGType == 2, self.cDI_H,\
543                    ifthenelse(self.nRGType == 3, self.cFI_H,\
544                    ifthenelse(self.nRGType == 4, self.cFLI_H,\
545                    ifthen(self.nRGType == 6, scalar(self.cDV_H)))))
546
547       # groundwater uptake
548       self.cPumpRate = (self.cPump * 1000) / cellarea()   # groundwater uptake (mm/h) for
      one cell
549
550       # Groundwater
551       #-------------
552       self.sCumQ_GW         = 0   # Cumulative runoff
553
554       # Channel routing
555       #-------------------
556       # Initial streamFlow (m³/s)
557       self.sQ_step = self.sQIni
558
559       # Initial water depth (m)
560       self.sWaterDepth = self.sWaterDepthIni
561
562       # Term for Alpha
563       self.AlpTerm = (self.cN / (pcraster.sqrt(self.sSlope))) ** self.cBeta
564       # Power for Alpha
565       self.AlpPow        = (2.0/3.0) * self.cBeta
566
567       # Initial approximation for Alpha
568       self.sAlpha = self.AlpTerm * (self.sStreamWidth + (2* self.sWaterDepth))** self.
      AlpPow
569
570       self.sCumIntoStream = 0
```

```
571       self.sCumRunoff     = 0
572       self.sCumVelocity   = scalar(0)
573       self.sCumLeak       = 0
574       self.sCumPumpRate   = 0
575       self.sCumInSoil     = 0
576       self.sCumQ_Inf      = 0
577
578       # Budget check
579       #--------------
580       self.sStartStorage  = 0
581       self.sEndStorage    = 0
582       self.sStartVol      = 0
583       self.sVolumen       = 0
584
585       # Evaluation
586       self.counter        = 0
587       self.sCumQ_gauged = 0
588       self.slogCumQ_gauged = 0
589       self.sCumQ_sim = 0
590
591       #....................................
592       # Initial settings of soil and storages
593       #....................................
594       # NOTE: use these lines for first use; for initialization of realistic storage
      values, repeat initialization run with output maps of preceeding run, until storage
      levels are stable
595       if self.isInit:
596         self.sSoilMoisture = ifthen(defined(self.nSoilType), scalar(0)) # soil moisture
      (mm)
597         self.sMTD_box = ifthen(self.bSOF_zone, scalar(0))              # content of
      "micro-topographic depression" (mm)
598         self.sUS_box  = ifthen(defined(self.cUS_K), scalar(0))  # content of upper
      storage(mm) including valley bottom storage
599         self.sLS_box  = ifthen(defined(self.cLS_K), scalar(0))  # content of lower
      storage (mm)
600         self.sGW_box  = scalar(0)                               # content of groundwater
      storage (mm)
601         self.sIntPrecOld = scalar(0)
602       else:
603         self.sSoilMoisture = self.sIniSoilMoisture  # soil moisture (mm)
604         self.sMTD_box = self.sIniMTD_box            # content of "micro-topographic
      depression" (mm)
605         self.sUS_box  = self.sIniUS_box             # content of upper storage(mm)
      including valley bottom storage
606         self.sLS_box  = self.sIniLS_box             # content of lower storage (mm)
607         self.sGW_box  = self.sIniGW_box             # content of groundwater storage (mm)
608         self.sIntPrecOld = self.sIntPrecIni
609
610       self.sUS_lat = 0
611       self.sLS_lat = 0
612       self.F = 0.9
613
614       """
615       ## OCIN: As a part of the Dynamic() module, it is neccesary to initialise the
```

```
616         ##       Timeoutputseries here. In dynamic() they will get sampled in each
617         ##       timestep
618         """
619
620         self.tsMeanPrec = TimeoutputTimeseries("tsMeanPrec",\
621                                                self, self.nSubCatch, noHeader=False)
622         self.tsMeanTemp = TimeoutputTimeseries("tsMeanTemp",\
623                                                self, 1, noHeader=False)
624         self.tsPotET = TimeoutputTimeseries("tsPotET",\
625                                                self, 1, noHeader=False)
626
627         self.tsSnowPack = TimeoutputTimeseries("tsSnowPack",\
628                                                self, 1, noHeader=False)
629         self.tsWaterContent = TimeoutputTimeseries("tsWaterContent",\
630                                                self, 1, noHeader=False)
631         self.tsMeltWater = TimeoutputTimeseries("tsMeltWater",\
632                                                self, 1, noHeader=False)
633
634         self.tsInterceptET = TimeoutputTimeseries("tsInterceptionET",\
635                                                self, 1, noHeader=False)
636         self.tsIntPrec = TimeoutputTimeseries("tsIntPrec",\
637                                                self, 1, noHeader=False)
638
639         self.tsQ_DirectStream = TimeoutputTimeseries("tsQdirectStream",\
640                                                self, self.nRunoffStations, noHeader=False)
641         self.tsQ_DirectUrban = TimeoutputTimeseries("tsQ_DirectUrban",\
642                                                self, self.nRunoffStations, noHeader=False)
643
644         self.tsSoilMoisture = TimeoutputTimeseries("tsSoilMoisture",\
645                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
646         self.tsToRunoffGeneration = TimeoutputTimeseries("tsToRunoffGeneration",\
647                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
648         self.tsActET = TimeoutputTimeseries("tsActET",\
649                                                self, 1, noHeader=False)
650
651         self.tsMTD_box = TimeoutputTimeseries("tsMTD_box",\
652                                                self, 1, noHeader=False)
653         self.tsUS_box = TimeoutputTimeseries("tsUS_box",\
654                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
655         self.tsLS_box = TimeoutputTimeseries("tsLS_box",\
656                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
657
658         self.tsPumpRate = TimeoutputTimeseries("tsPumpRate",\
659                                                self, self.nWells, noHeader=False)
660         self.tsPump = TimeoutputTimeseries("tsPump",\
661                                                self, self.nWells, noHeader=False)
662
663         self.tsQ_SOF = TimeoutputTimeseries("tsQ_SOF",\
664                                                self, 1, noHeader=False)
665         self.tsQ_US = TimeoutputTimeseries("tsQ_US",\
```

```
                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
        self.tsQ_LS = TimeoutputTimeseries("tsQ_LS",\
                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
        self.tsQ_out = TimeoutputTimeseries("tsQ_out",\
                                                self, 1, noHeader=False)

        self.tsToGW = TimeoutputTimeseries("tsToGW",\
                                                self, pcraster.nominal(self.nSoilType),
    noHeader=False)
        self.tsUS_IntoStream = TimeoutputTimeseries("tsUS_IntoStream",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsLS_IntoStream = TimeoutputTimeseries("tsLS_IntoStream",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsSOF_IntoStream = TimeoutputTimeseries("tsSOF_IntoStream",\
                                                self, self.nRunoffStations, noHeader=False)

        self.tsQ_GW = TimeoutputTimeseries("tsQ_GW",\
                                                self, (~self.bStream), noHeader=False)
        self.tsGW_box = TimeoutputTimeseries("tsGW_box",\
                                                self, (~self.bStream), noHeader=False)

        self.tsIntoStream = TimeoutputTimeseries("tsIntoStream",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsTotIntoStream = TimeoutputTimeseries("tsTotIntoStream",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsGW_IntoStream = TimeoutputTimeseries("tsGW_IntoStream",\
                                                self, self.nRunoffStations, noHeader=False)
        """self.tsQ_IntoStream = TimeoutputTimeseries("tsQ_IntoStream",\ NOT WORKING!!!
                                                self, self.nRGTypeStream, noHeader=False)"""

        self.tsQ_sim = TimeoutputTimeseries("tsQ_sim",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsWaterDepth = TimeoutputTimeseries("tsWaterDepth",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsVelocity = TimeoutputTimeseries("tsVelocity",\
                                                self, self.nRunoffStations, noHeader=False)

        """self.tsAll_box = TimeoutputTimeseries("tsAll_box",\ NOT WORKING!!!
                                                self, self.nRGType, noHeader=False)"""

        self.tsBalance = TimeoutputTimeseries("tsBalance",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsStartStorage = TimeoutputTimeseries("tsStartStorage",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsEndStorage = TimeoutputTimeseries("tsEndStorage",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsEndPrec = TimeoutputTimeseries("tsEndPrec",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsEndInterceptET = TimeoutputTimeseries("tsEndInterceptET",\
                                                self, self.nRunoffStations, noHeader=False)
        self.tsEndActualET = TimeoutputTimeseries("tsEndActualET",\
                                                self, self.nRunoffStations, noHeader=False)
```

```
717       self.tsEndIntoStream = TimeoutputTimeseries("tsEndIntoStream",\
718                                                   self, self.nRunoffStations, noHeader=False)
719       self.tsEndPumpRate = TimeoutputTimeseries("tsEndPumpRate",\
720                                                   self, self.nRunoffStations, noHeader=False)
721       self.tsEndQ_out = TimeoutputTimeseries("tsEndQ_out",\
722                                                   self, self.nRunoffStations, noHeader=False)
723
724       self.tsBalanceRouting = TimeoutputTimeseries("tsBalanceRouting",\
725                                                   self, self.nRunoffStations, noHeader=False)
726       self.tsRoutingPercent = TimeoutputTimeseries("tsRoutingPercent",\
727                                                   self, self.nRunoffStations, noHeader=False)
728
729       #----REPORT-----REPORT-----REPORT------REPORT-----Report------
730       """##
731       ##    # coverage of meteorological stations for the whole area
732       ##    self.rainZones = spreadzone("rainstat.map", scalar(0), scalar(1))
733       ##
734       ##    # create an infiltration capacity map (mm/6 hours), based on the
735       ##    # soil map
736       ##    self.infiltrationCapacity = lookupscalar("infilcap.tbl", "soil")
737       ##    self.report(self.infiltrationCapacity, "infilcap")
738       ##
739       ##    # generate the local drain direction map on basis of the elevation map
740       ##    self.ldd = lddcreate("dem.map", 1e31, 1e31, 1e31, 1e31)
741       ##    self.report(self.ldd, "ldd")
742       ##
743       ##    # initialise timeoutput
744       ##    self.runoffTss = TimeoutputTimeseries("runoff", self, "samples.map",
    noHeader=False)"""
745
746     def dynamic(self):
747       if self.currentTimeStep() % 100 == 0 or self.currentTimeStep() == self.cLengthPeriod:
748         print " ->Step: {0}/{1} ({2}) {3}".format(self.currentTimeStep(), int(self.
    cLengthPeriod),\
749                     int(self.cStartPeriod + self.currentTimeStep()), \
750                     time.strftime("%d/%m %H:%M:%S", time.localtime()))
751       # Calculate Julian date
752       ## OCIN: This module has been deeply changed in its form, preserving logic
753       ## Adapting complicated PCRaster operations to simple Python expressions
754       if self.sHour>=24:
755         self.sDay, self.sHour= self.sDay + self.sDayStep, 1
756       else:
757         self.sDay, self.sHour= self.sDay, self.sHour + self.sHourStep
758
759       ## is it leap year or not?
760       ## OCIN: PS: it is also implemented in "calendar" package, using it like this:
761       ##
762       ## import calendar
763       ## isLeapYear = calendar.isleap(self.sYear)
764       ##
765       ## but I left it like this to be more explicit on how to do it (it was wrong in
    Roser(2001)!!!)
766       isLeapYear = self.sYear % 4 == 0 and (self.sYear % 100 != 0 or\
767                                             self.sYear % 400 == 0)
```

