Fluid Mechanics Project

Patrick O'Boyle and Lukas Negron

Constants and Model

Constants Dynamic Display

In[ ]:=
```
(* Testing method - for initializing resetting *)
resetGlobalConstants [];
```

In[ ]:=
```
(* Dynamic wrapping of constants so we can see changes *)
Dynamic[globalConstants ]
```

Out[ ]= globalConstants

MasterPathList Dynamic Display + simplePaths View

In[ ]:=
```
(* For initalization of path list and resetting *)
resetMasterPathList [];
```

In[ ]:=
```
(* Dynamic wrapping allows us to observe changes *)
Dynamic[masterPathList ]
```

Out[ ]= masterPathList

In[ ]:=
```
(* Execute to show current connection view *)
(* A view of the paths, the prev path, and the next path *)
updateSimplePaths [] // TableForm
```

Out[ ]//TableForm=

{}

Calculations

Constants

In[ ]:=
```
(* List of modifiable constants. Used for various calculations *)

resetGlobalConstants [] :=
  globalConstants = <|
    "gravity" → Quantity[9.8, "Meters"/("Seconds")^2],
    "environmentTemp " → Quantity[4, "CelsiusHeatUnits "],
    "densityWater " → Quantity[1000, "Kilograms "/("Meters")^3](* Note:
     can implement changing density of water with respect to temperature *)
    |>;
```

In[ ]:=
```
(* used so that the constants are initalized when running intialization cells *)
resetGlobalConstants [];
```

In[ ]:=
```
(* Returns value for a constant. Can
 be used to change a constant's value as well *)

returnConstant [constant_] :=
     globalConstants [constant]
```

Ex : returnConstant["gravity"] = newValue

Pipe

In[ ]:=
```
(* Calculating the cross-sectional area of a pipe using its diameter *)

(* diameter Units: meters *)

(* Output Units: meters^2 *)
calcCrossArea [diameter_] :=
     N[Pi/4 * diameter^2]
```

In[ ]:=
```
calcCrossArea [Quantity[4, "Meters"]]
```

Out[ ]= $12.5664\ \mathrm{m}^2$

```
In[ ]:= (* Calculating the velocity of liquid through a pipe using the flowRate in and the diameter
(* flowRateIn = flowRateOut *)
(* flowRate = Velocity * CrossArea *)

(* flowIn Units: m^3/sec *)
(* diameter Units: m *)


(* Output Units: m/sec *)

calcPipeVelocity [flowIn_, diameter_] :=
    flowIn / calcCrossArea [diameter]
```

```
In[ ]:= calcPipeVelocity [Quantity[15, "Meters"^3 / "Seconds"], Quantity[4, "Meters"]]
```

```
Out[ ]= 1.19366 m/s
```

Pump

```
In[ ]:= (* Calculating the input flowRate a system with a pump *)
(* W_elec = (flowRate * densityOfLiquid * acclerationOfGravity * headValuePump) / efficiency *)
(* hPump →
 Z_B-Z_A (distance from free-surfaces of input reservoir to output reservoir *)

(* watts Units: watts *)
(* densityOfLiquid Units: kg/m^3 *)
(* efficiency: number on (0, 1] interval, a percentage *)
(* hPump: m *)

(* Output Units: m^3/sec *)

calcFlowRateForPump [watts_, densityOfLiquid_, efficiency_, hPump_] :=
     (watts * efficiency) / (densityOfLiquid * globalConstants ["gravity"] * hPump)
```

```
In[ ]:= pumpFlowRateTestOut = UnitSimplify [calcFlowRateForPump [Quantity[1000, "Watts"],
    globalConstants ["densityWater"], 0.75, Quantity[5, "Meters"]]]
```

```
Out[ ]= 0.0153061 W/Pa
```

```
In[ ]:= UnitConvert [pumpFlowRateTestOut, "Meters"^3 / "Seconds"]
```

```
Out[ ]= 0.0153061 m^3/s
```

In[ ]:=
```
(* Calculates a pump's capacity to do work *)
(* A is the input, B is the output *)
(* pressureA/B is calculated using the calcPressureReservoir function *)
(* pressureA/B is 0 if exposed to atmosphere *)
(* heightA < heightB *)

(* pressureA/B Units: pascals *)
(* densityLiq Units: kg/m^3 *)
(* heightA/B Units: m *)

(* Output Units: m *)

calcPumpHead[pressureA_, heightA_, pressureB_, heightB_, densityLiq_] :=
  heightB - pressureA/(densityLiq * globalConstants["gravity"]) +
    pressureB/(densityLiq * globalConstants["gravity"]) - heightA
```

In[ ]:=
```
calcPumpHead[Quantity[0, "Pascals"], Quantity[5, "Meters"],
 Quantity[0, "Pascals"], Quantity[10, "Meters"], globalConstants["densityWater"]]
```

Out[ ]= 5. m

Reservoir

In[ ]:=
```
(* Calculates the pressure exerted on a reservoir's free surface *)

(* densityOfUpperFluid Units: kg/m^3 *)
(* ceilingDis Units: m *)
(* gaugePressure Units: pascals *)

(* Output Units: pascals *)


calcPressureReservoir[densityOfUpperFluid_, ceilingDis_, gaugePressure_] :=
  (densityOfUpperFluid * globalConstants["gravity"] * ceilingDis) + gaugePressure
```

In[ ]:=
```
calcPressureReservoir[globalConstants["densityWater"],
 Quantity[1, "Meters"], Quantity[1000, "Pascals"]]
```

Out[ ]= 10 800. Pa

Parallel Network

In[ ]:=
```
(* Calculates the flow rate for each horizontal
 component of a parallel network across n segments *)
(* Also works for split series networks *)

(* flowIn Units: m^3/sec *)
(* diameters list Units: m -pipe diameters *)

(* Output Units: m^3/sec -list of flow rates for each segment *)
(* NOTE- If diameters is a list of Nulls,
the function will return data as if they were all in a 1:1 ratio.
   This allows us to test processing for branches of undefined diameters. *)

calcPartialFlowRates[flowIn_, diameters_] := Module[
  {segments = Length[diameters],
   unitFlow
  },
  unitFlow = flowIn / Total[diameters];

  (* Output the partial flows across each sequence *)
  Table[
   diameter * unitFlow
   , {diameter, diameters}
  ]

 ]
```

In[ ]:=
```
testDiametersList2 = Table[Quantity[diameter, "Meters"], {diameter, 2, 5}]
calcPartialFlowRates[Quantity[10, "Meters"^3 / "Seconds"], testDiametersList2]
```

Out[ ]= $\{2\ \text{m},\ 3\ \text{m},\ 4\ \text{m},\ 5\ \text{m}\}$

Out[ ]= $\left\{\frac{10}{7}\ \text{m}^3/\text{s},\ \frac{15}{7}\ \text{m}^3/\text{s},\ \frac{20}{7}\ \text{m}^3/\text{s},\ \frac{25}{7}\ \text{m}^3/\text{s}\right\}$

Differential Pressure

```
In[ ]:= (* Returns the deltaP, or change in pressure between
          two points with differing velocities and heights *)
        calcDiffPress[densityLiq_, vel1_, vel2_, h1_, h2_] :=
          densityLiq * (vel2^2 - vel1^2) / 2 + densityLiq * globalConstants["gravity"] * (h2 - h1)
```

```
In[ ]:= UnitConvert[calcDiffPress[globalConstants["densityWater"],
          Quantity[5, "Meters"/"Seconds"], Quantity[10, "Meters"/"Seconds"],
          Quantity[2, "Meters"], Quantity[2, "Meters"]], "Pascals"]
```

```
Out[ ]= 37 500. Pa
```

Components

Infinite Input Reservoir

```
In[ ]:= (* path- the path to add the input reservoir to
           heightOffGround - how high the reservoir is off the gruond
           reservoirHeight -
          the height of the actual reservoir (influences pressure calculations)
           pressureGauge - whether or not the container
           is pressurized (* 0 if exposed to atmosphere *)
           densityOfBottomFlu - the density of the fluid being pumped into the system*)

        (* Because this is an infinite reservoir, the volumes do not decrease,
        so the upper liquid and the bottom liquid will both fill half the
         container at all times. This is relevant for pressure calculations *)

        mapNewInfInputReservoir[path_,
          sizeHeight_, pressureGauge_, densityBottomFlu_] := Module[
          {startSeq = path,
           reservoirList = {{}}
          },

          reservoirList = {{"InfIR", sizeHeight, pressureGauge, densityBottomFlu}};

          masterPathList[startSeq, "start"] = reservoirList;
         ]
```

Unbounded Output Reservoir

In[ ]:=
```
(* Comments for inf input reservoir explain the parameters
  vol- monitors how much water is in the reservoir
  The reservoir is unbounded, so grad was put in so
 we have a method for measuring some amount of water put in
*)

mapNewUnboundedOutputReservoir [
  path_, sizeHeight_, pressureGauge_, vol_] := Module[
  {endSeq = path,
   reservoirList = {}
  },

  reservoirList = {"UnOR", sizeHeight, pressureGauge, vol};

  masterPathList[endSeq, "end"] = reservoirList;
 ]
```

