**Initialization Procedures:**

Initialize constants

Initialize file I/O

Initialize tables

Initialize terrain maps

Initialize network

Initialize meteorology

Initialize meteorology map

Initialize Dump

Initialize snow map

Initialize basin-wide value

Initialize model state

Initialize new month

Initialize new day

**Sediment Initialization Procedures:**

Initialize parameters

Initialize sediment table

Initialize fine resolution map

Initialize sediment map

**Setup for mass balance:**

Start water balance calculation

Start sediment balance calculation

**Start Perform calculations (Time loop):**

# DHSVM\_SWMM\_Network\_Coupling

**SWMM routing (Conduit routing underground):**

dynwave\_execute (dynwave.c)

findLinkFlows (dynwave.c)

dwflow\_findConduitFlow (dwflow.c)

updateNodeFlows (dynwave.c)

Determine erosion and routing scheme (sediment)

**Initialization Procedures:**

Initialize new step

Initialize channel network (if has network)

**Start grid loop (x,y):**

Read local meteorology data

Read soil temperature

**Mass energy balance**

**Routing process (consider snow):**

Subsurface routing

Channel routing

Pipe routing (RoutingPipe.c)

Surface routing

(Sediment routing, remove)

**Mass energy balance:**

Radiation Balance

Leaf drip impact calculation

Rainfall impact calculation

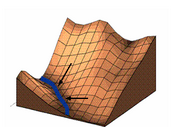
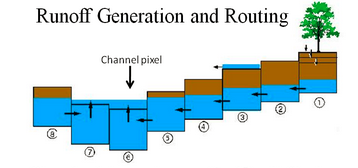
Interception storage & through fall calculation

ET calculation

Calculation for surface water & Road water & infiltration & unsaturated flow & channel grid water & surface temperature & sensible/no sensible heat & longwave balance

Aggregate radiation

**Perform Calculations (Time loop t++)**



**Dump intermediate results**

**End up with mass balance**

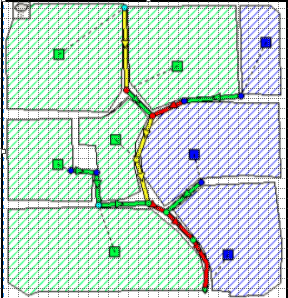
**Update mass balance & graphics:**

Draw output

Aggregation

Mass balance

# Grid layers design



* If cell has pipe node, then flow sink into this node as input for pipe. Need to modify the input q for pipe node.
* The other surface/channel routing is the same as DHSVM.
* The surface sub-regions do not need to be divided.

# DHSVM SWMM Coupling Code Structure

**Step 1. Modify MainDHSVM.c**

If (Options.HasNetwork) (line 413)

RouteChannel (DHSVMChannel.c)

Else if (Options.HasPipeChannelNetwokr){

RouteChannel (DHSVMChannel.c) //update routing in the channel

RoutePipe (RoutePipe.c) //update routing in the pipes network underground

}

**Step 2. Create an option “Pipe Channel Network” in InitConstants.c**

It is an option for the network, which is including pipes and channels

**Step 3. Create ContributionAreaNode.c for pipe flow (optional)**

This file should calculate the drainage area for each input node

**Step 4. Create RoutePipe.C**

RoutePipe.c should include the capabilities of the following functions:

**dynwave\_execute** (dynwave.c) **//** routes flows through drainage network over current time step

**findLinkFlows** (dynwave.c) **//** find new flow in each non-dummy conduit

**dwflow\_findConduitFlow** (dwflow.c) **//** updates flow in conduit link by solving finite difference form of continuity and momentum equations.

**updateNodeFlows** (dynwave.c) // updates cumulative inflow & outflow at link's end nodes

**findBypassedLinks** (dynwave.c) // check if link calculations can be skipped in next step

**findLimitedLinks** (dynwave.c) // identify any capacity-limited conduits

**findNodeDepths** (dynwave.c) //compute new depth for all nodes

# Appendix

int **dynwave\_execute** (double tStep)

//

// Input: links = array of topo sorted links indexes

// tStep = time step (sec)

// Output: returns number of iterations used

// Purpose: routes flows through drainage network over current time step.

