Delta-Tulare Basin Model Read-me

To run the model, execute main.py from the cord-combined folder. To plot the output data, run make\_plots.py from the cord-combined folder.

This model is an attempt to translate detailed, daily information about snowpack and runoff in CA’s Sierra Nevada into information about the timing and magnitude of water availability for local and imported water contracts in the Tulare Basin. The model is designed in two parts – the ‘northern’ portion, covering the SF Delta and 7 major reservoirs that control flow in the Sacramento/San Joaquin Basin, and the ‘southern’ portion, covering the system of canals, surface water storage, irrigated acreage, and direct groundwater recharge basins that control the flow of local and imported water in the Tulare Basin. The two sub-models are connected through San Luis Reservoir, with pumping and allocations from Delta-based surface water contracts (State Water Project, Central Valley Project – Delta, Central Valley Project –Exchange, and Central Valley Project – Cross Valley Canal) calculated in the ‘northern’ sub-model and delivered as input variables to the ‘southern’ sub-model, and storage/demand conditions at San Luis used to develop pumping constraints that are calculated in the ‘southern’ sub-model and delivered to the ‘northern’ sub-model. The sub-model interactions occur in each of the model daily timesteps.

This read-me document discusses the model in three parts:

1. Data Input
2. Pre-processing
3. Model classes, functions & output
4. Data Input

The simulation model is forced by timeseries data contained in cord-data.csv. The file uses daily data, and the simulation nature of the model means that the duration of the data controls the duration of the simulation. There is currently a data file that allows the model to run in ‘calibration mode’ (October 1996 – September 2016) for which there exists detailed and reasonably observed values for model output data that can be used to validate model results. There is also a longer (October 1905-September 2016; under development) data file that allows the model to run in expanded ‘extended mode’, but this data file only includes model forcing data – observed values that can be compared to model outputs do not exist (b/c many/all of the structures – dams, pumping, canals for which the model provides output data did not exist for the entirety of the simulation period. Other data files can be generated and used as model input to examine the impact of alternative scenarios (‘projection mode’). These data files can be of any duration, but must be continuous daily timeseries.

Basin List:

1. Sacramento – Shasta Dam (data key: SHA)
2. Feather – Oroville Dam (data key: ORO)
3. Yuba – New Bullards Dam (data key: YRS)
4. American – Folsom Dam (data key: FOL)
5. Stanislaus – New Melones Dam (data key: NML)
6. Tuolumne – Don Pedro Dam (data key: DNP)
7. Merced – Exchequer Dam (data key: EXC)
8. San Joaquin – Millerton Dam (data key: MIL)
9. Kaweah – Kaweah Dam (data key: KWH)
10. Tule – Success Dam (data key: SUC)
11. Kern – Isabella Dam (data key: ISB)

Input series for each basin:

In generating the ‘extended’ data input file, statistical relationships were used to generate synthetic timeseries of data for all of the data series except for full-natural-flow and snowpack. In developing projected scenarios, we only need complete timeseries for FNF & SNOW, the remaining series can be generated in pre-processing (currently migrating this code into the Python model)