```
768       if isLeapYear and self.sDay>366:
769         self.sYear += self.sYearStep
770         self.sDay = 1
771       elif (not isLeapYear) and self.sDay>365:
772         self.sYear += self.sYearStep
773         self.sDay = 1
774       else:
775         self.sDay += 1
776
777       if isLeapYear:
778         if self.sDay<=31:
779           self.sMon = 1
780         elif self.sDay<=60:
781           self.sMon = 2
782         elif self.sDay<=91:
783           self.sMon = 3
784         elif self.sDay<=121:
785           self.sMon = 4
786         elif self.sDay<=152:
787           self.sMon = 5
788         elif self.sDay<=182:
789           self.sMon = 6
790         elif self.sDay<=213:
791           self.sMon = 7
792         elif self.sDay<=244:
793           self.sMon = 8
794         elif self.sDay<=274:
795           self.sMon = 9
796         elif self.sDay<=305:
797           self.sMon = 10
798         elif self.sDay<=335:
799           self.sMon = 11
800         else:
801           self.sMon = 12
802       else:         ## not a leap year --> common year
803         if self.sDay<=31:
804           self.sMon = 1
805         elif self.sDay<=59:
806           self.sMon = 2
807         elif self.sDay<=90:
808           self.sMon = 3
809         elif self.sDay<=120:
810           self.sMon = 4
811         elif self.sDay<=151:
812           self.sMon = 5
813         elif self.sDay<=181:
814           self.sMon = 6
815         elif self.sDay<=212:
816           self.sMon = 7
817         elif self.sDay<=243:
818           self.sMon = 8
819         elif self.sDay<=273:
820           self.sMon = 9
821         elif self.sDay<=304:
```

```
822           self.sMon = 10
823         elif self.sDay<=334:
824           self.sMon = 11
825         else:
826           self.sMon = 12
827       #*************************
828       # For the budget check
829       #*************************
830
831       # Initial storage
832       # = snow (pack + water content) + soil moisture + all RG-storages
833       #   + groundwater storage + interception storage
834       if self.currentTimeStep() == self.cStartPeriod:
835         self.sStartStorage = catchmenttotal(self.sSnowPack, self.LDDstream) +\
836                              catchmenttotal(self.sWaterContent, self.LDDstream) +\
837           catchmenttotal(ifthen(defined(self.clone),\
838                                 cover(self.sSoilMoisture, 0)), self.LDDstream)+\
839           catchmenttotal(self.sAll_box, self.LDDstream) +\
840           catchmenttotal(ifthen(defined(self.clone),\
841                                   cover(self.sGW_box, 0)), self.LDDstream) +\
842           catchmenttotal(self.sIntPrecOld, self.LDDstream)
843
844       ## NOTE: This works for [1,9999999]
845       a = int(self.currentTimeStep() + self.cStartPeriod - 1)
846       # print str(a//1000).zfill(4), str(a%1000).zfill(3)
847       a = "".join([self.msPrec, str(a//1000).zfill(4), ".", str(a%1000).zfill(3)])
848       self.sPrec = readmap(a)
849       self.sPrec = pcraster.max(self.sPrec * 1.0, 0)
850       #sPrec= timeinput (msPrec);
851       #self.report(self.sPrec, "PrecMap")
852
853       #----------------------------
854       # Solar geometry (POTRAD 5)
855       # ----------------------------
856       # SolDec  :declination sun per day  between +23 & -23 [deg]
857       # HourAng :hour angle [-] of sun during day
858       # SolAlt  :solar altitude [deg], height of sun above horizon
859       self.SolDec  = -23.4 * pcraster.cos(360 * (self.sDay+10) / 365)
860       self.HourAng = 15 * (self.sHour - 12.01)
861       self.SolAlt  = scalar(pcraster.asin(scalar(pcraster.sin(self.Lat)*pcraster.sin(self.
    SolDec) + \
862                       pcraster.cos(self.Lat)*pcraster.cos(self.SolDec)*pcraster.cos(self.
    HourAng))))
863
864       # Solar azimuth
865       # ----------------------------
866       # SolAzi  :angle solar beams to N-S axes earth [deg]
867       self.SolAzi = scalar(pcraster.acos((pcraster.sin(self.SolDec)* pcraster.cos(self.Lat
    ) - \
868                    pcraster.cos(self.SolDec)* pcraster.sin(self.Lat)*pcraster.cos(self.
    HourAng)) \
869                                          /pcraster.cos(self.SolAlt)))
870       if self.sHour > 12:
871         self.SolAzi= 360 - self.SolAzi
```

```
872
873         # Additonal extra correction by R.Sluiter, Aug '99
874         if self.SolAzi > 89.994 and self.SolAzi < 90:
875           self.SolAzi = 90
876         if self.SolAzi > 269.994 and self.SolAzi < 270:
877           self.SolAzi = 270
878
879         # angle between the normal of the surface and the solar beams
880         self.sZenit = pcraster.acos((pcraster.sin(self.Lat)*pcraster.sin(self.SolDec)) + \
881                   (pcraster.cos(self.Lat)*pcraster.cos(self.SolDec)*pcraster.cos(self.
    HourAng)))
882         self.scosAbw = (pcraster.cos(self.SlopMap)*pcraster.cos(self.sZenit)) + \
883                        (pcraster.sin(self.SlopMap)*pcraster.sin(self.sZenit) * \
884                         pcraster.cos(self.SolAzi-self.AspMap))
885
886         ## NOTE: This works for [1,9999999]
887         a = int(self.currentTimeStep() + self.cStartPeriod - 1)
888         a = "".join([self.msTemp, str(a//1000).zfill(4), ".", str(a%1000).zfill(3)])
889         self.sTemp = readmap(a) - scalar(273.15) # directly using the input
890         ## NOTE: reading in Kelvin, using in Celsius
891         # sTemp= timeinput (msTemp); # directly using the input
892
893         # Potential Evapotranspiration Model:
894         #---------------------------------------
895         ## OCIN: This part was not implemented. Calculating ETP with Turc-Wendling, as in
896         ##       the original TACD model. ETP maps information was not available in Coello
897         ## Source: FAO 56 procedure for Global Radiation (Rg)
898
899         numdays = 366.0 if isLeapYear else 365.0
900         # dr: relative inverse distance from sun to earth
901         self.dr = 1 + 0.033 * math.cos(2 * math.pi * self.sDay / numdays)
902         # solar declination [rad]
903         self.SolDec = 0.409 * math.sin(2 * math.pi * self.sDay / numdays - 1.39)
904         # ws: solar radiation angle at dawn [rad]
905         self.ws = math.acos(-math.tan(self.Lat * math.pi / 180.0) * math.tan(self.SolDec))
906         # Ra: radiation outter atmosphere [MJ/m^2]
907         self.Ra = 24 * 60 * 0.082 / math.pi * self.dr*(self.ws*math.sin(self.Lat*math.pi/
    180.0)*\
908                                    math.sin(self.SolDec) + math.cos(self.Lat*math.pi/180.0
    )*\
909                                    math.cos(self.SolDec) * math.sin(self.ws))
910         # Daylength [h]
911         self.daylength = 24 * self.ws / math.pi
912         # Rg: Global radiation [MJ/(d * m^2)]
913         self.Rg = (0.25 + 0.5 * self.SunHours / self.daylength) * self.Ra
914
915         # now, calculating ETP
916         self.sPotET = ((self.Rg * 100) + 93 * self.fk)*(self.sTemp + scalar(22.0)) / \
917                       (150*(self.sTemp + scalar(123.0)))
918         #print "Ra = {0} MJ/m2, Rg = {1} mm/d".format(self.Ra, areamax(self.Rg))
919
920         ## NOTE: if ETP maps are available, comment previous lines and uncomment following
    3 lines of code
921         #a = int(self.currentTimeStep() + self.cStartPeriod - 1)
```