Infinite Pump

In[ ]:=
```
(* Prerequisite for adding a pump is to have an input reservoir *)
(* Set the input flowRate into system. No headvalue or electricity required *)

addInfPump[path_, flowRate_] := Module[
  {startSeq = path,
   pumpList = {}},

  pumpList = {"InfPu", flowRate};
  AppendTo[masterPathList[startSeq, "start"], pumpList];
 ]
```

In[ ]:=
```
(* modify the system's starting flow rate *)
modifyInfPump[path_, flowRate_] := Module[
  {startSeq = path,
   pumpList = {}
  },

  (* retrieve pump data and replace the flow rate reading *)
  pumpList = masterPathList[startSeq, "start"][[2]];
  pumpList[[2]] = flowRate;

  (* remove old pump data and add the changed pump data *)
  masterPathList[startSeq, "start"] =
   ReplacePart[masterPathList[startSeq, "start"], 2 → pumpList];
 ]
```

Pipes

In[ ]:=
```
(* adds a pipeNum number of pipes with diameter pipeD to a seq *)
addPipes[path_, pipeNum_, pipeD_]:= Module[{pipeList},
    pipeList = Table[{"P", pipeD}, pipeNum];
    masterPathList[path,"parts"] = Join[masterPathList[path, "parts"], pipeList];
]
```

Control Systems

In[ ]:=
```
(* adds a valve, which is a special case pipe, to a seq *)
addValve[seqKey_, maxD_] := Module[{localSeqKey = seqKey},
    AppendTo[masterPathList[localSeqKey, "parts"], {"V", maxD, maxD}]
  (* the second value in this list is the current diameter of the component,
  while the third value stays the maximum *)
 ]
```

In[ ]:=
```
(* changes a valve's diameter using which seqence it is a part of, which valve it is in th
changeValveDiameter [seqKey_ , valveIndex_ , newD_]:= Module [{newValveList },
    (* sets the diameter to the new diameter *)
    If[newD ≤ masterPathList [seqKey , "parts"][[valveIndex ,3]],
        newValveList = masterPathList [seqKey , "parts"][[valveIndex ]];
        newValveList [[2]] = newD ;
        masterPathList [seqKey , "parts"] = ReplacePart [masterPathList [seqKey , "parts"], valve
    ];
]
```

In[ ]:=
```
(* adds a gate , another special case pipe , to a sequence *)
addGate [seqKey_ ,d_]:= Module [{localSeqKey =seqKey },
    AppendTo [masterPathList [localSeqKey ,"parts"],{"G", d, d}] (* the second value is the curr
]
```

In[ ]:=
```
(* negates a gate , as in closes it if it is open or opens it if it is closed *)
negateGate [seqKey_ , gateIndex_ ] := Module [{newGateList },
    (* makes a copy map *)
    newGateList = masterPathList [seqKey , "parts"][[gateIndex ]];
    (* negates the gate in the copy map *)
    If[newGateList [[2]]=== newGateList [[3]],
        (* true *)
        newGateList [[2]]=0 ,
        (* false *)
        newGateList [[2]]=newGateList [[3]];
    ];
    (* maps the copy map to the part key *)
    masterPathList [seqKey , "parts"] = ReplacePart [masterPathList [seqKey , "parts"], gateIndex
]
```

General Input Reservoir

In[ ]:=
```
(* Similar to infinite input reservoir but has a finite amount
   of fluid available to transfer into the system and the ratio
   of the upper fluid to the lower fluid isn't necessarily 1:1  *)

(* NOTE- this component features a finite amount of fluid but it isn't
   used in the current setup for general water transfer. General input
   reservoirs are currently an infinite source but the pressure and
   upper fluid comes into play for calculating required head value *)

mapNewGenInputReservoir[path_, sizeHeight_, ceilDis_, pressureGauge_,
   volBottom_, volTop_, densityBottomFlu_, densityUpperFlu_] := Module[
   {startSeq = path,
    reservoirList = {{}}
   },

   reservoirList = {{"GenIR", sizeHeight, ceilDis, pressureGauge,
       volBottom, volTop, densityBottomFlu, densityUpperFlu}};

   masterPathList[startSeq, "start"] = reservoirList;
 ]
```

In[ ]:=
```
(* retrieves the bottomLiqDensity from seq1 for other operations *)
getLiqDensity[isInf_] :=
     (* works for both infinite and general setups *)
     If[isInf,
          (* inf setup *)
          masterPathList["seq1", "start"][[1, 4]],
          (* gen setup *)
          masterPathList["seq1", "start"][[1, 7]]
     ]
```

General Output Reservoir

In[ ]:=

```
(* Similar to unbounded output reservoir but has a maxVolume *)
(* Also features the ability to add an upper fluid to the reservoir *)

mapNewGenOutputReservoir [path_, sizeHeight_, ceilDis_, pressureGauge_,
  volBottom_, volTop_, maxVol_, densityUpperFlu_] := Module[
  {endSeq = path,
   reservoirList = {}
  },

  reservoirList = {"GenOR", sizeHeight, ceilDis,
    pressureGauge, volBottom, volTop, maxVol, densityUpperFlu};

  masterPathList[endSeq, "end"] = reservoirList;
 ]
```

Change Gauge Pressure

In[ ]:=
```
(* changes the gauge pressure on a reservoir *)
(* uses helper functions to automatically determine whether it's changing
 the input reservoir or an output, or if there's nothing to change *)
changeGaugePressureGen[seq_, newPress_] := Module[
  {seqKey = seq,
   hasInput = (seq == "seq1"),
   hasOutput = hasOutputR[seq]
  },

  (* Replace the list with the updated
   one. Pressure is at index 4 of the reservoirList *)

  If[hasInput,
       (* is inputRes *)
       masterPathList[seqKey, "start"] =
    ReplacePart[masterPathList[seqKey, "start"], {1, 4} → newPress],
   (* Parted[1,4] because pump could be at index 2*)
  ];

  If[hasOutput,
       (* is outputRes*)
       masterPathList[seqKey, "end"] =
    ReplacePart[masterPathList[seqKey, "end"], 4 → newPress]
  ];

 ]
```

In[ ]:=
```
(* true if the seq has an output reservoir *)
hasOutputR[seq_] := Module[
  {seqKey = seq,
   possibleOutputR
  },

  possibleOutputR = masterPathList[seqKey, "end"];

  Return[ListQ[possibleOutputR]]
 ]
```

General Pump

In[ ]:=

```
(* Prerequisite  for adding a pump is to have an input reservoir *)
(* Similar  to inf pump but has a head
 value and supplied  electricity  that changes *)
(* Head value and electricity  will determine  flow rate*)

addGenPump[path_, efficiency_, watts_, hPump_] := Module[
  {startSeq = path,
   fluDensity,
   flowRate,
   pumpList = {}
  },

  (* retrieve  fluid density  from the input
   reservoir  and use it to calculate  the pump's flow rate *)
  fluDensity = masterPathList[startSeq, "start"][[1, 7]];
  flowRate = calcFlowRateForPump[watts, fluDensity, efficiency, hPump];
  flowRate = UnitConvert[flowRate, "Meters"^3 / "Seconds"];

  pumpList = {"GenPu", flowRate, watts, hPump};
  AppendTo[masterPathList[startSeq, "start"], pumpList];
 ]
```

In[ ]:=
```
(* modify the pump characteristics , which will affect its flow rate *)
modifyGenPump [path_ , efficiency_ , watts_ , hPump_] := Module[
  {startSeq = path ,
   pumpList ,
   fluDensity ,
   flowRate
  },

  (* retrieve the pump component *)
  pumpList = masterPathList [startSeq , "start"][[2]];

  (* replace the previous pump data *)
  pumpList [[3]] = watts ;
  pumpList [[4]] = hPump ;
  fluDensity = masterPathList [startSeq , "start"][[1, 7]];
  flowRate = calcFlowRateForPump [watts , fluDensity , efficiency , hPump];
  pumpList [[2]] = UnitConvert [flowRate , "Meters"^3 / "Seconds"];

  (* remove old pump data and add the changed pump data *)
  masterPathList [startSeq , "start"] =
   ReplacePart [masterPathList [startSeq , "start"], 2 → pumpList]
 ]
```

Model

MasterPathList Functions + Keys + simplePaths

In[ ]:=
```
(* Holds the current list of paths- all the sequences , splits and merges *)
(* All paths have attributes ("attr"), components ("parts"),
where they start , and where they end *)

resetMasterPathList [] :=
  masterPathList = <||>;
```

In[ ]:=
```
(* Prints a key and its corresponding structure . Makes for an easier reading of the masterP
printMPL []:=
    Do[
        Print [key , " ", masterPathList [key]];
    ,{key , Keys [masterPathList ]}
    ];
```

In[ ]:=
```
(* Path key creation. Path keys are named
 based on when it was created and its type *)

createPathKey[pathType_, num_] := Module[
  {nextCompIndex = getLastKeyNum[] + 1},
  (* Find how many keys we have and add 1 to generate new key *)

  (* Return the type combined with the # it is. Ex: seq2, spl5, m4 *)
  If[SameQ[num, Null],
    (*True*)
    (* If a num isn't given, refer to the masterPathList *)
    Return[pathType <> ToString[nextCompIndex]],
    (*False*)
    (* If a specific num is given for the path key, use it *)
    Return[pathType <> ToString[num]]
   ];
 ]
```