1. Reservoir inflows (AF; data key: \_inf) – daily values for inflows to the ‘main’ watershed reservoirs (identified in the basin list). Reservoir inflows are different from hydrologic ‘full-natural-flows’, but can be statistically generated from full-natural-flow timeseries (res. inflow generation code under development)
2. Snowpack (index; data key: \_snow) – daily values of a single snowpack index at each reservoir. Snowpack indices can be any combination of multiple snowpack measurements within the basin. Different units/weights for snowpack measurements can be used. If the input data timeseries is short, it may be necessary to include a longer timeseries of the snowpack index and reservoir inflow to improve the predictive ability of a linear regression between the two variables (more on the snow/flow regressions in the pre-processing section). Snowpack data used in the model must be strictly increasing throughout a given wateryear (i.e., reflect the maximum snowpack observation to that point in the wateryear – no record of snow-melt is needed).
3. Full-natural-flow (AF; data key: \_fnf) – daily values of full-natural flow within the basin headwaters (i.e., flow at the river location of the main reservoir). FNF is used within the model to develop river indices that govern different environmental rules (not for actual model forcing)
4. Evaporation (AF; data key: \_evap) – daily values for reservoir evaporation
5. Precipitation (AF: data key: \_precip) – daily values for reservoir precipitation
6. Flood control index (index; data key: \_fci) – these are calculated in the pre-processing from other input variables (this can be moved inside the code as well)
7. ‘Gains’ (AF; data key: \_gains) – this is the change in flow between reservoir outflow and the downstream gauge point within each watershed. This is a mixture of consumptive withdrawals, GW flux, and inflow from ungauged tributaries/creeks downstream of the reservoir outflow point.

Calibration series for each basin:

1. Reservoir storage (tAF; data key: \_storage) – daily storage values at each reservoir allows for validation of reservoir operating rules
2. Reservoir outflow (AF; data key: \_otf) – daily outflow values at each reservoir allows for validation of reservoir operating rules. Note – outflow values are also needed to calculate input data series for ‘gains’

Additional input data:

1. Sacramento gains-to-delta (AF; data key: SAC\_gains) – daily values of the gains (or losses) of flow between the sum of all Sacramento River watershed’s (SHA; ORO; FOL; YRS) downstream gauges and the Sacramento river delta inflow (at Rio Vista).
2. San Joaquin gains-to-delta (AF; data key SJ\_gains) - daily values of the gains (or losses) of flow between the sum of all Sacramento River watershed’s (NML; DNP; EXC; MIL) downstream gauges and the San Joaquin river delta inflow (at Vernalis). Note: the Millerton downstream gauge here is taken to be the releases from the Mendota pool, but the downstream gauge to manage Millerton reservoir operations is a gauge at Gravelly Ford. This ‘disconnect’ between Millerton releases and flow on the San Joaquin is because we don’t have an operational model of agricultural decisions at the Mendota Pool (i.e., taking surplus flows from Millerton and using them rather than releasing them at the Mendota Pool). Releases at the Mendota pool are usually very small but we’ve done to testing as to the impact of this assumption.
3. Gains from the ‘Eastside Streams’ (AF; data key: EAST\_gains) – flow into the delta from the area including Consumnes, Mokelumne, and Calaveras Rivers. There are reservoirs on the Mokelumne and Calaveras that are not currently simulated.
4. Delta gains (AF; data key: delta\_depletions) – increase or decrease in flow between the Delta inflow measurements and delta outflow measurements
5. Old & Middle River flow (AF; data key: OMR) – flow on the Old & Middle River, including negative flows for pumping. In the calibration data set, this data series begins in 2008 – when it started to be recorded because of new environmental legislation based on the flow here. In ‘simulation/projection’ mode, the flow can be estimated using a linear combination of calculated San Joaquin inflow and pumping at Tracy & Jones (i.e., estimating this data series for future scenarios has to be done within the model from calculated outputs – not as part of preprocessing).

Additional calibration data:

1. Delta pumping, Tracy (AF; data key: TRP\_pump) – daily pumping at the Tracy pumping plant is used to validate Central Valley Project delta pumping
2. Delta pumping, Banks (AF; data key: HRO\_pump) daily pumping at the Banks pumping plant is used to validate State Water Project delta pumping
3. San Luis storage, State (tAF; data key: SLS\_storage) – daily storage in the state portion of San Luis Reservoir – note: this data is recorded monthly, daily values simply repeat the monthly storage value for every day of that month
4. San Luis storage, Federal (tAF; data key: SLS\_storage) – daily storage in the federal portion of San Luis Reservoir – note: this data is recorded monthly, daily values simply repeat the monthly storage value for every day of that month
5. South Bay Aqueduct pumping (AF; data key: SOB\_pump) – daily pumping from the California Aqueduct onto the South Bay Aqueduct, these values are used as demands for South Bay SWP contractors
6. Central Coast Aqueduct pumping (AF; data key: LAP\_pump) – daily pumping from the California Aqueduct onto the Central Coast Aqueduct, these values are used as demands for Central Coast SWP contractors
7. Southern California pumping (AF; data key: EDM\_pump) – daily pumping from the California Aqueduct upwards at the Edmonston pumping plant, these values are used as demands for Southern California SWP contractors
8. Pre-processing
9. Class initialization –
   1. Reservoirs – initialize\_northern\_res() & initialize\_southern\_res(), creates the reservoir objects used in the northern and southern model, respectively. Reservoirs require pre-processing to develop daily regressions between observed snowpack and annual flow (for making reservoir inflow projections). Initialize reservoir objects with .json parameter files: see reservoir/read\_me\_reservoir.txt for detailed explanation.
   2. Delta – initialize\_delta\_ops(), creates the expected releases needed to meet delta outflow requirements throughout the year. These are then sent back to the reservoir classes to calculate estimations of expected releases for instream flows and delta requirements from each reservoir. This information is used in turning reservoir inflow projections into ‘available storage’ – storage that can be released for exports. Initialize delta object with .json parameter files: see delta/read\_me\_delta.txt for detailed explanation.
   3. District – initialize\_water\_districts(), Initialize district objects with .json parameter files: see district/read\_me\_district.txt for detailed explanation. Also creates lists of district objects and ‘key dictionaries’ – holding district objects in a list indexed to their three letter ‘key’.
   4. Contract – initialize\_sw\_contracts() Initialize contract objects with .json parameter files: see contract/read\_me\_contract.txt for detailed explanation. Also creates lists of contract objects and ‘key dictionaries’ – holding contract objects in a list indexed to their three letter ‘key’.
   5. Waterbank – initialize\_water\_banks() Initialize waterbank objects with .json parameter files: see waterbank/read\_me\_district.txt for detailed explanation. Also creates lists of waterbank objects
   6. Canal – initialize\_sw\_contracts() Initialize canal objects with .json parameter files: see canal/read\_me\_canal.txt for detailed explanation. Also creates lists of canal objects
10. create\_object\_associations() – creates a number of dictionaries that link objects from different classes. These dictionaries are part of the model class (southern instance)
    1. model.canal\_district – dictionary indexed with canal names. Each index points to a list that has district, waterbank, and other canal objects in the order in which they occur on a canal. If canal object B is on a list for canal A (the list is indexed to canal A), then canal object A will appear on the list indexed to canal B. In this way, the canals are ‘linked’, and can be searched through (when finding district/bank demands and deliveries) in a way that recreates the physical connectivity of the canal interconnections. After this dictionary is created, a number of canal object variables are created that rely on the length of the dictionary lists (i.e., if there are seven objects on the canal list for the cross-valley canal, then there are seven nodes on that canal and we need variables for flow, turnout, etc. on that canal with seven index locations)
    2. model.canal\_priority – also a dictionary with indices representing canal names, but the lists pointed to by those index contain canal objects only. These are the canals that have ‘priority’ to spill uncontrolled (flood) water onto the indexed canals. For instance, flood water from the Friant-Kern canal has priority to spill into the Arvin-Edison Canal, but not the Cross-Valley Canal. Only if all demands have been met on ‘priority’ canals will flood water be spilled into non-priority canals.
    3. model.reservoir\_contract – dictionary that uses reservoir keys as indices for object lists. Each list has a number of contract objects, which are the surface water contracts held in the given reservoir
    4. model.contract\_reservoir – same as model.reservoir\_contract, but reverse (contracts are indices for lists of reservoir objects).
    5. district.reservoir\_contract – dictionary for each district that uses reservoir keys as indices for a 0/1 variable. If the key = 1, then that district has a contract at the given reservoir. If the key = 0, the district does not
    6. model.canal\_contract – dictionary that uses canal keys as indices for contract object lists. The lists represent all the contracts that originate using the indexed canal
    7. model.reservoir\_canal – dictionary that uses reservoir keys as indices for canal object lists. The lists represent the canals that are directly connected to the indexed reservoir
11. Pre-processing functions – there are a number of functions that initialize ‘state’ variables, used to forecast state variables that are important in model decision-making. Calculations are performed before the simulation loop to save run-time
    1. Water type indices – find\_running\_WYI() based on flow inputs, calculate the water year indices for the eight river index (ERI), Sacramento River index (SRI), and San Joaquin River index (SJI), used for determining environmental flows
    2. Delta gains – predict\_delta\_gains() develop a regression between full-natural flow into all northern reservoirs and the uncontrolled ‘gains’ to the delta. This is used in predicting delta pumping for decision-making at San Luis Reservoir
    3. Canal demands – find\_all\_triggers() & find\_flood\_trigger(), determine the ability to receive flood water from each southern model reservoir. The flood water space is calculated only for districts that have a contract with that specific reservoir. The maximum flood demands are used to help make decisions about when to start spilling water for flood control (before the reservoir reaches the top of the flood control stage, so that water that can’t be taken by a contractor isn’t ‘wasted’
    4. Carryover – find\_initial\_carryover() – the simulation starts with storage in each reservoir – based on this initial storage, set district carryover volumes and contract allocations (before pumping or flow adds to allocations)
    5. Recovery – init\_tot\_recovery() – based on their ownership shares at in-leiu and direct water banks, calculate the total recovery capacity ‘owned’ by each district
    6. Urban Demand – project\_urban() regression between SWP delta pumping and pumping at each of the ‘branches’ on the California aqueduct associated with the urban SWP contractors. This regression is used to estimate urban district demands in the fall/winter, rather than rely on pumping data (for better recharge/recovery decision-making). This can also be used when the model runs in ‘projection’ mode.