```
922       #a = "".join([self.msPotET, str(a//1000).zfill(4), ".", str(a%1000).zfill(3)])
923       #self.sPotET = readmap(a) # directly using the input
924       # -> sPotET = timeinput(msPotET);
925
926       #----REPORT-----REPORT-----REPORT------REPORT-----Report------
927       # Mean precipitation (mm/h)
928       #report tsMeanPrec = timeoutput(nSubCatch, sPrec);
929       self.tsMeanPrec.sample(self.sPrec)
930       # Annual precipitation including corrected snowfall (mm/a)
931       #report (cReportMaps) sCumPrec = if (time() ge cStartPeriod, sCumPrec + (if (sTemp
    <= cTT, sPrec * cSFCF, sPrec)), 0);
932       self.sCumPrec = self.sCumPrec + ifthenelse(self.sTemp <= self.cTT, \
933                                                  self.sPrec * self.cSFCF, self.sPrec)
934       if self.cLengthPeriod == self.currentTimeStep():
935         self.report(self.sCumPrec, "sCumPre")
936
937       #report (cReportMaps) sCumPrec_trans = if (time() ge cStartPeriod, sCumPrec_trans +
    (if (sTemp <= cTT, sPrec_trans , sPrec_trans)), 0);
938       # Mean temperature (°C)
939       #report tsMeanTemp = timeoutput(1, sTemp);
940       self.tsMeanTemp.sample(self.sTemp)
941       # Mean temperature during evaluation period (°C)
942       #sCumTemp = if (time() ge cStartPeriod, sCumTemp + sTemp, 0);
943       # report (cReportMaps) sMeanTemp = sCumTemp / cLengthPeriod;
944       self.sCumTemp = self.sCumTemp + self.sTemp
945       if self.cLengthPeriod == self.currentTimeStep():
946         self.report(self.sCumTemp/self.cLengthPeriod, "MeanTem")
947       # Potential evapotranspiration (mm/h)
948       #report tsPotET = timeoutput (1, sPotET);
949       self.tsPotET.sample(self.sPotET)
950
951       #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT TRANSPORT
952       #report (cReportMaps) sCumPrec_trans = if (time() ge cStartPeriod, sCumPrec_trans +
    (if (sTemp <= cTT, sPrec_trans * cSFCF, sPrec_trans)), 0);
953       #report tsMeanPrec_trans = timeoutput(nSubCatch, sPrec_trans);
954       #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT TRANSPORT
955
956       #----REPORT-----REPORT-----REPORT------REPORT-----Report------
957
958       #**************************
959       # snowroutine
960       #**************************
961       # Assign values for snow evaporation according to month
962       if self.sMon == 1:
963         self.sSnowET = self.cSnowETNovJan
964       elif self.sMon == 2:
965         self.sSnowET = self.cSnowETFeb
966       elif self.sMon <= 5:
967         self.sSnowET = self.cSnowETMarAprMay
968       elif self.sMon <= 9:
969         self.sSnowET = self.cSnowETSep
970       elif self.sMon <= 11:
```

```
971         self.sSnowET = self.cSnowETNovJan
972       else:
973         self.sSnowET = self.cSnowETDec
974 
975       ## only for test purposes
976       #report tsPrecAtSnowStation = timeoutput(nSnowStations, sPrec);
977       #report tsTempAtSnowStation= timeoutput(nSnowStations, sTemp);
978 
979       # Rain (mm/h) on snow (mm)?
980       # a) Add free water up to water content
981       self.sWaterContent = self.sWaterContent + ifthenelse(pcrand(self.sTemp > self.cTT,\
982                                                   self.sSnowPack > 0), self.sPrec, 0)
983 
984       #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT
985       #sWaterContent_trans = sWaterContent_trans + (if (sTemp > cTT and sSnowPack > 0,
    sPrec_trans, 0));
986       #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT
987 
988       # b) Substract added water from precipitation
989       self.sPrec = ifthenelse(pcrand(self.sTemp > self.cTT,\
990                         self.sSnowPack > 0), self.sWaterContent - \
991                                       (self.cCWH * self.sSnowPack), self.sPrec)
992       self.sPrec = pcraster.max(self.sPrec, 0)
993       self.sOverflow = pcraster.max(self.sWaterContent - (self.cCWH * self.sSnowPack), 0)
994 
995       # Limit water content to maximum water content (mm)
996       self.sWaterContent = pcraster.min(self.cCWH * self.sSnowPack, self.sWaterContent)
997 
998       # Check if there is new snow and add to snowpack (mm)
999       self.sSnowPack = self.sSnowPack + ifthenelse(self.sTemp <= self.cTT, self.sPrec *
    self.cSFCF, 0)
1000       self.sSnowPack_vorET = self.sSnowPack
1001 
1002       # Determine maximum evaporation of snow pack and substract it
1003       self.sSnowET   = pcraster.min(self.sSnowPack, pcraster.min(self.sSnowET, self.sPotET
    ))
1004       self.sSnowET   = cover(self.sSnowET,0)
1005       #self.sSnowET = scalar(0);
1006       self.sPotET   = self.sPotET - self.sSnowET
1007       self.sSnowPack = self.sSnowPack - self.sSnowET
1008 
1009       # Refreeze of fluid water in snow (temperature < cTT):
1010       # a)theoreticly possible amount (mm)
1011       self.sRefreeze = pcraster.max(self.cCFR * self.cCFMAX * (self.cTT - self.sTemp), 0)
1012 
1013       # b)real amount limited to water content (mm)
1014       self.sRefreeze = pcraster.min(self.sRefreeze, self.sWaterContent)
1015 
1016       # Add and substract refreeze (mm)
1017       self.sSnowPack = self.sSnowPack + self.sRefreeze
1018       self.sWaterContent = self.sWaterContent - self.sRefreeze
1019 
```

```
1020        # Melting snow (mm/h)(temperature > cTT_melt)?
1021        self.sMeltWater = ifthenelse(pcrand(self.sSnowPack > 0, self.sTemp > self.cTT_melt),
     \
1022                                     self.cCFMAX * (self.sTemp - self.cTT_melt), 0)
1023
1024        # Meltwater limited to snowpack
1025        self.sMeltWater = pcraster.min(self.sSnowPack , self.sMeltWater)
1026
1027        # All snow melted ?
1028        self.sSnowPack = self.sSnowPack - self.sMeltWater
1029
1030        # Theoretical water content (equals all available free and fixed water) (mm)
1031        self.sWaterContent = self.sWaterContent + self.sMeltWater
1032
1033        # Substract remaining water content from meltwater  (mm/h)
1034        # sWaterContent has to be taken in order to include the previously fixed water)
1035        self.sMeltWater = pcraster.max(self.sWaterContent - (self.cCWH * self.sSnowPack), 0)
1036
1037        # Output into soil (mm/h)
1038        self.sInSoil = ifthenelse(self.sTemp > self.cTT, self.sMeltWater + self.sPrec, self.
    sMeltWater)
1039        #auch bei Temperaturen über cTT wird wass aufgrund der Verdunstung aus dem SnowPack
    freigegeben
1040        self.sInSoil = cover(self.sInSoil, scalar(0))
1041
1042        # Limit water content to maximum water content (mm)
1043        self.sWaterContent = pcraster.min(self.cCWH * self.sSnowPack, self.sWaterContent)
1044
1045        #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1046        # Snow pack as water equivalent (mm)
1047        # reported for snow stations and as average for whole area     calar (tsPTQ,1);
1048        #report tsSnowAtStation = timeoutput (nSnowStations, sSnowPack);
1049        #report tsSnowPack      = timeoutput (1, sSnowPack);
1050        #report tsWaterContent = timeoutput (1, sWaterContent);
1051        #report tsMeltWater     = timeoutput (1, sMeltWater);
1052        self.tsSnowPack.sample(self.sSnowPack)
1053        self.tsWaterContent.sample(self.sWaterContent)
1054        self.tsMeltWater.sample(self.sMeltWater)
1055        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT
1056        #report tsSnowPack_trans      = timeoutput (1, sSnowPack_trans);
1057        #report tsWaterContent_trans      = timeoutput (1, sWaterContent_trans);
1058        #report tsMeltWater_trans     = timeoutput (1, sMeltWater_trans);
1059        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
    TRANSPORT TRANSPORT
1060
1061        #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1062
1063        #*************************************************
1064        # Interception
1065        #*************************************************
1066        if self.sMon == 1:
1067          self.sLeafarea = self.cLai_jan
1068          self.sCoverage = self.cCov_jan
```

```
      elif self.sMon ==2:
        self.sLeafarea = self.cLai_feb
        self.sCoverage = self.cCov_feb
      elif self.sMon == 3:
        self.sLeafarea = self.cLai_mar
        self.sCoverage = self.cCov_mar
      elif self.sMon == 4:
        self.sLeafarea = self.cLai_apr
        self.sCoverage = self.cCov_apr
      elif self.sMon == 5:
        self.sLeafarea = self.cLai_may
        self.sCoverage = self.cCov_may
      elif self.sMon == 6:
        self.sLeafarea = self.cLai_jun
        self.sCoverage = self.cCov_jun
      elif self.sMon == 7:
        self.sLeafarea = self.cLai_jul
        self.sCoverage = self.cCov_jul
      elif self.sMon == 8:
        self.sLeafarea = self.cLai_aug
        self.sCoverage = self.cCov_aug
      elif self.sMon == 9:
        self.sLeafarea = self.cLai_sep
        self.sCoverage = self.cCov_sep
      elif self.sMon == 10:
        self.sLeafarea = self.cLai_oct
        self.sCoverage = self.cCov_oct
      elif self.sMon == 11:
        self.sLeafarea = self.cLai_nov
        self.sCoverage = self.cCov_nov
      else:
        self.sLeafarea = self.cLai_dec
        self.sCoverage = self.cCov_dec

      # maximum interception storage capacity (mm)
      self.sStorageMax = 0.3 * (self.sCoverage * self.sLeafarea + (1 - self.sCoverage))
      self.sInSoil_old = self.sInSoil

      # intercepted precipitation (mm)
      self.sIntPrecNew = self.sStorageMax * (1 - (1 / \
                      (1 + (((self.sCoverage) * self.sInSoil) / self.sStorageMax))))
      self.sIntPrec = pcraster.min(self.sIntPrecOld + self.sIntPrecNew, self.sStorageMax)
      self.sInSoil = self.sInSoil - self.sIntPrecNew + (\
        pcraster.max(self.sIntPrecNew + self.sIntPrecOld - self.sStorageMax, 0))

      self.sInterceptionET = pcraster.min(self.sIntPrec, self.sPotET)
      self.sPotET = self.sPotET - self.sInterceptionET
      self.sIntPrecOld = self.sIntPrec - self.sInterceptionET

      ### report tsInterceptET = timeoutput (1, sInterceptionET);
      #report tsInterceptETCatch = timeoutput (nSubCatch, sInterceptionET);
      #report tsInterceptETBrugga = timeoutput (nBrugga, sInterceptionET);
      #report (cReportMaps) sCumInterceptET = if (time() ge cStartPeriod, sCumInterceptET
    + sInterceptionET, 0);
```