In[ ]:=
```
(* The number the last path key should contain from the masterPathList *)

getLastKeyNum[] := Return[Length[masterPathList]]
```

In[ ]:=
```
(* Cut the number off the base string to get the type of path it is *)
getPathType[path_] := StringSplit[path, RegularExpression["[0-9]"]][[1]]
```

In[ ]:=

```
(* returns a list of the pathKey,
the prevKey and the nextKey. This allows us to view the
 connections between paths without the excess information *)
updateSimplePaths[] := Module[
  {pathType,
   prev,
   next,
   simplePaths
  },

  simplePaths = Table[
        pathType = getPathType[key];
        prev = ToString[masterPathList[key, "start"]];
        next = ToString[masterPathList[key, "end"]];

        (* for a split's end map *)
        If[pathType == "spl",
         next = "N/a";
         ];

        (* for seq's that aren't connected to an end *)
        If[next == "Null" || StringContainsQ[next, "O"],
         next = "Output";
         ];

        (* for seqs that aren't connected to a start *)
        If[prev == "Null" || StringContainsQ[prev, "R"],
         prev = "Input";
         ];

        {key, "prev→" <> prev , "next->" <> next}

    , {key, Keys[masterPathList]}
   ];

  Return[simplePaths];
 ]
```

Path Creation

In[ ]:=

```
(* Creates a sequence *)
(* Sequences:
 The simplest path type. A line of components
   key- "seq#"
   mappings- attributes, parts, start, end
*)

createSeq[name_, startKey_] := Module[
  {seqKey = "undefinedSeqKey",
   seqMap = "undefinedMapping"
  },

  (* If a specific keyName was given,
  use it. Otherwise, do normal key creation*)
  If[SameQ[name, Null],
   (*true*)
   seqKey = createPathKey["seq", Null],
   (*false*)
   seqKey = name
  ];

  (* Sequence structure *)
  seqMap = seqKey → <|
     instantiatePathAttributes [],
     "parts" → {}, (* Changed from empty to 4 Nulls for slots in the UI *)
     "start" → startKey,
     "end" → Null
     |>;

  (* Add to the path list *)
  AppendTo[masterPathList, seqMap];

  (* set end of the one you linked. Do not setEnd if it isn't a sequence *)
  If[StringQ[startKey] && StringContainsQ[startKey, "seq"],
       setEnd[startKey, seqKey];
  ]
 ]
```

In[ ]:=

```
(* Creates a split *)
(* Splits:
 Allow for sequences to diverge and possibly merge. Branches
  are in the form {{T}, {M}, {B}}, where T is the number of sequences
 above the middle sequence, M is either empty or contains one sequence
 for the middle, and B is the number of sequence below the middle
  key- "spl#"
  mappings- attributes, start, branches
*)

createSpl[startKey_, {t_, m_, b_}, mergePairs_] := Module[
  {newSplKey = createPathKey["spl", Null],
   splMap = "undefinedMapping",
   branchKeys = createBranches[{t, m, b}],
   mergeStartKeys
  },

  (* Split structure *)
  splMap = newSplKey → <|
     "start" → startKey,
     "branches" → branchKeys
     |>;

  (* Add the split and its derivate sequences to the
   path list. Then add the merge points for the sequences *)
  AppendTo[masterPathList, splMap];
  appendSplitSequences[branchKeys];

  If[!SameQ[mergePairs, Null],
   (*true. merging occurs, so add the merges*)
   mergeStartKeys = createMergeStartKeys[branchKeys, mergePairs];
   appendMergeSequences[mergeStartKeys];
  ];

  (* set end of the one you linked *)
  If[StringQ[startKey],
       setEnd[startKey, newSplKey];
  ]
 ]
```

In[ ]:=
```
(* Creates a merge *)
(* Merges :
The point at which two or more sequences come together . Can hold multiple keys in its star
key- "mer#"
 mappings - attributes , start(multiple ), end
*)

createMer [startKeys_ ]:= Module [
{newMerKey = createPathKey ["mer", Null],
merMap = "undefinedMapping ",
newSeqKey = createPathKey ["seq", getLastKeyNum [] + 2]
},

    (* Merge structure *)
    merMap = newMerKey → <|
    "start" → startKeys ,
    "end" → newSeqKey
    |>;

    (* Links the merged sequences to the merge point *)
    setMergedSeqsEnds [startKeys , newMerKey ];

    (* Add merge to the path list *)
    AppendTo [masterPathList , merMap ];
    createSeq [newSeqKey , newMerKey ]

    (* IMPLEMENT : set the spl seq ends to this merge key *)
]
```

In[ ]:=
```
(* Called by create split. Takes a given number of t's, m's, and b's and returns the partit

createBranches [{t_, m_, b_}]:= Module[
{seqKeys = {},
numOfSplits = t+m+b,
prevKeyNum = getLastKeyNum [] + 1,
partedSeqKeys ={}
},

    (* Builds the keys for the sequences *)
    seqKeys = Table[
        i += prevKeyNum ; (* add each i to the prevKeyNum so the seq key numbers aren't dup
        createPathKey ["seq", i],
        {i, numOfSplits }];

    (* Takes the keys and partitions them *)
    partedSeqKeys = splitBranches [seqKeys , {t, m, b}];

    (* Returned partitioned list of seq keys *)
    Return [partedSeqKeys ];
]
```

In[ ]:=
```
(* Takes a list of seq keys and the number of t, m, and b splits to return the partitioned

splitBranches [seqKeys_ , {t_, m_, b_}]:=Module[
{partedSeqKeys  = {}},

    (* This takes the first t keys from seqKeys and adds it to partedSeqKeys  *)
    AppendTo[partedSeqKeys ,
        Partition[seqKeys , t, 1][[1]]
    ];

    (* This takes the first m keys after the first t keys from seqKeys and adds it to parte
    AppendTo[partedSeqKeys ,
        Partition[seqKeys , m, 1][[t+1]]
    ];

    (* This takes the last b keys from seqKeys and adds it to partedSeqKeys  *)
    AppendTo[partedSeqKeys ,
        Partition[seqKeys , b, 1][[t+m+1]]
    ];

    (* Returns the partitioned list *)
    Return[partedSeqKeys ];
]
```

In[ ]:=
```
(* Is called by create split. Takes each branch key and creates sequences for it*)

appendSplitSequences [partedSeqKeys_ ]:= Module[
{splitName  = Last[Keys[masterPathList ]],
unpartedSeqKeys  = Flatten[partedSeqKeys ]
},

    Do[
        createSeq[seqName , splitName ], (* Uses the split key that called it for the startKey
        {seqName , unpartedSeqKeys } (* Runs for each seqName *)
    ]
]
```

In[ ]:=
```
(* branchKeys- list of a split's derivative sequences (size n)
  mergePairs-
 list of truth values that tell if a pair of sequences merge (size n-1)

  Uses both pieces of information to
  output partitioned lists that hold the sequences.
   These partitioned lists are used in merge creation *)
```

```
createMergeStartKeys[branchKeys_, mergePairs_] := Module[
  {partitionedMerges = {},
   (* The set of partitioned sequences for multiple merges from a split *)
   currentPartition, (* The set of merging sequences we are building *)
   unpartedKeys = Flatten[branchKeys],
   firstSeq,
   secondSeq,
   alreadyAdded (* if it is already in the sequence *)
  },

  currentPartition = {};
  Do[
   firstSeq = unpartedKeys[[mergeCount]];
   secondSeq = unpartedKeys[[mergeCount + 1]];
   alreadyAdded = MemberQ[currentPartition, firstSeq];


   If[mergePairs[[mergeCount]],
    (*true*)
        If[! alreadyAdded,
         (*true*)
         AppendTo[currentPartition, firstSeq]
         ];
    AppendTo[currentPartition, secondSeq],

    (*false*)
    AppendTo[partitionedMerges, currentPartition];
    currentPartition = {};
   ]
   , {mergeCount, Length[mergePairs]}
  ];

  (* Building has now stopped because all the merge truth values
   have been iterated through. So append the last built partition *)
  AppendTo[partitionedMerges, currentPartition];
  partitionedMerges = Select[partitionedMerges, ! SameQ[#, {}] &];
  (* Filter out empty sets*)
  Return[partitionedMerges]
 ]
```

In[ ]:=
```
(* Is called by create split. Takes
 each branch key and creates sequences for it*)

appendMergeSequences [mergeStartKeys_ ] :=
 Do[
  createMer[startKeyList ]
  , {startKeyList , mergeStartKeys }
 ]
```

In[ ]:=
```
setMergedSeqsEnds [startKeys_ , merKeyP_] := Module[
  {merKey = merKeyP},
  Do[
    setEnd[startKey , merKey];
    , {startKey , startKeys}
   ];
 ]
```