//

{

int converged;

// --- initialize

if ( ErrorCode ) return 0;

Steps = 0;

converged = FALSE;

Omega = OMEGA;

initRoutingStep();

// --- keep iterating until convergence

while ( Steps < MaxTrials )

{

// --- execute a routing step & check for nodal convergence

initNodeStates();

**findLinkFlows(tStep);**

converged = findNodeDepths(tStep);

Steps++;

if ( Steps > 1 )

{

if ( converged ) break;

// --- check if link calculations can be skipped in next step

findBypassedLinks();

}

}

if ( !converged ) NonConvergeCount++;

// --- identify any capacity-limited conduits

findLimitedLinks();

return Steps;

}

void **findLinkFlows**(double dt)

{

int i;

// --- find new flow in each non-dummy conduit

#pragma omp parallel num\_threads(NumThreads) //(5.1.008)

{

#pragma omp for //(5.1.008)

for ( i = 0; i < Nobjects[LINK]; i++)

{

if ( isTrueConduit(i) && !Link[i].bypassed )

**dwflow\_findConduitFlow**(i, Steps, Omega, dt);

}

}

// --- update inflow/outflows for nodes attached to non-dummy conduits

for ( i = 0; i < Nobjects[LINK]; i++)

{

if ( isTrueConduit(i) ) **updateNodeFlows**(i);

}

// --- find new flows for all dummy conduits, pumps & regulators

for ( i = 0; i < Nobjects[LINK]; i++)

{

if ( !isTrueConduit(i) )

{

if ( !Link[i].bypassed ) findNonConduitFlow(i, dt);

**updateNodeFlows**(i);

}

}

}

void **dwflow\_findConduitFlow**(int j, int steps, double omega, double dt)

//

// Input: j = link index

// steps = number of iteration steps taken

// omega = under-relaxation parameter

// dt = time step (sec)

// Output: returns new flow value (cfs)

// Purpose: updates flow in conduit link by solving finite difference

// form of continuity and momentum equations.

//

{

int k; // index of conduit

int n1, n2; // indexes of end nodes

double z1, z2; // upstream/downstream invert elev. (ft)

double h1, h2; // upstream/dounstream flow heads (ft)

double y1, y2; // upstream/downstream flow depths (ft)

double a1, a2; // upstream/downstream flow areas (ft2)

double r1; // upstream hyd. radius (ft)

double yMid, rMid, aMid; // mid-stream or avg. values of y, r, & a

double aWtd, rWtd; // upstream weighted area & hyd. radius

double qLast; // flow from previous iteration (cfs)

double qOld; // flow from previous time step (cfs)

double aOld; // area from previous time step (ft2)

double v; // velocity (ft/sec)

double rho; // upstream weighting factor

**double sigma; // inertial damping factor**

double length; // effective conduit length (ft)

double dq1, dq2, dq3, dq4, dq5, // terms in momentum eqn.

dq6; // term for evap and infil losses

double denom; // denominator of flow update formula

double q; // new flow value (cfs) , outflow

**double barrels; // number of barrels in conduit**

TXsect\* xsect = &Link[j].xsect; // ptr. to conduit's cross section data

char isFull = FALSE; // TRUE if conduit flowing full

char isClosed = FALSE; // TRUE if conduit closed

// --- adjust isClosed status by any control action

if ( Link[j].setting == 0 ) isClosed = TRUE;

// --- get flow from last time step & previous iteration

k = Link[j].subIndex;

barrels = Conduit[k].barrels;

qOld = Link[j].oldFlow / barrels;

qLast = Conduit[k].q1;

// --- get most current heads at upstream and downstream ends of conduit

n1 = Link[j].node1;

n2 = Link[j].node2;

z1 = Node[n1].invertElev + Link[j].offset1;

z2 = Node[n2].invertElev + Link[j].offset2;

h1 = Node[n1].newDepth + Node[n1].invertElev;

h2 = Node[n2].newDepth + Node[n2].invertElev;

h1 = MAX(h1, z1);

h2 = MAX(h2, z2);

// --- get unadjusted upstream and downstream flow depths in conduit

// (flow depth = head in conduit - elev. of conduit invert)

y1 = h1 - z1;

y2 = h2 - z2;

y1 = MAX(y1, FUDGE);

y2 = MAX(y2, FUDGE);

// --- flow depths can't exceed full depth of conduit

y1 = MIN(y1, xsect->yFull);

y2 = MIN(y2, xsect->yFull);

// -- get area from solution at previous time step

aOld = Conduit[k].a2;

aOld = MAX(aOld, FUDGE);

// --- use Courant-modified length instead of conduit's actual length

length = Conduit[k].modLength;

// --- find surface area contributions to upstream and downstream nodes

// based on previous iteration's flow estimate

findSurfArea(j, qLast, length, &h1, &h2, &y1, &y2);