```
1122      #report (cReportMaps) sStorageMax          = sStorageMax;
1123      #report tsIntPrec  = timeoutput (1, sIntPrec);
1124      #report (cReportMaps) sIntPrecIni_new      = if (defined (clone), sIntPrecOld);
1125
1126      self.tsInterceptET.sample(self.sInterceptionET)
1127      self.sCumInterceptET = self.sCumInterceptET + self.sInterceptionET
1128      self.sIntPrecIni_new = ifthen(defined(self.clone), self.sIntPrecOld)
1129      if self.cLengthPeriod == self.currentTimeStep():
1130        self.report(self.sCumInterceptET, "sCumIET")
1131        self.report(self.sIntPrecOld, "sIPrInw")
1132      self.tsIntPrec.sample(self.sIntPrec)
1133
1134      #************************************************
1135      # Direct runoff of cells with stream or settlements
1136      #************************************************
1137
1138      # Split the direct input into stream <-> soil routine/urban split
1139      self.sStreamAreaFraction = pcraster.min(((self.sStreamWidth * self.sStreamLength) /
    cellarea()), scalar(1))
1140      self.sDirectIntoStream = ifthenelse(self.bStream, (self.sStreamAreaFraction * self.
    sInSoil), scalar(0))
1141      self.sInSoil = self.sInSoil - self.sDirectIntoStream
1142
1143      # Seperate handling of cells with settlements
1144      self.sUrbanFlux = ifthenelse(self.bUrban, self.sInSoil, scalar(0))
1145      self.sInSoil    = ifthenelse(self.bUrban, scalar(0), self.sInSoil)
1146
1147      # Route the urban runoff over settlement cells (within this timestep!)
1148      self.sUrbanState = self.sUrbanFlux * (1 - self.sUrbanSplit)
1149      self.sUrbanFlux = self.sUrbanFlux * self.sUrbanSplit
1150      #sUrbanFlux = if (sUrbanFlux > 0, sUrbanFlux, 0)
1151      self.sUrbanFlux_ = catchmenttotal(self.sUrbanFlux, self.LDDstream)
1152      self.sUrbanFlux_ = ifthenelse(self.bStream, scalar(0), self.sUrbanFlux_)
1153
1154      # Direct the urban runoff immediatelly into stream, when urban runoff meets stream
1155      self.sUrbanIntoStream = ifthenelse(self.bStream, upstream(self.LDDstream, self.
    sUrbanFlux_) + self.sUrbanFlux, 0)
1156
1157      # Direct the urban runoff into next downstream cell, if there is no stream,
1158      # and substract runoff into stream
1159      self.sInSoil = self.sInSoil + self.sUrbanState
1160      self.sInSoil = cover(self.sInSoil, scalar(0))
1161
1162      # Add the input into stream together
1163      self.sIntoStream = self.sDirectIntoStream + self.sUrbanIntoStream
1164
1165      #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1166      # Direct runoff into stream
1167      ### report tsQ_DirectStream = timeoutput (nRunoffStations,
    catchmenttotal(cover(sDirectIntoStream,0), StreamLDD));
1168      self.tsQ_DirectStream.sample(catchmenttotal(cover(self.sDirectIntoStream,0),\
1169                                                  self.StreamLDD))
1170      # Direct runoff of settlements into stream
1171      ### report tsQ_DirectUrban  = timeoutput (nRunoffStations,
```

```
    catchmenttotal(cover(sUrbanIntoStream,0), StreamLDD));
1172        self.tsQ_DirectUrban.sample(catchmenttotal(cover(self.sUrbanIntoStream,0), \
1173                                                   self.StreamLDD))
1174        #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1175
1176        #*******************************
1177        # Zone of saturated overlandflow
1178        #*******************************
1179
1180        # Input of rain or meltwater into "micro-topographic depression" storage (mm)
1181        self.sMTD_box = self.sMTD_box + self.sInSoil
1182        self.sMTD_box_old = self.sMTD_box
1183
1184        # Runoff out  of cells of saturated overlandflow, modified by slope (mm/h)
1185        # a) storage runoff
1186        self.sQ_SOF = pcraster.min((self.sMTD_box * self.cMTD_K) * self.cSlopeFactor, self.
    sMTD_box)
1187        self.sMTD_box = pcraster.max(self.sMTD_box - self.sQ_SOF, scalar(0))
1188
1189        #b) plus runoff over limit of cMTD(mm/h)
1190        self.sQ_SOF = self.sQ_SOF + pcraster.max(self.sMTD_box - self.cMTD, scalar(0))
1191
1192        # Check that storage is not above limit
1193        self.sMTD_box = pcraster.min(self.sMTD_box, self.cMTD)
1194
1195        # Evaporation out of MTD storage
1196        self.sActET_SOF = pcraster.min(self.sMTD_box, self.sPotET)
1197
1198        # Set evaporation out off MTD storage to zero if there is snow
1199        self.sActET_SOF = ifthenelse(self.sSnowPack > 0, scalar(0), self.sActET_SOF)
1200
1201        self.sMTD_vorET = self.sMTD_box
1202
1203        # Calculate new MTD content
1204        self.sMTD_box = self.sMTD_box - self.sActET_SOF
1205
1206        # Set actual ET_SOF to snow evaporation if there is snow
1207        self.sActET_SOF = self.sSnowET + self.sActET_SOF
1208
1209        ##########################################################################
1210        # Soil routine (for all zones except zones of saturated overlandflow) #
1211        ##########################################################################
1212
1213        # Save present soil moisture at beginn of timestep
1214        self.sSoilMoistureAtBegin = self.sSoilMoisture
1215
1216        # Calculate changing soil moisture content (after adding one mm infiltrated water a
    time)
1217        # and percolating water to runoff generation routine using two external functions
1218        #sToRunoffGeneration = soiloutput (sInSoil, sSoilMoisture, cFieldCapacity, cBETA);
1219        self.Q_dr = pcraster.max((self.sSoilMoisture + self.sInSoil - self.cFieldCapacity),
    scalar(0))
1220        self.I_net = self.sInSoil - self.Q_dr;
1221        self.sToRunoffGeneration = (self.sSoilMoisture / self.cFieldCapacity)**(self.cBETA)
```

```
    * self.I_net
1222
1223        #report (cReportMaps) SMtoQ = sToRunoffGeneration; # this is for debuging
1224        #sToRunoffGeneration = cover(sToRunoffGeneration, scalar(0));
1225        #sSoilMoisture = soilwater (sInSoil, sSoilMoisture, cFieldCapacity, cBETA);
1226        self.sSoilMoisture = pcraster.max((self.sSoilMoisture + self.I_net - \
1227                                  self.sToRunoffGeneration), scalar(0))
1228        #report (cReportMaps) SMmaps = sSoilMoisture; # this is for debuging
1229        if self.cLengthPeriod == self.currentTimeStep():    # this is for debuging
1230          self.report(self.sToRunoffGeneration, "SMtoQ")
1231          self.report(self.sSoilMoisture, "SMmaps")
1232        # due to the soilwater and soilouput is outside function, so here i define it by
    myself Ren
1233
1234        # Calculate actual evapotranspiration out of soil:
1235        # actual ET equals potential ET if certain fraction of field capacity is reached
    (cLP [-])
1236        self.sMeanSoilMoisture = (self.sSoilMoisture + self.sSoilMoistureAtBegin) / 2
1237        self.sFractionOfPotET = self.sMeanSoilMoisture / (self.cLP * self.cFieldCapacity)
1238        self.sActET = pcraster.min(self.sPotET * self.sFractionOfPotET, \
1239                                   pcraster.min(self.sPotET, self.sSoilMoisture))
1240
1241        # Set evapotranspiration outoff soil to zero if there is snow
1242        self.sActET = ifthenelse(self.sSnowPack > 0, 0, self.sActET)
1243        ## OCIN: self.report(self.sActET, "myActET") -->WORKS
1244        # Calculate new soil moisture content
1245        self.sSoilMoisture = pcraster.max((self.sSoilMoisture - self.sActET), 0)
1246
1247        # Set actual ET to snow evaporation if there is snow
1248        self.sActET = self.sSnowET + self.sActET
1249
1250        # Combine actual evatranspiration of SOF-zone and soil routine zones
1251        self.sActET = cover(self.sActET_SOF, self.sActET)
1252
1253        #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1254        # Actual soil moisture content (mm) and discharge to runoff generation routine (mm/h)
1255        # report tsSoilMoisture = timeoutput (nSoilType, sSoilMoisture);
1256        self.tsSoilMoisture.sample(self.sSoilMoisture)
1257        self.tsToRunoffGeneration.sample(self.sToRunoffGeneration)
1258        # Actual evapotranspiration (mm/h)
1259        self.tsActET.sample(self.sActET)
1260        self.sCumActET = self.sCumActET + self.sActET
1261
1262        if self.cLengthPeriod == self.currentTimeStep():    # this is for debuging
1263          self.report(ifthen(defined(self.nSoilType), self.sSoilMoisture), "InSMNew")
1264          self.report(self.sCumActET, "CumAcET")
1265
1266        #***********************************************
1267        # Runoff Generation Routine - Upper storage (US)
1268        #***********************************************
1269
1270        # Input of percolated water to upper storage (mm)
1271        ##sUS_box_pre = sUS_box - sUS_lat;
1272        self.sUS_box = self.sUS_box + self.sToRunoffGeneration
```