In[ ]:=
```
(* Helper method for paths. Allows us to come back and change/add
 to the attributes later and not have to change a lot of code *)
instantiatePathAttributes [] :=
 "attr" → <|"flowRate" → Null,
   "avgVelocity" → Null, "avgDiameter" → Null, "height" → Null|>
```

Path Modification

In[ ]:=
```
(* set a path's end to a desired endKey *)
setEnd[givenPath_ , desired_] := Module[
  {path = givenPath},
  masterPathList[path, "end"] = desired ;
 ]
```

In[ ]:=
```
(* set a path's end to a desired startKey *)
setStart[givenPath_ , desired_] := Module[
  {path = givenPath},
  masterPathList[path, "start"] = desired ;
 ]
```

In[ ]:=
```
(* set a seq's height *)
setHeight[seqKey_, height_] := Module[
  {seq = seqKey},
  masterPathList[seq, "attr", "height"] = height;
 ]
```

In[ ]:=
```
(* Remove a path from the model *)
removePath[pathKey_] := Module[
  {path = pathKey},
  masterPathList  = Delete[masterPathList , path];
 ]
```

Processing - Main

In[ ]:=
```
(* The main function that calls the processing methods *)
(* Handles the distribution of flowRate , temperature ,
etc and processing of attributes like avgDiameter and avgVelocity *)
processSysMain [] := Module[
  {},
      processSysDiam [];
      processSysFR [];
      processSysVels [];
 ]
```

In[ ]:=
```
(* Goes through the linked branches to process
 the flowRate across members using established rules *)
(* System must be built before this is executed *)

(* seq/mer→seq -flowIn=flowOut , so it'll be the same
  seq→split -proportions are used
   mer -sum of incoming seq's flowRates
*)
processSysFR [] := Module[
  {originPath = Keys[masterPathList ][[1]],
   hasPump ,
   pathType ,
   flowRate ,
   prevKey ,
   prevKeyType
  },
```

```
hasPump = containsPump[originPath];
(* Flow rate based on origin seq is set *)
(* The origin seq's pipe, pump, and reservoir affect this value *)
If[hasPump,
      (*true*)
      setOriginFRPump[originPath],
      (*false*)
      setOriginFRGrav[originPath]
];

(* Sets flow rates for the sequences on the masterPathList *)
Do[
 (* pathType is "mer" or "seq" or "spl" *)
 pathType = getPathType[key];
 prevKey = masterPathList[key, "start"];

 (* Skips the first seq so an
  error isn't thrown when the prev info is retrieved *)
 If[! StringQ[prevKey],
       Continue[];
 ];

 prevKeyType = getPathType[prevKey];


 (*-------------SPL CHECK------------------- *)
 (* If it's a spl, process it and move to the next key *)
 If[pathType == "spl",
       setFRspl[key];
       Continue[];
 ];

 flowRate = masterPathList[key, "attr", "flowRate"];
 (*---------------SEQ CHECKS------------------- *)
 (* check for gate/valve closings *)
 If[masterPathList[key, "attr", "avgDiameter"] == 0,
       masterPathList[key, "attr", "flowRate"] = 0;
       Continue[];
 ];
 (* If the path is a seq that starts from a
   merge execute the flowRate process for a merge lead-in *)
 If[pathType == "seq" && prevKeyType == "mer",
       setFRmerLeadIn[key];
```

```
    ];
    (* If the path is a seq that starts from a seq,
    execute the flowRate process for a seq lead-in *)
    If[pathType == "seq" && prevKeyType == "seq",
          setFRseqLeadIn[key];
    ];

    , {key, Keys[masterPathList]}
  ];

 ]
```

In[ ]:=

```
(* Prerequisite: flow rates have to be processed and parts have to be added *)
(* Updates the avgDiameter for the sequences in the network *)
processSysDiam[] := Module[
  {pathType},

  (* runs through the masterPathList
   and processes the diameter if it's a seq *)
  Do[
    pathType = getPathType[key];

    If[SameQ[pathType, "seq"],
     (* if it's a seq *)
     updateAvgDiameter[key];
    ];

    , {key, Keys[masterPathList]}
   ];
 ]
```

In[ ]:=
```
(* Prerequisite : flow rates have to be processed and
   parts have to be added. Avg Diameter also has to be updated *)
(* Updates the velocity values for the sequences in the network *)
processSysVels [] := Module[
  {pathType},

  (* runs through the masterPathList
   and processes the velocity if it's a seq *)
  Do[
    pathType = getPathType[key];

    If[SameQ[pathType, "seq"],
     (* if it's a seq *)
     updateAvgVelocity [key];
    ];

    , {key, Keys[masterPathList ]}
   ];
 ]
```

In[ ]:=
```
(* returns true if all the sequences have atleast a part in them *)
(* checks the seq end references to make sure there
 are no Null maps. Each must lead in or have an outputRes *)
allSeqsComplete[] := Module[
  {seqKeys = Select[Keys[masterPathList], StringContainsQ[#, "seq"] &],
   (* get the seq names *)
   hasOutputOrConnection ,
   hasPart ,
   fullyComplete = True (* the boolean return value *)
  },

  (* run through the sequences. Make sure they have an end mapping and parts *)
  Do[
       hasOutputOrConnection = !SameQ[Null, masterPathList[seqKey, "end"]];
       hasPart = !SameQ[{}, masterPathList[seqKey, "parts"]];
       If[!(hasOutputOrConnection && hasPart),
            fullyComplete = False;
        ];
   ,{seqKey, seqKeys}
  ];

  (* Note- will return true if masterPathList is empty. This
    doesn't matter because seq1 will be present when this is ran *)
  Return[fullyComplete]
 ]
```

Processing - Flow Rate Functions

In[ ]:=
```
(* Checks to see if the defined originPath contains a pump *)
containsPump[originPath_] := Module[
  {startPath = originPath ,
   reservoirList
  },

  (* checks the size of the startSeq's start
   reference. It will have two elements if a pump is present *)
  reservoirList = masterPathList[startPath, "start"];
  Return[Length@reservoirList > 1];
 ]
```

In[ ]:=
```
(* Retrieves the pump's flowRate and sets the seq's flowRate attribute *)
setOriginFRPump[originPath_] := Module[
  {path = originPath,
   flowRate},

  (* gets the flowRate from the pump in the beginning path *)
  (* the pump is the second element in the originSeq if present. Then
   the flowRate for the pump is the pump's second element *)
  flowRate = masterPathList[path, "start"][[2]][[2]];

  (* set the flowRate *)
  masterPathList[path, "attr", "flowRate"] = flowRate;
 ]
```

In[ ]:=
```
(* Retrieves the flowRate based on gravity and starting
 pipe diameter and sets the seq's flowRate attribute*)
setOriginFRGrav[originPath_] := Module[
  {path = originPath,
   startingPipeD,
   height,
   waterVelocity,
   flowRate},

  (* retrieves the pipe diameter and height from the pipe in the origin seq *)
  startingPipeD = masterPathList[path, "parts"][[1, 2]];
  (* retrieves the height of the input reservoir,
  located at index 1 of the start link and index 2 in the inputR list *)
  height = masterPathList[path, "start"][[1, 2]];

  (* calculation for velocity of water exiting the reservoir through a pipe *)
  waterVelocity = Sqrt[2 * globalConstants["gravity"] * height];
  flowRate = waterVelocity * calcCrossArea[startingPipeD];
  (* set the flowRate *)
  masterPathList[path, "attr", "flowRate"] = flowRate;
 ]
```

In[ ]:=
```
(* set the flowRate for a given seq based on the merge that came before *)
setFRmerLeadIn[seqKey_] := Module[
  {path = seqKey,
   mergeKey,
   prevKeys,
   prevFR,
   flowRates},

  (* retrieving - get the merge it came from and then get
     that merges' startKeys (the branches that lead in to it) *)
  mergeKey = masterPathList[path, "start"];
  prevKeys = masterPathList[mergeKey, "start"];

  (* calculating and setting *)
  flowRates = Table[masterPathList[key, "attr", "flowRate"], {key, prevKeys}];
  masterPathList[path, "attr", "flowRate"] = Total@flowRates;

 ]
```

In[ ]:=
```
(* set the flowRate for a given seq based on the seq that came before *)
setFRseqLeadIn[seqKey_] := Module[
  {path = seqKey,
   prevKey,
   prevFR
  },

  (* retrieving *)
  prevKey = masterPathList[path, "start"];
  prevFR = masterPathList[prevKey, "attr", "flowRate"];

  (* setting *)
  masterPathList[path, "attr", "flowRate"] = prevFR;
 ]
```

In[ ]:=
```
(* set the flowRate for a split's
 sequences based on the flowRate of the prev seq *)
setFRspl[splKey_] := Module[
  {path = splKey,
   prevSeq,
   prevFR,
   avgDiameters,
   branchKeys,
   partialFlows,
   seq,
   currentFlow,
   currentBranch},

  (* retrieving *)
  prevSeq = masterPathList[path, "start"];
  prevFR = masterPathList[prevSeq, "attr", "flowRate"];
  branchKeys = Flatten@masterPathList[path, "branches"];
  avgDiameters =
   Table[masterPathList[branch, "attr", "avgDiameter"], {branch, branchKeys}];

  (* calculating and setting *)
  partialFlows = calcPartialFlowRates[prevFR, avgDiameters];

  (* set the seq's new flow rate *)
  For[seq = 1, seq ≤ Length@partialFlows, seq++,
   currentBranch = branchKeys[[seq]];
   currentFlow = partialFlows[[seq]];
   masterPathList[currentBranch, "attr", "flowRate"] = currentFlow;
  ]
 ]
```