// --- compute area at each end of conduit & hyd. radius at upstream end

a1 = getArea(xsect, y1);

a2 = getArea(xsect, y2);

r1 = getHydRad(xsect, y1);

// --- compute area & hyd. radius at midpoint

yMid = 0.5 \* (y1 + y2);

aMid = getArea(xsect, yMid);

rMid = getHydRad(xsect, yMid);

// --- alternate approach not currently used, but might produce better

// Bernoulli energy balance for steady flows

//aMid = (a1+a2)/2.0;

//rMid = (r1+getHydRad(xsect,y2))/2.0;

// --- check if conduit is flowing full

if ( y1 >= xsect->yFull &&

y2 >= xsect->yFull) isFull = TRUE;

// --- set new flow to zero if conduit is dry or if flap gate is closed

if ( Link[j].flowClass == DRY ||

Link[j].flowClass == UP\_DRY ||

Link[j].flowClass == DN\_DRY ||

isClosed ||

aMid <= FUDGE )

{

Conduit[k].a1 = 0.5 \* (a1 + a2);

Conduit[k].q1 = 0.0;;

Conduit[k].q2 = 0.0;

Link[j].dqdh = GRAVITY \* dt \* aMid / length \* barrels;

Link[j].froude = 0.0;

Link[j].newDepth = MIN(yMid, Link[j].xsect.yFull);

Link[j].newVolume = Conduit[k].a1 \* link\_getLength(j) \* barrels;

Link[j].newFlow = 0.0;

return;

}

// --- compute velocity from last flow estimate

v = qLast / aMid;

if ( fabs(v) > MAXVELOCITY ) v = MAXVELOCITY \* SGN(qLast);

// --- compute Froude No.

Link[j].froude = link\_getFroude(j, v, yMid);

if ( Link[j].flowClass == SUBCRITICAL &&

Link[j].froude > 1.0 ) Link[j].flowClass = SUPCRITICAL;

// --- find inertial damping factor (sigma)

if ( Link[j].froude <= 0.5 ) sigma = 1.0;

else if ( Link[j].froude >= 1.0 ) sigma = 0.0;

else sigma = 2.0 \* (1.0 - Link[j].froude);

// --- get upstream-weighted area & hyd. radius based on damping factor

// (modified version of R. Dickinson's slope weighting)

rho = 1.0;

if ( !isFull && qLast > 0.0 && h1 >= h2 ) rho = sigma;

aWtd = a1 + (aMid - a1) \* rho;

rWtd = r1 + (rMid - r1) \* rho;

// --- determine how much inertial damping to apply

if ( InertDamping == NO\_DAMPING ) sigma = 1.0;

else if ( InertDamping == FULL\_DAMPING ) sigma = 0.0;

// --- use full inertial damping if closed conduit is surcharged

if ( isFull && !xsect\_isOpen(xsect->type) ) sigma = 0.0;

// --- compute terms of momentum eqn.:

// --- 1. friction slope term

if ( xsect->type == FORCE\_MAIN && isFull )

dq1 = dt \* forcemain\_getFricSlope(j, fabs(v), rMid);

else dq1 = dt \* Conduit[k].roughFactor / pow(rWtd, 1.33333) \* fabs(v);

// --- 2. energy slope term

dq2 = dt \* GRAVITY \* aWtd \* (h2 - h1) / length;

// --- 3 & 4. inertial terms

dq3 = 0.0;

dq4 = 0.0;

if ( sigma > 0.0 )

{

dq3 = 2.0 \* v \* (aMid - aOld) \* sigma;

dq4 = dt \* v \* v \* (a2 - a1) / length \* sigma;

}

// --- 5. local losses term

dq5 = 0.0;

if ( Conduit[k].hasLosses )

{

dq5 = findLocalLosses(j, a1, a2, aMid, qLast) / 2.0 / length \* dt;

}

// --- 6. term for evap and seepage losses per unit length

dq6 = link\_getLossRate(j, qOld, dt) \* 1.5 \* dt \* v / link\_getLength(j); //(5.1.008)

// --- combine terms to find new conduit flow

denom = 1.0 + dq1 + dq5;

q = (qOld - dq2 + dq3 + dq4 - dq6) / denom;

// --- compute derivative of flow w.r.t. head

Link[j].dqdh = 1.0 / denom \* GRAVITY \* dt \* aWtd / length \* barrels;

// --- check if any flow limitation applies

Link[j].inletControl = FALSE;

Link[j].normalFlow = FALSE;

if ( q > 0.0 )