```
1273
1274        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
1275        ##sUS_trans_pre = sUS_trans - sUS_lat_trans;
1276        #sUS_trans = sUS_trans + sToRunoffGeneration_trans;
1277        #sUS_konz_mix = if (sUS_box > 0, sUS_trans/ sUS_box, 0);
1278        ###sUS_konz_pre = if (sUS_box_pre > 0, sUS_trans_pre/ sUS_box_pre, 0);
1279        ###sUS_konz_ev  = if ((sToRunoffGeneration + sUS_lat) > 0,
    (sToRunoffGeneration_trans + sUS_lat_trans)/ (sToRunoffGeneration + sUS_lat), 0);
1280        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
1281
1282        # Runoff out of upper storage, modified by slope (mm/h)
1283        self.sQ_US = pcraster.min((self.sUS_box * self.cUS_K) * self.cSlopeFactor, self.
    sUS_box)
1284        self.sUS_box = pcraster.max(self.sUS_box - self.sQ_US, scalar(0))
1285
1286        #-----------------------------------------
1287        #TEST plus runoff over limit of cells (mm/h)
1288        self.sQ_US = self.sQ_US + pcraster.max(self.sUS_box - self.cUS_H, 0)
1289
1290        # Check that storage is not above limit
1291        self.sUS_box = pcraster.min(self.sUS_box, self.cUS_H)
1292        #-----------------------------------------
1293
1294        # If upper and lower storage exist, then percolation into lower storage, else into
    groundwater (mm/h)
1295        self.sStorageLeak = ifthenelse (defined(self.cUS_T), pcraster.min(self.sUS_box, self
    .cUS_T),\
1296                                        pcraster.min(self.sUS_box, self.cAll_P))
1297
1298        # Remaining water in upper storage
1299        self.sUS_box = self.sUS_box - self.sStorageLeak
1300
1301        #*********************************************
1302        # Runoff Generation Routine - Lower storage (LS)
1303        #*********************************************
1304
1305        # Input of percolated water to upper storage (mm)
1306        ##sLS_box_pre = sLS_box - sLS_lat;
1307        self.sLS_box = self.sLS_box + self.sStorageLeak
1308
1309        self.sLS_box_prePump = self.sLS_box
1310
1311        # Groundwater uptake from US in the valley bottom
1312        ## OCIN: self.sLS_box = self.sLS_box - ifthenelse(defined(self.nWells),
    self.cPumpRate, scalar(0))
1313        ## OCIN: self.sLS_box = pcraster.max(self.sLS_box, scalar(0))
1314        self.sLS_box = pcraster.max(self.sLS_box - ifthenelse(defined(self.nWells), self.
    cPumpRate, 0), 0)
1315        # self.sLS_box = max (self.sLS_box - ifthenelse(defined(self.nWells),
    self.cPumpRate, scalar(0)), scalar(0))
1316        self.sPumpRate = self.sLS_box_prePump - self.sLS_box
1317        if self.currentTimeStep() >= self.cStartPeriod:
1318          self.sCumPumpRate = self.sCumPumpRate + ifthen(defined(self.clone), cover(self.
    sPumpRate, 0))
```

```
      else:
        self.sCumPumpRate = scalar(0)

      # Runoff out of lower storage, modified by slope (mm/h)
      self.sQ_LS = pcraster.min((self.sLS_box * self.cLS_K) * self.cSlopeFactor, self.
    sLS_box)

      # Remaining water in storage (mm)
      self.sLS_box = pcraster.max(self.sLS_box - self.sQ_LS,scalar(0))

      # flow at catchment outlet
      self.sQ_out = ifthenelse(pcrand(self.nOut == 1, self.nSoilType == 6), self.sQ_US +
    self.sQ_LS, scalar(0))
      if self.currentTimeStep() >= self.cStartPeriod:
        self.sCumQ_out = self.sCumQ_out + ifthen(defined(self.clone), cover(self.sQ_out, 0
    ))
      else:
        self.sCumQ_out = ifthenelse(self.clone, scalar(0), scalar(0))

      # Percolation out of lower storage into groundwater
      self.sToGroundwater = pcraster.min(self.sLS_box, self.cAll_P)

      # Remaining water in lower storage
      self.sLS_box = self.sLS_box - self.sToGroundwater

      #----REPORT-----REPORT-----REPORT------REPORT-----Report------
      # Water in storage (mm)
      ##    report tsMTD_box = timeoutput (1, sMTD_box);
      ##    report tsUS_box  = timeoutput (nSoilType, sUS_box);
      ##    report tsLS_box  = timeoutput (nSoilType, cover (sLS_box, 0));
      self.tsMTD_box.sample(self.sMTD_box)
      self.tsUS_box.sample(self.sUS_box)
      self.tsLS_box.sample(cover(self.sLS_box, scalar(0)))

      ##    report tsPumpRate = timeoutput (nWells, sPumpRate);
      ##    report tsPump = timeoutput (nWells, sLS_box);
      self.tsPumpRate.sample(self.sPumpRate)
      self.tsPump.sample(self.sLS_box)

      # Water flow out off storages (mm/h)
      ##    report tsQ_SOF = timeoutput (1, sQ_SOF);
      ##    report tsQ_US  = timeoutput (nSoilType, sQ_US);
      ##    report tsQ_LS  = timeoutput (nSoilType, cover (sQ_LS, 0));
      ##    report tsQ_out = timeoutput (1 , sQ_out);
      self.tsQ_SOF.sample(self.sQ_SOF)
      self.tsQ_US.sample(self.sQ_US)
      self.tsQ_LS.sample(cover(self.sQ_LS, 0))
      self.tsQ_out.sample(self.sQ_out)

      # Accumulate quantities for budget check
      if self.currentTimeStep() >= self.cStartPeriod:
        self.sCumQ_SOF = self.sCumQ_SOF + self.sQ_SOF
        self.sCumQ_US = self.sCumQ_US + self.sQ_US
        self.sCumQ_LS = self.sCumQ_LS + self.sQ_LS
```

```
1370      else:
1371        self.sCumQ_SOF = scalar(0)
1372        self.sCumQ_US = scalar(0)
1373        self.sCumQ_LS = scalar(0)
1374
1375      #*************************************************************
1376      # Runoff Generation Routine - Combine runoffs (SOF, US, LS)
1377      #*************************************************************
1378
1379      # Cover missing values in storage runoffs
1380      self.sQ_SOF = cover (self.sQ_SOF, 0)
1381      self.sQ_US  = cover (self.sQ_US, 0)
1382      self.sQ_LS  = cover (self.sQ_LS, 0)
1383
1384      # Combine storage runoffs
1385      self.sQ_ = self.sQ_SOF + self.sQ_LS + self.sQ_US
1386
1387      # Combine percolation from upper and lower storage into groundwater
1388      # plus NULL percolation for SOF-zone (mm/h)
1389      self.sToGroundwater = cover(self.sToGroundwater, self.sStorageLeak, 0)
1390
1391
1392      # report (cReportMaps) sCumToGW = if(clone, if (time() ge cStartPeriod, sCumToGW +
    sToGroundwater, 0));
1393      if self.currentTimeStep() >= self.cStartPeriod:
1394        self.sCumToGW = self.sCumToGW + ifthenelse(defined(self.clone), self.
    sToGroundwater, 0)
1395      else:
1396        self.sCumToGW = scalar(0)
1397      if self.cLengthPeriod == self.currentTimeStep():    # this is for debuging
1398        self.report(self.sCumToGW, "CumToGW")
1399
1400      ##report tsToGW = timeoutput (nSoilType, sToGroundwater);
1401      ##report tsUS_IntoStream = timeoutput (nRunoffStations,
    catchmenttotal(if(nRGType!=6,cover(sQ_US,0),0), StreamLDD));
1402      ##report tsLS_IntoStream = timeoutput (nRunoffStations,
    catchmenttotal(if(nRGType!=6,cover(sQ_LS,0),0), StreamLDD));
1403      ##report tsSOF_IntoStream = timeoutput (nRunoffStations,
    catchmenttotal(if(nRGType!=6,cover(sQ_SOF,0),0), StreamLDD));
1404      self.tsToGW.sample(self.sToGroundwater)
1405      self.tsUS_IntoStream.sample(\
1406        catchmenttotal(ifthenelse(self.nRGType!=6,cover(self.sQ_US,0),scalar(0)), self.
    StreamLDD))
1407      self.tsLS_IntoStream.sample(\
1408        catchmenttotal(ifthenelse(self.nRGType!=6,cover(self.sQ_LS,0),scalar(0)), self.
    StreamLDD))
1409      self.tsSOF_IntoStream.sample(\
1410        catchmenttotal(ifthenelse(self.nRGType!=6,cover(self.sQ_SOF,0),scalar(0)), self.
    StreamLDD))
1411
1412      #*****************************************************
1413      # Groundwater storage (one parameter for whole area except RG-Type valley bottom (6))
1414      #*****************************************************
1415
```