In[ ]:=

```
(* checks a sequence's parts for closed gates/valves. The
 function will return true if there is a closing *)
closedGatesValves[seqKey_] := Module[
  {path = seqKey,
   partsList,
   closing = False
  },

  partsList = masterPathList[path, "parts"];

  (* loop through each part *)
  Do[
   (* check the part's diameter value *)
   If[QuantityMagnitude[part[[2]]] == 0,
     closing = True;
    ];
   ,{part, partsList}
  ];

  Return[closing]
 ]
```

Processing - Updates for avgDia/velocity and return function for diff pressure

In[ ]:=
```
(* Looks at a seq's partsList to determine the average diameter *)
updateAvgDiameter[seqKey_] := Module[
  {path = seqKey,
   partsList,
   avgDiameter,
   diameters,
   diameter
  },

  (* check for gate/valve closings *)
  If[closedGatesValves[path],
       masterPathList[path, "attr", "avgDiameter"] = 0;
       Return[];
  ];

  (* retrieve the parts *)
  partsList = masterPathList[path, "parts"];

  (* get diameters for the parts *)
  diameters = Table[
    diameter = part[[2]]
    , {part, partsList}
   ];

  (* update *)
  masterPathList[path, "attr", "avgDiameter"] = Mean[diameters];
 ]
```

In[ ]:=
```
(* Uses the seq's flowRate and the avg Diameter
 to update the avg velocity through the sequence *)
updateAvgVelocity[seqKey_] := Module[
  {path = seqKey,
   flowRate,
   avgDiameter,
   crossArea,
   velocity
  },

  (* retrieving *)
  flowRate = masterPathList[path, "attr", "flowRate"];
  avgDiameter = masterPathList[path, "attr", "avgDiameter"];
  crossArea = calcCrossArea[avgDiameter];

  (* Handles the division by 0 error *)
  If[avgDiameter === 0 || flowRate === 0,
       (* true *)
       velocity = 0,
       (* false *)
       velocity = flowRate / crossArea;
  ];

  (* update *)
  masterPathList[path, "attr", "avgVelocity"] = velocity;

 ]
```

In[ ]:=
```
(* finds the differential pressure between two sequences in the system *)
findDiffPress[densityLiq_, seq1_, seq2_] := Module[
  {path1 = seq1,
   path2 = seq2,
   vel1,
   vel2,
   h1,
   h2,
   diffPress
  },

  (* retrieve velocities and heights*)
  vel1 = masterPathList[path1, "attr", "avgVelocity"];
  vel2 = masterPathList[path2, "attr", "avgVelocity"];
  h1 = masterPathList[path1, "attr", "height"];
  h2 = masterPathList[path2, "attr", "height"];

  (* return the differential pressure *)
  diffPress = calcDiffPress[densityLiq, vel1, vel2, h1, h2];
  Return[diffPress]
 ]
```

Processing- Water Transfer using inf reservoirs and inf pump

In[ ]:=
```
(* fills the outputs in the system using fillUnOutput as a helper *)
fillUnOutputs[seconds_] := Module[
  {endVal
  },

  (* if it's a sequence with an unbounded output reservoir,
  fill it for the given time*)
  Do[
    endVal = masterPathList[key, "end"];
    If[ListQ[endVal],
          fillUnOutput[key, seconds];
    ]
    , {key, Keys[masterPathList]}
   ];
 ]
```

In[ ]:=
```
(* fills an output on a given sequence
 based on the amount of water coming in and time *)
fillUnOutput[seqKey_, seconds_] := Module[
  {path = seqKey,
   flowOut,
   outputR, (* output reservoir *)
   currentVol,
   addedAmount
  },

  (* retrieving the flow into the
   output and information for the output reservoir *)
  flowOut = masterPathList[path, "attr", "flowRate"];
  outputR = masterPathList[path, "end"];

  currentVol = outputR[[4]];
  addedAmount = seconds * flowOut;

  (* Dealing with case where flowOut is 0 and unit conflict happens *)
  If[flowOut == 0,
   addedAmount = 0;
  ];

  masterPathList[path, "end"] =
   ReplacePart[masterPathList[path, "end"], 4 → (currentVol + addedAmount)];
 ]
```

Processing- Gen Pumps

In[ ]:=
```
(* Go through the structure to determine the required
 head value to be able to pump water throughout the network *)
(* uses the findMaxHeight and findMaxRHeadVal helper functions
 to determine what the required head value is for a pump *)
findHeadVal[] :=
 Return[Max[{findMaxHeight[], findMaxRHeadVal[]}]]
```

In[ ]:=
```
(* This uses the height of each sequence to determine head value. Another
 approach must be used in addition to factor in output reservoirs *)
findMaxHeight[] := Module[
  {originHeight = masterPathList["seq1", "attr", "height"],
   heights
  },

  (* build list of all differences
   in height from the inputR to each sequence *)
  heights = Table[
    masterPathList[key, "attr", "height"] - originHeight
    , {key, Keys[masterPathList]}
   ];

  (* filter the results from splits and merges -they don't have a height *)
  heights = Select[heights, QuantityQ[#] &];

  (* return max height difference *)
  Return[Max[heights]]
 ]
```

In[ ]:=
```
(* Uses pressure exerted on reservoirs in addition
 to height to determine the head value a pump would need *)
findMaxRHeadVal[] := Module[
  {endVal,
   originR,
   densityUpperFluO, ceilingDisO,
   gaugePressureO, heightO, (* calculation variables *)
   densityLiq,
   pressureOriginR,
   headVals = {},
   pressureB, densityUpperFluB, ceilingDisB, guagePressureB,
   heightB, gaugePressureB (* calculation variables *),
  },


  (* calc pressure on origin reservoir free surface *)
  originR = masterPathList["seq1", "start"][[1]];
  densityUpperFluO = originR[[8]];
  ceilingDisO = originR[[3]];
  gaugePressureO = originR[[4]];
  pressureOriginR =
```

```
  calcPressureReservoir [densityUpperFluO , ceilingDisO , gaugePressureO ];

 (* retrieve other values needed for calculating the
  head value between the input and each output reservoir *)
 heightO = masterPathList ["seq1", "attr", "height"];
 densityLiq = masterPathList ["seq1", "start"][[1, 7]];


 (* build list of head values *)
 headVals = Table[
   endVal = masterPathList [key, "end"];
   (* if it's a seq with an output reservoir *)
   If[ListQ[endVal],
        (* retrieve values for calculating pressure on reservoirB *)
        densityUpperFluB = endVal[[8]];
        ceilingDisB = endVal[[3]];
        gaugePressureB = endVal[[4]];

        (* calculate pressure on B, retrieve its height, and then use the
     data from the origin reservoir along with the data for the given
     output reservoir to calculate the required head value for a pump *)
        pressureB = calcPressureReservoir [densityUpperFluB ,
      ceilingDisB , gaugePressureB ];
        heightB = masterPathList [key, "attr", "height"];

        calcPumpHead [pressureOriginR , heightO , pressureB , heightB , densityLiq ]
   ]

   , {key, Keys[masterPathList ]}
  ];

 (* filter the null results out *)
 headVals = Select[headVals , QuantityQ [#] &];

 Return [Max[headVals ]];
]
```

Processing- Generalized water transfer

In[ ]:=
```
(* returns true if water can be distributed, false if it can't *)
(* water can be distributed if the pump's head value
 (capacity to do work) is appropriate for the constructed network *)
(* if no pump is present, head value requirement must be 0 or less*)
headValCheck[] := Module[
  {neededHV,
   pumpHV,
   hasPump = containsPump["seq1"],
   passed = False
  },

  neededHV = findHeadVal[]; (* finds required headVal *)

  If[hasPump,
       pumpHV = masterPathList["seq1", "start"][[2, 4]];
   (* retrieve pump's actual head value *)
       passed = pumpHV ≥ neededHV,
       (* no pump present *)
       passed = neededHV ≤ Quantity[0, "Meters"];
  ];

  (* compares head values. The actual must be equal to or bigger than
   what we need. If a pump isn't present, the required must be ≤ 0 *)
  Return[passed];
 ]
```

In[ ]:=
```
(* fills an output on a given sequence
 based on the amount of water coming in and time *)
(* also factors in the pump's head value to
 determine if the water can be distributed *)
fillGenOutput[seqKey_, seconds_] := Module[
  {path = seqKey,
   pumpData,
   outputR, (* output reservoir *)
   flowOut,
   currentVol,
   maxVol,
   addedAmount,
   newVol
  },

  (* retrieving the flow into the
```

```
   output and information for the output reservoir *)
  flowOut = masterPathList[path, "attr", "flowRate"];
  outputR = masterPathList[path, "end"];

  currentVol = outputR[[5]];
  maxVol = outputR[[7]];
  addedAmount = seconds * flowOut;

  (* can't fill up beyond the max volume *)
  If[currentVol + addedAmount > maxVol,
       (* true *)
       newVol = maxVol,
       (* false *)
       newVol = currentVol + addedAmount
  ];

  (* Dealing with case where flowOut is 0 and unit conflict happens *)
  If[flowOut == 0,
   addedAmount = 0;
  ];


  (* assuming pump is powerful enough, move water *)
  masterPathList[path, "end"] =
   ReplacePart[masterPathList[path, "end"], 5 → (newVol)]

]
```