1. Model Classes

Northern Model Classes:

Input timeseries data for each watershed and rules for reservoir operations are largely stored in a ‘reservoir’ class, and delta rules/additional input timeseries data is stored in a ‘delta’ class. Data flows between these two classes inform how water is released from storage and pumped through the delta.

1. Reservoir

Data timeseries for the reservoir class include snowpack, inflow, full natural flow, downstream gains, precipitation, and evaporation. Parameters for reservoir objects are set through the KEY\_properties.json files in the /reservoir folder, with an explanation in reservoir/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **rights\_call()** | Determine if reservoir *gains* are inflow to delta or in-basin environmental requirements requiring reservoir release  *reservoir.gains\_to\_delta sent to delta class* |
| **release\_environmental()** | Calculate release for flood control and in-basin environmental requirements downstream of the reservoir. *reservoir.envmin, reservoir.din, sent to delta class* |
| **find\_available\_storage()** | Using snow-flow regression & expected env. releases, make seasonal risk-based predictions of water available for export. *reservoir.available\_storage sent to delta class* |
| **find\_flow\_pumping()** | Based on short-term flow projections (flow share), calculate (i) total days until reservoir fills (expected); (ii) average release rate to avoid reservoir fill; (iii) peak volume expected above flood capacity |
| **step**() | Water balance on reservoir, accounting for releases for flood control, environmental flows, delta inflow/outflow req., and releases for export  *reservoir.sodd received from delta class* |

***Output***:

*Daily timeseries data:* reservoir\_results\_no.csv – daily storage, top of flood pool, available storage, and outflow from each reservoir. Column headers are reservoir keys + timeseries type