```
1416        # Input of percolated water to upper storage (mm)
1417        #sGW_box_pre = sGW_box;
1418        #sGW_box = sGW_box + sToGroundwater;
1419        self.sGW_box_pre = self.sGW_box
1420        self.sGW_box = self.sGW_box + self.sToGroundwater
1421
1422        # Groundwater runoff into stream (mm/h), RG-Type != 6
1423        #sGW_IntoStream = if ((nRGType != 6) and bStream, sGW_box, 0);
1424        #sGW_box       = if ((nRGType != 6) and bStream, 0, sGW_box);
1425        #sIntoStream   = sIntoStream + sGW_IntoStream;
1426        self.sGW_IntoStream = ifthenelse((self.nRGType!=6) & self.bStream, self.sGW_box,
    scalar(0))
1427        self.sGW_box = ifthenelse((self.nRGType!=6) & self.bStream, scalar(0), self.sGW_box)
1428        self.sIntoStream = self.sIntoStream + self.sGW_IntoStream
1429
1430        # Runoff out of groundwater storage, modified by slope (mm/h)
1431        #sQ_GW = if (defined (cGW_H), min((sGW_box * cGW_K) * cSlopeFactor, sGW_box), 0);
1432        #sGW_box =  if (defined (cGW_H), max(sGW_box - sQ_GW,0));
1433        self.sQ_GW = ifthenelse (defined (self.cGW_H), pcraster.min((self.sGW_box * self.
    cGW_K) * self.cSlopeFactor, self.sGW_box), scalar(0))
1434        self.sGW_box = ifthenelse (defined (self.cGW_H), pcraster.max(self.sGW_box - self.
    sQ_GW, scalar(0)), scalar(0))
1435
1436        #-----------------------------------------
1437        #TEST plus runoff over limit of cells (mm/h)
1438        #sQ_ = sQ_ + if (defined (cGW_H), max (sGW_box - cGW_H, 0), 0);
1439        self.sQ_ = self.sQ_ +ifthenelse(defined(self.cGW_H), pcraster.max(self.sGW_box -
    self.cGW_H, scalar(0)), scalar(0))
1440
1441        # Check that storage is not above limit
1442        #sGW_box = if (defined (cGW_H), min (sGW_box, cGW_H));
1443        self.sGW_box = ifthenelse (defined (self.cGW_H), pcraster.min(self.sGW_box, self.
    cGW_H), 0)
1444
1445        #************************************************
1446        # Runoff into stream and lateral flows
1447        #************************************************
1448
1449        # Runoff out of "stream cell" into stream (mm/h)
1450        #sQ_IntoStream = if (bStream and (nRGType != 6), sQ_, 0);
1451        #sIntoStream   = sIntoStream + sQ_IntoStream;
1452        #sQ_           = if (bStream and (nRGType != 6), 0, sQ_);
1453        #sQ_US         = if (bStream and (nRGType != 6), 0, sQ_US);
1454
1455        self.sQ_IntoStream = ifthenelse(self.bStream & (self.nRGType != 6), self.sQ_, 0)
1456        self.sIntoStream   = self.sIntoStream + self.sQ_IntoStream
1457        self.sQ_           = ifthenelse(self.bStream & (self.nRGType != 6), 0, self.sQ_)
1458        self.sQ_US         = ifthenelse(self.bStream & (self.nRGType != 6), 0, self.sQ_US)
1459
1460        # Lateral flows within groundwater storage (mm)
1461        #sGW_box = if (defined (cGW_H), sGW_box + upstream (LDD, sQ_GW));
1462        self.sGW_box = ifthenelse (defined (self.cGW_H), self.sGW_box + upstream (self.LDD,
    self.sQ_GW), self.sGW_box)
1463
```

```
1464        # Lateral flows (mm/h)
1465        # a) SOF-zone
1466        self.sMTD_box = self.sMTD_box + upstream(self.LDD, self.sQ_)
1467
1468        # b) Rest of storages
1469        # If the same RG-type is downstream, then runoff stays in the upper storage,
1470        # else it is combined and flows into lower storage, if there
1471        # if downstream RG-Type is valley bottom all storages are combined, including
    GW-flow and flow into the lower storage
1472        #sQ_US    = if (nRGType == downstream (LDD, nRGType), sQ_US, 0);
1473        #sUS_box = sUS_box + if (defined(cUS_T), upstream (LDD, sQ_US), upstream(LDD, sQ_));
1474        #sLS_box  = sLS_box + if (nRGType != 6, upstream (LDD, sQ_ - sQ_US), upstream (LDD,
    sQ_ - sQ_US + sQ_GW));
1475        self.sQ_US    = ifthenelse(self.nRGType == downstream(self.LDD, self.nRGType), self.
    sQ_US, 0)
1476        self.sUS_box  = self.sUS_box + ifthenelse (defined(self.cUS_T), upstream (self.LDD,
    self.sQ_US), upstream(self.LDD, self.sQ_));
1477        self.sLS_box  = self.sLS_box + ifthenelse (self.nRGType != 6, upstream (self.LDD,
    self.sQ_ - self.sQ_US), upstream (self.LDD, self.sQ_ - self.sQ_US + self.sQ_GW))
1478
1479        #sUS_lat = if (defined(cUS_T), upstream (LDD, sQ_US), upstream(LDD, sQ_));
1480        #sLS_lat = if (nRGType != 6, upstream (LDD, sQ_ - sQ_US), upstream (LDD, sQ_ -
    sQ_US + sQ_GW));
1481        self.sUS_lat  = ifthenelse(defined(self.cUS_T), upstream (self.LDD, self.sQ_US),
    upstream(self.LDD, self.sQ_))
1482        self.sLS_lat  = ifthenelse (self.nRGType != 6, upstream (self.LDD, self.sQ_ - self.
    sQ_US), upstream (self.LDD, self.sQ_ - self.sQ_US + self.sQ_GW))
1483
1484        # Lateral flow is diverted into upper storage if lower storage is full (D- and
    F-zone)
1485        self.sQ_LSfull = ifthenelse (defined(self.cLS_H), pcraster.max(self.sLS_box - self.
    cLS_H, 0), 0)
1486
1487        self.sUS_box   =  self.sUS_box + self.sQ_LSfull
                                                # add water to US
1488        self.sLS_box   = ifthenelse (defined(self.cLS_H), pcraster.min(self.sLS_box, self.
    cLS_H), self.sLS_box)                # set LS to limit
1489
1490        # Groundwater storage
1491        #report tsQ_GW   = timeoutput (not bStream, sQ_GW);
1492        #report tsGW_box = timeoutput (not bStream, sGW_box);
1493        #report (cReportMaps) sIniGW_box_new = if (defined (clone), sGW_box);
1494        #report (cReportMaps) sIniMTD_box_new = if (defined (clone), sMTD_box);
1495        #report (cReportMaps) sIniUS_box_new  = if (defined (clone), sUS_box);
1496        #report (cReportMaps) sIniLS_box_new  = if (defined (clone), sLS_box);
1497        self.tsQ_GW.sample(self.sQ_GW)
1498        self.tsGW_box.sample(self.sGW_box)
1499        if self.cLengthPeriod == self.currentTimeStep():
1500          self.report(self.sGW_box, "GW_ini")
1501          self.report(self.sMTD_box, "MTD_ini")
1502          self.report(self.sUS_box, "US_ini")
1503          self.report(self.sLS_box, "LS_ini")
1504
1505        # Water flow into streams
```

```
1506      #report tsIntoStream   = timeoutput (nRunoffStations, catchmenttotal
    (cover(sIntoStream, 0), StreamLDD));
1507      #report tsTotIntoStream = maptotal(sIntoStream * cMmToCubM);
1508      #report tsGW_IntoStream = timeoutput (nRunoffStations,
    catchmenttotal(cover(sGW_IntoStream,0), StreamLDD));
1509      self.tsIntoStream.sample(catchmenttotal (cover(self.sIntoStream, 0), self.StreamLDD))
1510      self.tsTotIntoStream.sample(maptotal(self.sIntoStream * self.cMmToCubM))
1511      self.tsGW_IntoStream.sample(catchmenttotal(cover(self.sGW_IntoStream,0), self.
    StreamLDD));
1512
1513      #sQ_IntoStream = sQ_IntoStream * sRGTypeStreamPart;
1514      #report tsQ_IntoStream = timeoutput (nRGTypeStream, sQ_IntoStream);
1515      #report (cReportMaps) IntoStream = if ((bStream == 1), sIntoStream);
1516      self.sQ_IntoStream = self.sQ_IntoStream * self.sRGTypeStreamPart
1517      if self.cLengthPeriod == self.currentTimeStep():
1518        #self.tsQ_IntoStream.sample(self.sQ_IntoStream)
1519        self.report(ifthenelse(self.bStream,self.sIntoStream, scalar(0)), "IntoStr")
1520
1521      # Accumulate quantity for budget check
1522      #sCumQ_GW = if (time() ge cStartPeriod, sCumQ_GW + sQ_GW, 0);
1523      #sCumIntoStream = if (time() ge cStartPeriod, sCumIntoStream + sIntoStream, 0);
1524      if self.currentTimeStep() >= self.cStartPeriod:
1525        self.sCumQ_GW = self.sCumQ_GW + self.sQ_GW
1526        self.sCumIntoStream = self.sCumIntoStream + self.sIntoStream
1527      else:
1528        self.sCumQ_GW = scalar(0)
1529        self.sCumIntoStream = scalar(0)
1530
1531      #*************************************************************
1532      # Channel routing using kinematic wave with Manning's equation
1533      #*************************************************************
1534      # Set hourly discharge and cumulative water depth to zero
1535      self.sQ             = scalar(0)
1536      self.sCumWaterDepth = scalar(0)
1537
1538      # Convert sIntoStream (mm/h) into lateral inflow per distance along stream ((m³/s)/m)
1539      self.sIntoStream = self.sIntoStream * self.cMmToCubM / self.sStreamLength
1540      if self.cLengthPeriod == self.currentTimeStep():
1541        self.report(self.sIntoStream, "sQmap3")
1542
1543      if (self.currentTimeStep() == self.cStartPeriod):
1544        if self.sVolumen > 0:
1545          self.sStartVol = catchmenttotal (cover (self.sVolumen,0), self.StreamLDD)
1546        else:
1547          self.sStartVol = catchmenttotal ((self.sAlpha*(self.sQ_step**self.cBeta))*self.
    sStreamLength*1000/cellarea(), self.StreamLDD)
1548      else:
1549        self.sStartVol = self.sStartVol
1550      # loop for kinematic wave calculation (in order to decrese the routing modelling
    timestep!!)
1551      #foreach step in aHour {
1552      #    Calculate stream runoff (m³/s)
1553      #   sQ_step = kinematic (StreamLDD, sQ_step, sIntoStream, sAlpha, cBeta, cTimeStep,
    sStreamLength);
```