In[ ]:=
```
(* fills the gen outputs in the system using fillGenOutput as a helper *)
fillGenOutputs[seconds_] := Module[
  {endVal
  },

  (* if it's a sequence with a generalized output reservoir,
  fill it for the given time *)
  Do[
    endVal = masterPathList[key, "end"];
    If[ListQ[endVal],
          fillGenOutput[key, seconds];
    ]
    ,{key, Keys[masterPathList]}
   ];
 ]
```

Setup and Runtime

In[ ]:=
```
(* Initializes the global constants for
 calculations and the masterPathList for our model *)
initializeLists[] :=
 resetGlobalConstants[];     resetMasterPathList[];
```

Controller/Graphics Creation

Seq Positions + Points

In[ ]:=
```
(* takes a {pathName, xStart, yStart, xEnd, yEnd}
 sublist and only returns the x,y of each *)
getStartAndEndPts[pathName_, seqLocs_] := Module[
  {loc,
   pathAndPos,
   startPt,
   endPt
  },

  (* find where the path info is in the list and pull its info *)
  loc = getSeqPos[seqLocs, pathName];
  pathAndPos = seqLocs[[loc]];

  (* retrieve start and ending locations *)
  startPt = {pathAndPos[[2]], pathAndPos[[3]]};
  endPt = {pathAndPos[[4]], pathAndPos[[5]]};

  Return[{startPt, endPt}]
 ]
```

In[ ]:=
```
(* Returns a list of sequences with {start, end} points where each path'
 s position and length is based off of height and layer length *)
(* unit should be in string form. Typically units will be "Meters" *)
(* height scale is a multiplier on each coords yPos *)
seqNameAndPos[layerLength_, heightScale_, units_] := Module[
  {layerList = layerList[], (* determines every path's layer (dis from origin) *)
   posList, (* positions *)
   key,
   layer,
   height,
   xPos,
   yPos,
   endX
  },

  (* filter out non-sequences *)
  layerList = Select[layerList, StringContainsQ[#[[1]], "seq"] &];

  posList = Table[
    key = keyLayer[[1]];
    layer = keyLayer[[2]];

    (* determine yPos using the appropriate seq's height *)
    height = UnitConvert[masterPathList[key, "attr", "height"], units];
    yPos = QuantityMagnitude[height] * heightScale;

    (* determine xPos. each layer is of equal length *)
    xPos = layer * layerLength;

    (* determine where the path ends *)
    endX = xPos + layerLength;

    (* output *)
    (* yPos won't change between the path start and end *)
    {key, xPos, yPos, endX, yPos}

    , {keyLayer, layerList}(* run for each key-layer pair *)
   ];

  Return[posList]
 ]
```

In[ ]:=
```
(* looks through the list of {seqName, xStart, yStart, xEnd, yEnd}
 subsets and returns the loc of the one you're searching for *)
getSeqPos[seqLocs_, seqName_] :=
     Position[seqLocs, seqName][[1, 1]]
```

In[ ]:=
```
(* Returns the a list of each sequence
 and the layer of the build that it's in *)
(* Layer is determined by the distance to the starting sequence *)
layerList[] :=
 Table[
  {key, distanceToOrigin[key]}
  , {key, Keys[masterPathList]}
 ]
```

In[ ]:=
```
(* determine a sequences distance to the origin *)
distanceToOrigin[seq_] := Module[
  {currentPath = seq,
   dis = 1
  },

  (* keep looping until arrival at seq1 *)
  While[! SameQ[currentPath, "seq1"],
   (* go to prev sequence *)
   currentPath = masterPathList[currentPath, "start"];

   (* if it's a merge,
   just choose the first sequence because they all come from the same split *)
   (* merges have lists of sequences for the start value *)
   If[ListQ[currentPath],
        currentPath = currentPath[[1]];
   ];

   (* if the previous path is a sequence *)
   (* splits and merges are in the same layer,
   we don't want to increase distance for those *)
   If[StringContainsQ[currentPath, "seq"],
        dis++
   ];
  ];

  Return[dis]
 ]
```

In[ ]:=

Spl/Mer Locations

In[ ]:=
```
(* uses the sequence that leads-
 in to a split to determine the split's coordinates .*)
getSplNameLoc [seqLocs_ , splName_ ] := Module[
  {splKey = splName ,
   prevSeq
  },

  prevSeq = masterPathList [splKey , "start"];

  (* return the name with the prev seq's end coordinate *)
  Return [{splKey , getStartAndEndPts [prevSeq , seqLocs ][[2]]}]

 ]
```

In[ ]:=
```
(* uses the sequence that leads-
 in to a split to determine the split's coordinates . outputs without name! *)
findSplLoc [seqLocs_ , splName_ ] :=
     getSplNameLoc [seqLocs , splName ][[2]]
```

In[ ]:=
```
(* uses the sequence that the merge leads-
 in to in order to determine the merge's coordinates . *)
getMerNameLoc [seqLocs_ , merName_ ] := Module[
  {merKey = merName ,
   nextSeq
  },

  nextSeq = masterPathList [merKey , "end"];

  (* return the name with the next seq's start coordinate *)
  Return [{merKey , getStartAndEndPts [nextSeq , seqLocs ][[1]]}]

 ]
```

In[ ]:=
```
(* uses the sequence that the merge leads-
 in to in order to determine the merge's coordinates . outputs without name! *)
findMerLoc [seqLocs_ , merName_ ] :=
     getMerNameLoc [seqLocs , merName ][[2]]
```

In[ ]:=
```
(* returns list of where all the splits are located *)
allSplNameLocs[seqLocs_] := Module[
  {splKeys
  },

  (* get the split keys *)
  splKeys = Select[Keys[masterPathList], StringContainsQ[#, "spl"] &];

  (* find their locations *)
  Table[
       getSplNameLoc[seqLocs, splKey]
   , {splKey, splKeys}
  ]

 ]
```

In[ ]:=
```
(* returns list of where all the merges are located *)
allMerNameLocs[seqLocs_] := Module[
  {merKeys
  },

  (* get the merge keys *)
  merKeys = Select[Keys[masterPathList], StringContainsQ[#, "mer"] &];

  (* find their locations *)
  Table[
       getMerNameLoc[seqLocs, merKey]
   , {merKey, merKeys}
  ]
 ]
```

Spl/Mer Rectangles + Labels

In[ ]:=

```
(* returns colored rectangles at the
 correct positions to designate each split *)
getSplRects[seqLocs_, layerLength_, rectColor_] := Module[
  {splNameLocs = allSplNameLocs[seqLocs],
   width = layerLength / 3,
   height = layerLength / 7,
   xVal,
   yVal
  },

  Table[
       xVal = splSet[[2, 1]];
       yVal = splSet[[2, 2]];
       (* create rectangle *)
       {rectColor,
    Rectangle[{xVal - width / 2, yVal - height / 2}, {xVal + width / 2, yVal + height / 2}]}
   , {splSet, splNameLocs}(* run for each set of
    coordiantes that designates a split *)
  ]

 ]
```

In[ ]:=
```
(* returns colored rectangles at the
 correct positions to designate each merge *)
getMerRects[seqLocs_, layerLength_, rectColor_] := Module[
  {merNameLocs = allMerNameLocs[seqLocs],
   width = layerLength / 3,
   height = layerLength / 7,
   xVal,
   yVal
  },

  Table[
       xVal = merSet[[2, 1]];
       yVal = merSet[[2, 2]];
       (* create rectangle *)
       {rectColor,
    Rectangle[{xVal - width / 2, yVal - height / 2}, {xVal + width / 2, yVal + height / 2}]}
   , {merSet, merNameLocs}(* run for each set of
    coordiantes that designates a merge *)
  ]

 ]
```

In[ ]:=
```
(* puts customizable labels at the correct positions for splits *)
getSplLabels[seqLocs_, layerLength_,
  labelColor_, fontType_, fontSize_] := Module[
  {splNameLocs = allSplNameLocs[seqLocs],
   xVal,
   yVal
  },

  Table[
       xVal = splSet[[2, 1]];
       yVal = splSet[[2, 2]];
       (* create label *)
       {labelColor, Style[Text[splSet[[1]], {xVal, yVal}],
     FontSize → fontSize, FontFamily → fontType]}
   , {splSet, splNameLocs}(* run for each set of
    coordiantes that designates a spl *)
  ]
 ]
```