1. Delta

Data timeseries for the delta class include Sacramento River gains (gains between the gauged point on each tributary and the Rio Vista delta inflow gauge); San Joaquin River gains (gains between the gauged point on each tributary and the Vernalis delta inflow gauge); Eastside Streams gains (delta gains that are not counted through Vernalis or Rio Vista); Delta depletions (losses/gains between delta inflow and delta outflow measurements); Old/Middle River flow (can be estimated within model by total flow at Vernalis – does not need to be included in future scenario data generation). Parameters for the (single) delta object are set through the Delta\_properties.json files in the /delta folder, with an explanation in delta/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **calc\_vernalis\_rule()** | Calculates the release from SJ reservoirs needed to meet delta inflow requirements at Vernalis  *reservoir.gains\_to\_delta and reservoir.envmin taken from the reservoir class;*  *reservoir.din sent to the reservoir class* |
| **assign\_releases()** | Calculate the percentage contribution of each SAC reservoir to the delta inflow requirements at Rio Vista. *reservoir.available\_storage, reservoir.S received from reservoir class* |
| **calc\_rio\_vista\_rule()** | Calculates the release from SAC reservoirs needed to meet delta inflow requirements at Rio Vista  *reservoir.gains\_to\_delta and reservoir.envmin taken from the reservoir class;*  *reservoir.din sent to the reservoir class* |
| **calc\_outflow\_release()** | Calculates the release from SAC reservoirs needed to meet delta outflow requirements  *reservoir.gains\_to\_delta, reservoir.envmin, and reservoir.din taken from the reservoir class;*  *reservoir.dout sent to the reservoir class* |
| **find\_max\_pumping**() | Calculates the maximum allowed pumping at the delta, based on NMFS BiOPS and D1641 rules |
| **meet\_OMR\_rules()** | Find if flow on the Old/Middle River requires reductions in maximum allowed pumping |
| **find\_release**() | Use available delta storage & current season to find if pumps will run at their maximum level; projects total annual SWP & CVP pumping  *reservoir.available\_storage received from reservoir class*  *delta.forecastCVPstorage & delta.forecastSWPstorage sent to southern model* |
| **hypothetical\_pumping**() | Check if there is enough space at San Luis to pump at desired levels, if not, calculate how much pumping would have happened if there was space |
| **calc\_flow\_bounds()** | Find releases needed to pump at the desired level – either the max calculated in find\_release or at the ‘tax free rate’ |
| **distribute\_export\_releases()** | Take total releases needed to operate pumps at desired level and distribute them among the project storage reservoirs. Incorporate potential flood avoidance releases from reservoir.find\_flow\_pumping  *reservoir.min\_daily\_uncontrolled and reservoir.available\_storage received from reservoir class*  *reservoir.sodd sent to reservoir class* |
| **step()** | Takes all releases from reservoirs and calculates total pumping for each project  *reservoir.sodd received from reservoir class*  *delta.HRO\_pump and delta.TRP pump sent to southern model* |

***Output***:

*Daily timeseries data:* reservoir\_results\_no.csv – daily CVP & SWP pumping, CVP & SWP allocation projections, X2 calculations, Sacramento index projections, and San Joaquin index projections. Column headers are delta keys + timseries type

Southern Model Classes:

In the southern model, the flow of information generally moves from Reservoir > Contract > District > Waterbank > District > Canal > Reservoir, forming an information loop that allows for the matching of supply and demand through simulated rules instead of an optimization process

1. Reservoir

The reservoir class operates in the same fashion as in the northern model, except the find\_flow\_pumping() function occurs after the step() function. Note – San Luis reservoir is a reservoir class object, but it does not have many of the same features as the other reservoirs that are a part of a watershed. Some functions for San Luis are slightly different, and it does not use all the reservoir functions (e.g. release\_environmental) b/c it is operated using a different class of rules. For instance – instead of ‘available storage’ projections (b/c there are no natural inflows to San Luis), forecastCVPallocation/forecastSWPallocation (from northern model) are used to project contract allocations. Parameters for other reservoir objects (Millerton, Isabella, Kaweah, and Success) are set through the KEY\_properties.json files with an explanation in reservoir/readme.doc