```
1554
1555        #    Calculate new water depth (m)
1556        #   sWaterDepth = sAlpha * (sQ_step**cBeta) / sStreamWidth;
1557        #   report (cReportMaps) sDmap = sWaterDepth; # this is for debuging
1558
1559        #    Extra calculation of Alpha to reach a more precise water depth
1560        #   sAlpha = AlpTerm * (sStreamWidth + (2* sWaterDepth))** AlpPow;
1561        #   sWaterDepth = sAlpha * (sQ_step**cBeta) / sStreamWidth;
1562        #   sAlpha = AlpTerm * (sStreamWidth + (2* sWaterDepth))** AlpPow;
1563
1564        #    Add up the discharge for the whole timestep
1565        #   sQ = sQ + sQ_step ;
1566        #   report (cReportMaps) sQmap = sQ; # this is for debuging
1567
1568        #    For calculting the mean water depth for the whole timestep
1569        #   sCumWaterDepth = sCumWaterDepth + sWaterDepth;
1570        #}
1571
1572        """KINEMATIC WAVE CALCULATION"""
1573        for step in self.aHour:
1574          # Calculate stream runoff (m³/s)
1575          self.sQ_step2 = kinematic(self.StreamLDD, self.sQ_step, self.sIntoStream, self.
    sAlpha,\
1576                           self.cBeta, self.cNrSteps, 86400/len(self.aHour), self.
    sStreamLength)
1577          self.sQ_step = self.sQ_step2
1578
1579          # Calculate new water depth (m)
1580          self.sWaterDepth = self.sAlpha * (self.sQ_step**self.cBeta) / self.sStreamWidth
1581          if self.cLengthPeriod == self.currentTimeStep():
1582            self.report(self.sWaterDepth, "sDmap")
1583
1584          # Extra calculation of Alpha to reach a more precise water depth
1585          self.sAlpha = self.AlpTerm * (self.sStreamWidth + (2* self.sWaterDepth))** self.
    AlpPow
1586          self.sWaterDepth = self.sAlpha * (self.sQ_step**self.cBeta) / self.sStreamWidth
1587          self.sAlpha = self.AlpTerm * (self.sStreamWidth + (2* self.sWaterDepth))** self.
    AlpPow
1588
1589          # Add up the discharge for the whole timestep
1590          self.sQ = self.sQ + self.sQ_step
1591          if self.cLengthPeriod == self.currentTimeStep():
1592            self.report(self.sQ, "sQmap")
1593          # For calculting the mean water depth for the whole timestep
1594          self.sCumWaterDepth = self.sCumWaterDepth + self.sWaterDepth
1595
1596        # Calculate mean discharge per timestep
1597        self.sQ_sim = self.sQ / self.cNrSteps
1598
1599        # Convert sQ from m³/s in mm/h and add zero for missing values (= no stream cells)
1600        self.sQ_mm = cover(self.sQ_sim / self.cMmToCubM , 0)
1601
1602        # Calculate average water depth during timestep
1603        self.sAverageWaterDepth = self.sCumWaterDepth / self.cNrSteps
```

```
1604
1605        # Calculate mean water velocity during timestep (m/s)
1606        self.sVelocity = self.sQ_sim / (self.sStreamWidth * self.sAverageWaterDepth)
1607
1608        #----REPORT-----REPORT-----REPORT------REPORT-----Report------
1609        # Discharge at the gauging points (m3/s)
1610        #report tsQ_sim   = timeoutput(nRunoffStations, sQ_sim);
1611        self.tsQ_sim.sample(self.sQ_sim)
1612
1613        # Water depth (m) and velocity in streams (m/s)
1614        #report tsWaterDepth = timeoutput (nRunoffStations, sAverageWaterDepth);
1615        #report tsVelocity   = timeoutput (nRunoffStations, sVelocity);
1616        self.tsWaterDepth.sample(self.sAverageWaterDepth)
1617        self.tsVelocity.sample(self.sVelocity)
1618
1619        # Mean velocity for the evaluation period (m/s)
1620        #sCumVelocity = if (time() ge cStartPeriod, sCumVelocity + sVelocity, 0);
1621        self.sCumVelocity = self.sCumVelocity + self.sVelocity
1622
1623        if self.cLengthPeriod == self.currentTimeStep():
1624          self.report(self.sCumVelocity / self.cLengthPeriod, "MeanVel")
1625
1626        # Accumulate runoff for budget check (mm)
1627        #sCumRunoff = if (time() ge cStartPeriod, sCumRunoff + sQ_mm, 0);
1628        self.sCumRunoff = self.sCumRunoff + self.sQ_mm
1629
1630        #**********************
1631        #    Final storage
1632        #**********************
1633
1634        # Add up water of runoff generation storages (mm)
1635        self.sAll_box = cover (self.sMTD_box, self.sLS_box, scalar(0))
1636        self.sAll_box = ifthenelse (defined(self.clone), self.sAll_box + cover (self.sUS_box
    , 0), self.sAll_box)
1637
1638        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
1639        #sAll_trans = cover (sMTD_trans, sLS_trans, scalar (0));
1640        #sAll_trans = if (defined (clone), sAll_trans + cover (sUS_trans, 0));
1641        #TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT TRANSPORT
1642        #report tsAll_box = timeoutput(nRGType, sAll_box);
1643        #self.tsAll_box.sample(self.sAll_box)
1644
1645        #************************************************
1646        # BUDGET CHECK - without channel routing
1647        #************************************************
1648        # Budget checks can be done during the evaluation period for the whole catchment
1649        # and every subcatchment gauged through a station marked in nRunoffStations
1650
1651        # Initial storage + Precipitation =  ActualET + Runoff + Final storage
1652        # Final cumulative amount of precipitation (including snow), interception, actual
    evatranspiration, runoff (mm), pumping and unterground outflow
1653        self.sEndPrec       = catchmenttotal (self.sCumPrec, self.LDDstream)
1654        self.sEndInterceptET = catchmenttotal (self.sCumInterceptET, self.LDDstream)
1655        self.sEndActualET   = catchmenttotal (self.sCumActET, self.LDDstream)
```

```
1656      self.sEndIntoStream = catchmenttotal (self.sCumIntoStream, self.LDDstream) -
      catchmenttotal(self.sCumLeak, self.LDDstream)
1657      self.sEndPumpRate = catchmenttotal (self.sCumPumpRate, self.LDDstream)
1658      self.sEndQ_out = ifthenelse (self.nEndPit == 1, areatotal(self.sCumQ_out, self.clone
      ), 0)
1659      self.sEndQ_Inf = catchmenttotal (self.sCumQ_Inf, self.LDDstream)
1660
1661      # Final storage
1662      # = snow (pack + water content) + soil moisture + all RG-storages + groundwater
      storage + interception storage
1663      #sEndStorage = if (time() ge cStartPeriod, catchmenttotal(sSnowPack, LDDstream) +
      catchmenttotal(sWaterContent, LDDstream) +
1664      #                 catchmenttotal(if (defined (clone), cover (sSoilMoisture,0)),
      LDDstream) + catchmenttotal(sAll_box, LDDstream) +
1665      #                  catchmenttotal(if(defined(clone), cover (sGW_box,0)), LDDstream)
      + catchmenttotal (sIntPrecOld, LDDstream),0);
1666      if self.currentTimeStep() >= self.cStartPeriod:
1667        self.sEndStorage = catchmenttotal(self.sSnowPack, self.LDDstream)
1668        self.sEndStorage = self.sEndStorage + catchmenttotal(self.sWaterContent, self.
      LDDstream)
1669        self.sEndStorage = self.sEndStorage + catchmenttotal(ifthenelse(defined (self.
      clone), cover (self.sSoilMoisture,0),0), self.LDDstream)
1670        self.sEndStorage = self.sEndStorage + catchmenttotal(self.sAll_box, self.LDDstream)
1671        self.sEndStorage = self.sEndStorage + catchmenttotal(ifthenelse(defined(self.clone
      ), cover (self.sGW_box,0),0), self.LDDstream)
1672        self.sEndStorage = self.sEndStorage + catchmenttotal (self.sIntPrecOld, self.
      LDDstream)
1673      else:
1674        self.sEndStorage = 0
1675
1676      # Check water balance
1677      self.sBalance = self.sStartStorage + self.sEndPrec - self.sEndInterceptET - self.
      sEndActualET
1678      self.sBalance = self.sBalance - self.sEndIntoStream  - self.sEndPumpRate - self.
      sEndQ_out
1679      self.sBalance = self.sBalance - self.sEndStorage + self.sEndQ_Inf
1680
1681      self.tsBalance.sample(self.sBalance)
1682      self.tsStartStorage.sample(scalar(self.sStartStorage))
1683      self.tsEndStorage.sample(scalar(self.sEndStorage))
1684      self.tsEndPrec.sample(scalar(self.sEndPrec))
1685      self.tsEndInterceptET.sample(self.sEndInterceptET)
1686      self.tsEndActualET.sample(self.sEndActualET)
1687      self.tsEndIntoStream.sample(self.sEndIntoStream)
1688      self.tsEndPumpRate.sample(self.sEndPumpRate)
1689      self.tsEndQ_out.sample(self.sEndQ_out)
1690
1691      #*********************************
1692      # BUDGET CHECK - channel routing
1693      #*********************************
1694      if self.currentTimeStep() >= self.cStartPeriod:
1695        self.sEndVol = catchmenttotal (self.sVolumen, self.StreamLDD)
1696      else:
1697        self.sEndVol = 0
```

```
      self.sBalanceRouting = self.sEndIntoStream - self.sCumRunoff + self.sStartVol - \
                             self.sEndVol - self.sEndQ_Inf
      if self.currentTimeStep() >= self.cStartPeriod:
        self.sRoutingPercent = (self.sBalanceRouting / self.sEndIntoStream) * 100
      else:
        self.sRoutingPercent = 0

      self.tsBalanceRouting.sample(scalar(self.sBalanceRouting))
      self.tsRoutingPercent.sample(scalar(self.sRoutingPercent))

      #**************************************************
      # EVALUATION: LIKELIHOOD MEASURES AND VOLUME ERROR
      #preparations for the connected evaluation model EVAL
      #**************************************************