In[ ]:=
```
(* puts customizable labels at the correct positions for merges *)
getMerLabels[seqLocs_, layerLength_,
  labelColor_, fontType_, fontSize_] := Module[
  {merNameLocs = allMerNameLocs[seqLocs],
   xVal,
   yVal
  },

  Table[
       xVal = merSet[[2, 1]];
       yVal = merSet[[2, 2]];
       (* create label *)
       {labelColor, Style[Text[merSet[[1]], {xVal, yVal}],
     FontSize → fontSize, FontFamily → fontType]}
   , {merSet, merNameLocs}(* run for each set of
    coordiantes that designates a merge *)
  ]
 ]
```

Seq Buttons

In[ ]:=
```
(* outputs a list of text graphics
 at a position centered above each sequence *)
(* When buttons are pressed, the sequence will be selected in the UI *)
getSeqButtons[seqLocs_, fontSize_, fontType_, layerLength_, height_] := Module[
  {seqStartPos,
  },

  Table[
   (* With command- used for substituting values. We needed
      it to solve a bug where our button actions weren't working *)
   Inset[With[{seqName = seq[[1]]},
     Button[Style[seqName, FontSize → fontSize, FontFamily → fontType],
      selected = seqName;
      (* will set global variable for "selected" to the selected sequence *)
      If[!isDone, (* creates the build options if they
         user hasn't locked the building *)
       createBuildOptions[];
      ]
     ]],
        (* Position *)
    {seq[[2]] + layerLength / 2, seq[[3]] + height}
   ]
   , {seq, seqLocs}(* run for every sequence *)
  ]

 ]
```

In[ ]:=

Point Output for Seq/Spl/Mer

In[ ]:=

```
(* determines the set of 3 points for each seq that leads in to another seq *)
(* if a seq doesn't lead to another seq,
a set of only 2 points is returned (the seq's start and end) *)
(* these are needed for making lines to connect sequences *)
(* these subsets of points will later
 be used with Line[] to connect sequences *)
allSeqPoints[seqLocs_] := Module[
  {currentKey,
   next,
   connectionPoints
  },

  connectionPoints = Table[
    currentKey = pathLoc[[1]];
    next = masterPathList[currentKey, "end"];

    (* if it's a seq, process it's coordinate data *)
    If[StringContainsQ[currentKey, "seq"],

          If[StringQ[next] && StringContainsQ[next, "seq"],
               (* true. there is a next path *)
               seqConnectPts[seqLocs, currentKey, next],
               (* false, there is no other path to lead in to *)
               getStartAndEndPts[currentKey, seqLocs]
           ]
    ]

    , {pathLoc, seqLocs}
    (* run for each {pathName, xStart, yStart, xEnd, yEnd} sublist *)
   ];

  (* filter out null results *)
  connectionPoints = Select[connectionPoints, !SameQ[#, Null] &];
  (* return list of connections *)
  Return[connectionPoints]
 ]
```

In[ ]:=
```
(* uses the x of the second path and the y of
 the first path to figure out the "intersection" point *)
(* The intersection is where the first path connects to the second path OR
 where the first path goes up/down to connect to the second path *)
(* The intersection is returned with the first and last points! *)
seqConnectPts[seqLocs_, path1_, path2_] := Module[
  {seq1Loc,
   seq2Loc,
   seq1Data,
   seq2Data,
   seq1Start,
   seq1End,
   seq2Start
  },

  (* retrieve seq1 location data *)
  seq1Loc = getSeqPos[seqLocs, path1];
  seq1Data = seqLocs[[seq1Loc]];
  seq1Start = {seq1Data[[2]], seq1Data[[3]]};
  seq1End = {seq1Data[[4]], seq1Data[[5]]};

  (* retrieve seq2 location data *)
  seq2Loc = getSeqPos[seqLocs, path2];
  seq2Data = seqLocs[[seq2Loc]];
  seq2Start = {seq2Data[[2]], seq2Data[[3]]};


  (* the intersection point is the middle one. The first 2 elements
   are the bottom left corner. last 2 are the top right corner*)
  (* the first point is the first seq's loc. The
   third point is the last seq's loc *)
  Return[{seq1Start, seq1End, seq2Start}]

 ]
```

In[ ]:=
```
(* uses splPoints to find all the point sets for all the
 vertical lines that connect sequences in the various splits *)
splPtsAll[seqLocs_] := Module[
  {splKeys,
   allSplPoints,
  },

  splKeys = Select[Keys[masterPathList], StringContainsQ[#, "spl"] &];

  (* Process all splits *)
  allSplPoints = Table[
        splPoints[seqLocs, splKey]
    , {splKey, splKeys}
   ]

 ]
```

In[ ]:=
```
(* uses merPoints to find all the point sets for all the
 vertical lines that connect sequences in the various merges *)
merPtsAll[seqLocs_] := Module[
  {merKeys,
   allMerPoints,
  },

  merKeys = Select[Keys[masterPathList], StringContainsQ[#, "mer"] &];

  (* Process all splits *)
  allMerPoints = Table[
        merPoints[seqLocs, merKey]
    , {merKey, merKeys}
   ]

 ]
```

In[ ]:=
```
(* Returns a list of points from top
 to bottom for all the branches of a split *)
(* used later with Line[] to make a vertical
 line connecting the split's sequences *)
splPoints[seqLocs_, spl_] := Module[
  {pts,
   splKey = spl
  },

  pts = Table[
    getStartAndEndPts[branchKey, seqLocs][[1]](* pick only the starting point *)
    , {branchKey, Flatten[masterPathList[splKey, "branches"]]}
    (* run for each seq on the split *)
   ];

  (* add the merge start point. This allows the vertical
   line to extend above and below the merging sequences *)
  pts = Join[pts, {findSplLoc[seqLocs, splKey]}];

  Return[pts]
 ]
```

In[ ]:=
```
(* similar to splPoints. Returns a list of
 points from top to bottom for all converging splits *)
(* used with Line[] to connect them visually *)
merPoints[seqLocs_, mer_] := Module[
  {pts,
   merKey = mer
  },

  pts = Table[
    getStartAndEndPts[branchKey, seqLocs][[2]] (* pick only the end point *)
    , {branchKey, masterPathList[merKey, "start"]}
    (* run for each seq on the split *)
   ];

  (* add the merge start point. This allows the vertical
   line to extend above and below the merging sequences *)
  pts = Join[pts, {findMerLoc[seqLocs, merKey]}];

  Return[pts]
 ]
```

Seq1 Graphics (InputR and Pump)

In[ ]:=

```
(* Outputs rectangle to represent the input reservoir  *)
(* The rectangle will be attached to the pipe using its bottom right corner
 so it doesn't clip into the structure *)(* leftShift will move it *)
(* downShift moves the rectangle downwards so it can correctly
 rest on the line that represents the seq1 pipes *)
getInputRect[seqLocs_, color_, layerLength_, downShift_] := Module[
  {startPoint = getStartAndEndPts["seq1", seqLocs][[1]],
   leftShift = layerLength / 2,
   rect, (* stores graphics *)
   bottomLeft,
   topRight
  },

  (* seq1 starts at startPoint. Now we're shifting
   it to the left by leftShift for creating the rectangle *)
  bottomLeft = startPoint - {leftShift, 0};

  (* find topright corner for rectangle *)
  topRight = {bottomLeft[[1]] + layerLength / 2, bottomLeft[[2]] + layerLength / 2};

  (* shift the corners down by downShift *)
  bottomLeft = bottomLeft - {0, downShift};
  topRight = topRight - {0, downShift};

  rect = {color, Rectangle[bottomLeft,
     {bottomLeft[[1]] + layerLength / 2, bottomLeft[[2]] + layerLength / 2}]};

  Return[rect]
 ]
```

In[ ]:=
```
(* outputs circle to represent the pump *)
(* uses the start point of seq1 to position itself *)
(* the circle will be shifted by rightShift to put it into proper position *)
getPumpCirc[seqLocs_, color_, layerLength_, rad_] := Module[
  {startPoint = getStartAndEndPts["seq1", seqLocs][[1]],
   rightShift = layerLength / 4,
   circ, (* stores graphics *)
   circLoc
  },

  (* find circLoc by shifting seq1's start point *)
  circLoc = startPoint + {rightShift, 0};

  circ = {color, Disk[circLoc, rad]}

 ]
```

Output Reservoirs Graphic

In[ ]:=
```
(* represents an output reservoir with a colored rectangle *)
createOutputCirc[seqLocs_, seqKey_, color_, rad_] := Module[
  {endPt = getStartAndEndPts[seqKey, seqLocs][[2]],
   circ, (* stores graphics *)
   circLoc
  },

  {color, Disk[endPt, rad]}
 ]
```

In[ ]:=
```
(* creates output disks for the sequences that end in outputs *)
createOutputCircs[seqLocs_, color_, rad_] := Module[
  {outputSeqs,
   },

  (* get the sequences that aren't linked to a spl/seq/mer *)
  outputSeqs = Select[seqLocs[[All, 1]], ListQ[masterPathList[#, "end"]] &];

  Table[
       createOutputCirc[seqLocs, outputSeq, color, rad]
   , {outputSeq, outputSeqs}
  ]

 ]
```