***Output***:

*Daily timeseries data:* contract\_results.csv – daily storage, top of flood pool, available storage, and outflow from each reservoir. Column headers are reservoir keys + timeseries type

1. Contract

The contract class uses snowpack-based flow projection timeseries as input which are generated within the model in the reservoir class (or, in the case of delta contracts, from the delta class) to generate contract allocation projections. Multiple contracts can be stored within one reservoir, sharing the flow forecast based on contract priority/seniority. Parameters for contract class objects are set through the KEY\_properties.json with an explanation in contract/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **calc\_allocation()** | Finds the total volume that is allocated under the contract for the year, based on reservoir available storage & contract seniority. This allocation value includes water that has been delivered on that contract already – so it is an estimate of the full value of the contract (in that year), not just the additional future projection.  *reservoir.available\_storage received from reservoir class*  *contract.allocation sent to the district class* |
| **find\_storage\_pool()** | Find the total ‘storage pool’ of a contract, which refers to the total contract amount that is currently available – either already having been delivered to districts/waterbanks or currently held in reservoir storage  *reservoir.S, received from reservoir class*  *contract.storage\_pool, sent to district class* |

***Output***:

*Daily timeseries data:* contract\_results.csv – Different contract ‘colors’ are stacked so that the final value in a series is equal to the series total (for plotting in a stacked area chart). Contract colors include daily contract deliveries, carryover deliveries, flood deliveries, and turnback pool (intra-contract trading) deliveries. Column headers are contract keys + timeseries type

*Annual timeseries data:* contract\_results\_annual.csv – Annual values are not stacked and represent the total value delivered in a year from the different contract ‘colors’ - contract deliveries, carryover deliveries, flood deliveries, and turnback pool (intra-contract trading) deliveries. Column headers are contract keys + timeseries type

1. District

The district class uses timeseries of contract allocations and crop et, generated within the southern model in the contract class and the crop class, to develop projections for individual supplies and water demands. These projections are used as triggers for various decisions about when to request different ‘colors’ of water. Parameters for district class objects are set through the KEY\_properties.json files with an explanation in district/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **Demand Calculations – find\_baseline\_demands(); calc\_demand(); find\_pre\_flood\_demand; get\_urban\_demand()** | Demand is the basic attribute of a district class object – it is calculated here in monthly (from irrigation demand data); daily (for daily simulation) and annual (for calculating thresholds to determine whether or not to  recharge/recover/carryover/turnback water that the district owns. |
| **Supply information – update\_balance()** | Water supplies for individual districts take allocation & storage information from the contract class to give districts a running estimate of how much water they thing they have left (projected\_supply). These values are used with annual demand estimations to calculate thresholds with which districts make decisions |
| **Decisionmaking – open\_recovery(); open\_recharge(); set\_turnback\_pool(); make\_turnback\_purchases();calc\_carryover()** | These are the functions where districts take the supply/demand information and make decisions about recharge/recovery/turnback/carryover |
| **Canal Deliveries –**  **find\_node\_demand(); find\_node\_output(); set\_request\_constraints(); set\_demand\_priority(); find\_leiu\_priority\_space(); set\_deliveries();** | These functions are called when we loop through a canal and find a district node or a waterbank node (districts that are waterbank members call these functions). The determine what the maximum demands at each node are, how much the district would like to request under each contract type, what the priority of the district’s request would be, and the water deliveries themselves. |
| **Account Adjustment – give\_paper\_trade(); get\_paper\_trade(); direct\_recovery\_delivery(); adjust\_accounts(); adjust\_bank\_accounts(); adjust\_recovery(); absorb\_storage()** | When water is delivered, we need to record that delivery and update waterbank accounts, paper trade accounts, contract accounts, and district variables like demand and recharge space |
| **Recording State Variables – reset\_recharge\_recovery(); accounting(); accounting\_banking\_activity();**  **accounting\_leiubank(); accounting\_as\_df(); annual\_results\_as\_df(); bank\_as\_df()** | Record state variables like contract allocations, deliveries, recharge, and water bank accounts as Data Frame timeseries for export to CSV output files |