{

// --- check for inlet controlled culvert flow

if ( xsect->culvertCode > 0 && !isFull )

q = culvert\_getInflow(j, q, h1);

// --- check for normal flow limitation based on surface slope & Fr

else

if ( y1 < Link[j].xsect.yFull &&

( Link[j].flowClass == SUBCRITICAL ||

Link[j].flowClass == SUPCRITICAL )

) q = checkNormalFlow(j, q, y1, y2, a1, r1);

}

// --- apply under-relaxation weighting between new & old flows;

// --- do not allow change in flow direction without first being zero

if ( steps > 0 )

{

q = (1.0 - omega) \* qLast + omega \* q;

if ( q \* qLast < 0.0 ) q = 0.001 \* SGN(q);

// omega = under-relaxation parameter

}

// --- check if user-supplied flow limit applies

if ( Link[j].qLimit > 0.0 )

{

if ( fabs(q) > Link[j].qLimit ) q = SGN(q) \* Link[j].qLimit;

}

// --- check for reverse flow with closed flap gate

if ( link\_setFlapGate(j, n1, n2, q) ) q = 0.0;

// --- do not allow flow out of a dry node

// (as suggested by R. Dickinson)

if( q > FUDGE && Node[n1].newDepth <= FUDGE ) q = FUDGE;

if( q < -FUDGE && Node[n2].newDepth <= FUDGE ) q = -FUDGE;

// --- save new values of area, flow, depth, & volume

Conduit[k].a1 = aMid;

Conduit[k].q1 = q;

Conduit[k].q2 = q;

Link[j].newDepth = MIN(yMid, xsect->yFull);

aMid = (a1 + a2) / 2.0;

aMid = MIN(aMid, xsect->aFull);

Conduit[k].fullState = link\_getFullState(a1, a2, xsect->aFull); //(5.1.008)

Link[j].newVolume = aMid \* link\_getLength(j) \* barrels;

Link[j].newFlow = q \* barrels;

}

void **updateNodeFlows**(int i)

//

// Input: i = link index

// q = link flow rate (cfs)

// Output: none

// Purpose: updates cumulative inflow & outflow at link's end nodes.

//

{

int k, m;

int barrels = 1;

int n1 = Link[i].node1;

int n2 = Link[i].node2;

double q = Link[i].newFlow;

double uniformLossRate = 0.0;

// --- compute any uniform seepage loss from a conduit

if ( Link[i].type == CONDUIT )

{

k = Link[i].subIndex;

uniformLossRate = Conduit[k].evapLossRate + Conduit[k].seepLossRate;

barrels = Conduit[k].barrels;

}

// --- update total inflow & outflow at upstream/downstream nodes

if ( q >= 0.0 )

{

Node[n1].outflow += q + uniformLossRate;

Node[n2].inflow += q;

}

else

{

Node[n1].inflow -= q;

Node[n2].outflow -= q - uniformLossRate;

}

// --- add surf. area contributions to upstream/downstream nodes

Xnode[Link[i].node1].newSurfArea += Link[i].surfArea1 \* barrels;

Xnode[Link[i].node2].newSurfArea += Link[i].surfArea2 \* barrels;

// --- update summed value of dqdh at each end node

Xnode[Link[i].node1].sumdqdh += Link[i].dqdh;

if ( Link[i].type == PUMP )

{

k = Link[i].subIndex;

m = Pump[k].pumpCurve;

if ( Curve[m].curveType != PUMP4\_CURVE )

{

Xnode[n2].sumdqdh += Link[i].dqdh;

}

}

else Xnode[n2].sumdqdh += Link[i].dqdh;

}

int **findNodeDepths**(double dt)

{

int i;

int converged; // convergence flag

double yOld; // previous node depth (ft)

// --- compute outfall depths based on flow in connecting link

for ( i = 0; i < Nobjects[LINK]; i++ ) link\_setOutfallDepth(i);

// --- compute new depth for all non-outfall nodes and determine if

// depth change from previous iteration is below tolerance

converged = TRUE;

#pragma omp parallel num\_threads(NumThreads) //(5.1.008)

{

#pragma omp for private(yOld) //(5.1.008)

for ( i = 0; i < Nobjects[NODE]; i++ )

{

if ( Node[i].type == OUTFALL ) continue;

yOld = Node[i].newDepth;

setNodeDepth(i, dt);

Xnode[i].converged = TRUE;

if ( fabs(yOld - Node[i].newDepth) > HeadTol )

{

converged = FALSE;

Xnode[i].converged = FALSE;

}

}

} //(5.1.008)

return converged;

}