      #sQ_gauged = timeinputscalar (tsQ_gauged, nRunoffStations);
      #counter = if (defined (sQ_gauged), counter + 1, counter);
      #sQ_gauged = if (defined (sQ_gauged), sQ_gauged, 0);
      #sCumQ_gauged += sQ_gauged;
      #slogCumQ_gauged += if (sQ_gauged > 0, log10(sQ_gauged), 0);
      self.sQ_sim = ifthenelse( defined (self.nRunoffStations), self.sQ_sim, 0)
      self.sCumQ_sim = self.sCumQ_sim + self.sQ_sim

      if self.cLengthPeriod == self.currentTimeStep():
        self.report(self.sCumQ_sim/ self.cLengthPeriod, "AvgQsim")


  def moveFiles(lastStep):
    origin = [r"InSMNew*", r"sIPrInw*", r"GW_ini*", r"LS_ini*", r"MTD_ini*", r"US_ini*"]
    destin = [r"..\OUT\INI\SM_ini.map", r"..\OUT\INI\IntPrec_ini.map",\
              r"..\OUT\INI\GW_ini.map", r"..\OUT\INI\LS_ini.map",\
              r"..\OUT\INI\MTD_ini.map", r"..\OUT\INI\US_ini.map"]
    for i in range(len(origin)):
      os.popen("move /Y {0} {1}".format(origin[i], destin[i]))
    """origin = [r"..\OUT\INI\SM_ini.map", r"..\OUT\INI\IntPrec_ini.map",\
              r"..\OUT\INI\GW_ini.map", r"..\OUT\INI\LS_ini.map",\
              r"..\OUT\INI\MTD_ini.map", r"..\OUT\INI\US_ini.map"]
    destin = [r"..\INI\SM_ini.map", r"..\INI\IntPrec_ini.map",\
              r"..\INI\GW_ini.map", r"..\INI\LS_ini.map",\
              r"..\INI\MTD_ini.map", r"..\INI\US_ini.map"]
    for i in range(len(origin)):
      os.popen("copy /Y {0} {1}".format(origin[i], destin[i]))"""


  def modelExec(lastStep, ISINIT = False):
    """Execute TACD2 model up to given timestep"""
    # 94 -> 9314,    365 -> 9585,     730 -> 9950,    1252 -> 10472
    ## define here map of the catchment
    myModel = TACD2("..\\MAPS\\clone", ISINIT)
    ## Call the dynamic framework with the model, the last and first time steps
    ## Note: first time step should always be 1 (PCRaster condition)
    dynModelFw = DynamicFramework(myModel, lastTimeStep=lastStep, firstTimestep=1)
    dynModelFw.setQuiet(quiet = True)
    dynModelFw.run()
```

```
1752    moveFiles(lastStep)
1753
1754
1755  def readParams():
1756    """ reads para_ini.tbl file and returns its information as a dict """
1757    params = dict()
1758    with open("..\\INI\\para_ini.tbl", "r") as f:
1759      for line in f.readlines():
1760        data = [x for x in line.strip().split()]
1761        params[data[1]] = float(data[2])
1762    return params
1763
1764  ################## MODEL PARAMETERS ##################
1765  ti = None       #time interval and keys form params{}
1766  lkeys = ["pStartYear","pStartDay","pStartPeriod","pEndPeriod","pTT_urban","pTT",
      "pTT_melt","pTT_melt_forest","pTT_melt_urban","pSFCF","pCFMAX_urban","pCFMAX","pCWH",
      "pCFR","pUrbanSplit","pLP","pFC1","pFC2","pFC3","pFC4","pFC5","pFC6","pBETA1","pBETA2",
      "pBETA3","pBETA4","pBETA5","pBETA6","pAll_P","pMTD","pMTD_K","pDH_K","pDI_K_u","pDI_K_l"
      ,"pDI_H","pDI_T","pFI_K_u","pFI_K_l","pFI_H","pFI_T","pFLI_K_u","pFLI_K_l","pFLI_H",
      "pFLI_T","pEDI_K","pDV_K_u","pDV_K_l","pDV_H","pDV_T","pGW_K","pUS_H","pGW_H","pThres",
      "p_Exf","p_Inf","pPump_1","pPump_2","pPump_3","pBeta","pQIni","pWaterDepthIni",
      "pSnowETSep","pSnowETOct","pSnowETNovJan","pSnowETDec","pSnowETFeb","pSnowETMarAprMay"]
1767  mysubset = ["pStartYear","pStartDay","pStartPeriod","pEndPeriod","pTT_urban","pTT",
      "pTT_melt","pTT_melt_forest","pTT_melt_urban","pSFCF","pCFMAX_urban","pCFMAX","pCWH",
      "pCFR","pUrbanSplit","pLP","pFC1","pFC2","pFC3","pFC4","pFC5","pFC6","pBETA1","pBETA2",
      "pBETA3","pBETA4","pBETA5","pBETA6","pAll_P","pMTD","pMTD_K","pDH_K","pDI_K_u","pDI_K_l"
      ,"pDI_H","pDI_T","pFI_K_u","pFI_K_l","pFI_H","pFI_T","pFLI_K_u","pFLI_K_l","pFLI_H",
      "pFLI_T","pEDI_K","pDV_K_u","pDV_K_l","pDV_H","pDV_T","pGW_K","pUS_H","pGW_H","pThres",
      "p_Exf","p_Inf","pPump_1","pPump_2","pPump_3","pBeta","pQIni","pWaterDepthIni",
      "pSnowETSep","pSnowETOct","pSnowETNovJan","pSnowETDec","pSnowETFeb","pSnowETMarAprMay"]
1768  Qobs = None     # Observed Q at exit
1769  tic = None      # time at model start
1770  ######################################################
1771
1772  def writeParams(params):
1773    global lkeys
1774    """ writes params dictionary in para_ini.tbl file in INI folder, ready to be used """
1775    with open("..\\INI\\para_ini.tbl", "w") as f:
1776      for key in lkeys:
1777        #print "parameter {0} {1}\n".format(key, params[key])
1778        f.write("parameter {0} {1}\n".format(key, params[key]))
1779    pass
1780
1781
1782  def saveParams(params, num):
1783    global lkeys
1784    """ writes params dictionary in par#.tbl file in SCRIPTS folder, where # number is
1785    sent as parameter """
1786    with open("..\\SCRIPTS\\par{0}.tbl".format(int(num)), "w") as f:
1787      for key in lkeys:
1788        f.write("parameter {0} {1}\n".format(key, params[key]))
1789    pass
1790
1791
```

```
1792  def list2params(mylist):
1793    """from a given list, create the corresponding params dictionary"""
1794    global lkeys
1795    params = dict()
1796    for i in range(len(lkeys)):
1797      params[lkeys[i]] = mylist[i]
1798    return params
1799
1800
1801  def params2list(params):
1802    """from a given params dictionary, created a list"""
1803    global lkeys, ti
1804    mylist = []
1805    for k in lkeys:
1806      mylist.append(params[k])
1807    return mylist
1808
1809
1810  def loadObserved():
1811    global Qobs, obsNcol
1812    Qobs = []
1813    i, l, tmp = 0, None, None
1814    with open("..\\INI\\tsQ_obs.tss", "r") as f:
1815      for i in range(2):
1816        l = f.readline()
1817      obsNcol = int(l.strip()) - 1
1818      for i in range(obsNcol + 1):
1819        l = f.readline()
1820      i = 0
1821      for l in f.readlines():
1822        tmp = [float(x) for x in l.strip().split()]
1823        i += 1
1824        Qobs.append(tmp[1:])
1825
1826
1827  def calFunc(lparams):
1828    global ti, Qobs, obsNcol
1829    """ Function used for model calibration, returns sum of normalized squared error
1830    for the given parameters dictionary. This is used for optimize.minimize() """
1831    params = list2params(lparams)
1832    # forcing some of the parameters, to make them independent in minimize() sampling
1833    params["pStartYear"] = 2008
1834    params["pStartDay"] = 210
1835    #params["pStartPeriod"] = 9221 #for Payande
1836    params["pStartPeriod"] = 9221
1837    params["pEndPeriod"] = ti + params["pStartPeriod"] - 1 # setting new EndPeriod
1838    params["pPump_1"] = 0
1839    params["pPump_2"] = 0
1840    params["pPump_3"] = 0
1841    params["pTT_urban"] = -0.1
1842    params["pTT"] = -0.7759
1843    params["pTT_melt"] = -0.54095
1844    params["pTT_melt_forest"] = 1.5
1845    params["pTT_melt_urban"] = -1
```

```
    # now, writing some of the parameters
    writeParams(params)
    ans = [0.0 for x in range(obsNcol)]
    modelExec(ti)
    # now, open Estimated Q to calculate error
    with open("..\\OUT\\tsQ_sim.tss", "r") as f:
      for i in range(obsNcol + 4):
        l = f.readline()
      i = 0
      for l in f.readlines():
        Qsim = [float(x) for x in l.strip().split()]
        for j in range(obsNcol):
          ans[j] += ((Qobs[i][j] - Qsim[j+1])/Qobs[i][j])**2
        i += 1
    a = sum(ans)
    print "   " + str(a)
    return a


  def duration():
    global tic
    toc = time.time()
    h = (toc - tic) // 3600
    m = (toc - tic) // 60 - h * 60
    s = (toc - tic) - h * 3600 - m * 60
    print "Time Elapsed: {0:02d}h {1:02d}m {2:02.5f}s".format(int(h), int(m), s)


  def calibrate():
    global ti
    """ Model calibration according to the input data using scipy.optimize.minimize() """
    #tempInterv = [12, 24, 47, 94, 188, 365, 730, 1252]
    #tempInterv = [47, 94, 188, 365, 730, 1095, 1252]
    tempInterv = [94, 188, 365, 730, 1095, 1252]
    # par_ini.tbl: 94 -> 9314,    365 -> 9585,     730 -> 9950,    1252 -> 10472
    # sequential length usage for speeding up parameter calibration
    pcount = 1  # parameter counter for saving intermediate optimization steps

    for i in range(len(tempInterv)):
      ti = tempInterv[i]
      condition = False
      # FIRST PART: MODEL WARMUP
      #if i==0:
        #print "First Initialization..."
        #modelExec(ti, True)
      """print "Warmup for time=" + str(ti) + "..."
      modelExec(ti, False)"""
      duration()
      # SECOND PART: MODEL CALIBRATION
      print "Starting error minimization..."
      while not condition:
        x0 = params2list(readParams())
        print "iteration #{0} (l = {1})".format(pcount, ti)
        sol = optimize.minimize(calFunc, x0, options={'maxiter': 3, 'disp':True})
```

```
        saveParams(readParams(), pcount)
        pcount += 1
        print sol
        condition = sol.success


tic = time.time()
print time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime())
loadObserved()
#print Qobs
#for i in range(10):
#  print "Iteration: "+str(i+1)
#  modelExec(365, False)
modelExec(365, False)
#calibrate()
duration()
print time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime())

```