***Output***:

*Daily timeseries data:* district\_results.csv – Values are stacked so that the final value in a series is equal to the series total (for plotting in a stacked area chart). Different types of supplies (remaining projected allocation, ‘paper trade’ balance, and total carryover water) are stacked in positive numbers and deliveries from various contract ‘colors’ (contract, in-leiu bank deliveries, deliveries from out-of-district groundwater banks, private well pumping, in-leiu bank pump-out (to banking partners), recharge deliveries, and uncontrolled deliveries) are stacked negatively. Column headers are district keys + timeseries type

leiu\_results.csv (for districts that operate in-leiu groundwater banks) – Total banking accounts of the districts’ banking partners are stacked so that the values add up to be the total in-leiu accounts for all members at the bank. Column headers are district keys (the district that operates the leiu-bank) + \_district keys (the in-leiu account holder)

*Annual timeseries data:* district\_results\_annual.csv – Annual values are not stacked and represent the annual deliveries to a district from the different contract ‘colors’: contract deliveries, carryover deliveries, flood deliveries, and turnback pool (intra-contract trading) deliveries. Column headers are contract keys + timeseries type

leiu\_results\_annual.csv (for districts that operate in-leiu groundwater banks) – total annual change in banking accounts for each banking partner (not stacked). Column headers are contract keys + timeseries type

1. Waterbank

The waterbank class uses timeseries of district object decisions, generated within the southern model, to open capacity in recharge and recovery infrastructure. Parameters for waterbank class objects are set through the KEY\_properties.json files with an explanation in waterbank/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **find\_node\_demand()** | Finds either the recharge or recovery capacity available at a waterbank node |
| **find\_priority\_space(); set\_demand\_priority(); set\_deliveries()** | Divides capacity in the waterbank among individual members in the bank, by ownership priority and delivers the water to storage in the bank |
| **adjust\_recovery(); sum\_storage(); absorb\_storage()** | Update water bank accounts and infrastructure use |
| **accounting(); bank\_as\_df(); annual\_bank\_as\_df()** | Record bank accounts and store as dataframe for export to CSV |

***Output***:

*Daily timeseries data:* bank\_results.csv – Total banking accounts of each bank owner are stacked so that the values add up to be the accounts for all members at the bank. Column headers are waterbank keys + \_district keys (the banking account holder)

*Annual timeseries data:* bank\_results\_annual.csv – total annual change in banking accounts for each banking partner (not stacked). Column headers are waterbank keys + \_district keys (the banking account holder)

1. Canal

The canal class uses timeseries of delivery requests from district and waterbank objects, generated within the southern model, to route water from the surface and groundwater sources, through canals and to their requested destination. Capacity, turnout, and directional constrains limit the size of the requests based on priority. Parameters for canal class objects are set through the KEY\_properties.json files with an explanation in canal/readme.doc

|  |  |
| --- | --- |
| ***Function name*** | ***Description*** |
| **check\_flow\_capacity()** | Finds remaining flow capacity at the current canal node |
| **find\_priority\_fractions(); find\_turnout\_adjustment()** | What percent of each request priority level can be filled at the canal node based on the turnout capacity |
| **update\_canal\_use();** | Check to make sure each canal node has enough flow capacity for requested flows, and if not, update flows on previous canal nodes and re-calculate capacity/priority sharing rules |
| **find\_bi\_directional();** | If canal is bi-directional, check if there is flow on the canal and lock the flow direction for the duration of a timestep |
| **accounting(); accounting\_as\_df()** | Store flow and turnout timeseries at each canal node as dataframes for export to CSV. |

***Output***:

*Daily timeseries data:* canal\_results.csv – Daily flow at each canal node (not stacked). Column headers are canal keys + \_node keys (district, waterbank, or other canal, depending on the node)