Interface

Global Variables

```
(* initializes the UI global variables *)
initializeUIglobals []:=
(* the currently selected path/component index *)
selected = "seq1";
(* used for storing current screens and closing them *)
currentDisplayScreen = Null;
currentOptionsScreen = Null;
currentPropertiesScreen = Null;
currentRunOptionsScreen = Null;
currentAllStarScreen = Null;
selectedScreen = Null;
(* stores user input *)
inputDynamic = Null;
(* Boolean that notes if the setup is an infinite or general one *)
isInfinite = Null;
(* Boolean that notes if the user is done *)
isDone = False;
(* Finds value of window sizes to automatically scale *)
windowSizeX = AbsoluteCurrentValue [WindowSize][[1]];
windowSizeY = AbsoluteCurrentValue [WindowSize][[2]];

(* Used for saving the current setup so you can reload it *)
masterPathListTemp = Null;

(* sets directory correctly for pictures/other files *)
SetDirectory [NotebookDirectory []];
```

In[ ]:=
```
initializeUIglobals [];
```

Starting Window

In[ ]:=

```
(* creates and displays the starting menu in a new window *)
startingMenu []:=
currentDisplayScreen =
CreatePalette [
    (* Graphical elements *)
    (* Positions and sizes are proportions of the user's screen dimensions *)
    {Graphics [{
              {
             Inset[ImageResize [Import["Pictures /background2 .png"], {windowSizeX , windowSizeY }]
              Inset[Style["Fluid Mechanics for English Majors", White, FontSize → windowSize
              Inset[Button[Style["Start!", FontFamily → "Comic Sans MS"], resetWorkingScreens
             Inset[Button[Style["Quit :(", FontFamily → "Comic Sans MS"], NotebookClose [], Ima
             Inset[Button[Style["Help?", FontFamily → "Comic Sans MS"], helpingScreen [], Imag
             }
        },
         ImageSize → {windowSizeX , windowSizeY },
        PlotRange → {{0, windowSizeX }, {0, windowSizeY }}
    ]},

    (* Palette Options *)
     WindowSize → {windowSizeX , windowSizeY },
    Saveable → False ,
    WindowTitle → "Fluid Mechanics for English Majors",
    WindowFloating → False
];
```

Helping Screen

```
In[ ]:= (* creates and displays the help screen in a new window (not done) *)
helpingScreen []:=Module[{},
(* NotebookClose []; *) (* will be uncommented later but for testing it closes the code file *
CreatePalette [
    {Graphics [{
             {
            Inset[ImageResize [Import["Pictures /background2 .png"],{960 ,540}],{Center ,Center },{C
             Inset[Style["Fluid Mechanics for English Majors", White , FontSize →40 ,FontFamil
             Inset[Button[Style["Start!",FontFamily →"Comic Sans MS"],workingScreen [],ImageSiz
            }
        },
        ImageSize →{960 ,540},PlotRange →{{0 ,960},{0 ,540}}
    ]},
    WindowSize →{960 ,540},
    Saveable →False ,
    WindowTitle →"Fluid Mechanics for English Majors",
    WindowFloating →False
    ]
]
(* NOTE- will contain text from final report *)
```

New Working Screens

Initialization

```
In[ ]:= (* resets the global variables and closes any screens that are open *)
resetWorkingScreens [] := Module[{},
    selected = "seq1";
    closeNotebooks [];
    inputDynamic = Null;
    isInfinite = Null;
    isDone = False;
    resetMasterPathList [];

    (* initializes the startup *)
    initializeWorkingScreens [];
]
```

```
In[ ]:= (* resets the inputDynamic *)
resetInputDynamic [] :=
    inputDynamic = Null;
```

In[ ]:=
```
(* creates the first sequence and sets up for input reservoir and pump creation *)
initializeWorkingScreens [] := Module[{},
    createSeq[Null, Null];
    askInfinite[];
]
```

In[ ]:=
```
(* prompts the user if they want to have an infinite setup *)
askInfinite[] := Module[{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic[];

    CreateDialog[{

        (* creates the text cells *)
        TextCell[Style["Would you like to have a general or an infinite setup?", FontFamily
        TextCell[Style["An infinite setup does not have boundaries on the reservoirs, while

        (* puts the buttons in a row below the text *)
        (* each button calls the respective input reservoir creation functions *)
        Row[{
            Button[Style["General!", FontFamily → "Comic Sans MS"], DialogReturn[askAndCrea
            Button[Style["Infinite!", FontFamily → "Comic Sans MS"], DialogReturn[askAndCre
        }]
    }]
]
```

In[ ]:=
```
(* this is called if the user chooses general at the beginning *)
(* prompts the user for the input reservoir options and creates one *)
askAndCreateNewGenInput [] := Module[{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic[];

    CreateDialog[{

        (* user instructions *)
        TextCell[Style["Create your input reservoir!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {h, d, p, dl, du} where", F
        TextCell[Style["h is the height of the reservoir in meters", FontFamily → "Comic
        TextCell[Style["d is the distance from the free surface to the ceiling in meters",
        TextCell[Style["p is the gauge pressure of the reservoir in pascals (0 if unpressur
        TextCell[Style["dl is the density of the lower fluid in kilograms/meter^3(for water
        TextCell[Style["du is the density of the upper fluid/gas in kilograms/meter^3 (0 if u
```

```
        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 5,
                DialogReturn[

                    (* creates the reservoir *)
                    mapNewGenInputReservoir[
                        "seq1",
                        Quantity[inputDynamic[[1]], "Meters"],
                        Quantity[inputDynamic[[2]], "Meters"],
                        Quantity[inputDynamic[[3]], "Pascals"],
                        Quantity[0, "Meters"^3],
                        Quantity[0, "Meters"^3],
                        Quantity[inputDynamic[[4]], "Kilograms"/"Meters"^3],
                        Quantity[inputDynamic[[5]], "Kilograms"/"Meters"^3]
                    ];

                    (* calls the pump creation function *)
                    askPump[];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=

```
(* this is called if the user chooses infinite at the beginning *)
(* prompts the user for the input reservoir options and creates one *)
askAndCreateNewInfInput [] := Module[{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Create your input reservoir!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {h, p, v} where", FontFamily
        TextCell[Style["h is the distance from the bottom of the reservoir to the top in m
        TextCell[Style["p is the gauge pressure of the reservoir in pascals (0 if unpressur
        TextCell[Style["d is the density of the fluid in kilograms/meter^3 (for water it is 1

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 3,
                DialogReturn [

                    (* creates the reservoir *)
                    mapNewInfInputReservoir [
                        "seq1",
                        Quantity[inputDynamic[[1]], "Meters"],
                        Quantity[inputDynamic[[2]], "Pascals"],
                        Quantity[inputDynamic[[3]], "Kilograms"/"Meters"^3]
                    ];

                    (* calls the pump creation function *)
                    askPump[];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
askPump [] :=  Module [{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* creates the text cells *)
        TextCell [Style ["Would you like a pump ?", FontFamily → "Comic Sans MS"]],

        (* puts the buttons in a row below the text *)
        (* the "Sure" button calls the respective input reservoir creation functions *)
        (* the "Nah" button starts the display and checks the height *)
        Row[{
            Button [Style ["Sure ", FontFamily → "Comic Sans MS"],
                DialogReturn [
                    If[isInfinite ,
                        askAndCreateNewInfPump [],
                        askAndCreateNewGenPump []
                    ]
                ]
            ],


            Button [Style ["Nah", FontFamily → "Comic Sans MS"],
                DialogReturn [
                    checkHeights [];
                    displaySelected [];
                ]
            ]
        }]
    }]
]
```

In[ ]:=
```
(* this is called after the user make a general input reservoir and says they want a pump *)
(* prompts the user for their pump options and creates one *)
askAndCreateNewGenPump [] := Module [{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
```

```
        TextCell[Style["Create your pump!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {e, w, h} where", FontFamily
        TextCell[Style["e is a number from 0-1 that describes how efficient the pump is",
        TextCell[Style["w is the maximum wattage of the pump", FontFamily → "Comic Sans M
        TextCell[Style["h is the head value of the pump in meters", FontFamily → "Comic S

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 3,
                DialogReturn[

                    (* creates the pump *)
                    addGenPump[
                        "seq1",
                        inputDynamic[[1]],
                        Quantity[inputDynamic[[2]], "Watts"],
                        Quantity[inputDynamic[[3]], "Meters"]
                    ];

                    (* starts up the display screen *)
                    checkHeights[];
                    displaySelected[];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ]
]
```

In[ ]:=
```
(* this is called after the user make an infinite input reservoir *)
(* prompts the user for their pump options and creates one *)
askAndCreateNewInfPump [] := Module[{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Create your pump!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the desired flow rate of your pump in meters^3/second"

        (* where the user inputs the flow rate *)
        InputField[Dynamic[inputDynamic ]],

        (* what the user presses when they're done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* will only work if the input is in the correct format *)
            If[NumberQ[inputDynamic ],
                DialogReturn [

                    (* creates the pump *)
                    addInfPump [
                        "seq1",
                        Quantity[inputDynamic , "Meters"^3/"Seconds"]
                    ];

                    (* starts the display screen *)
                    checkHeights [];
                    displaySelected [];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ]
]
```

In[ ]:=
```
startDisplay [] := Module [{},
     createAllStarOptions  [];
      createBuildOptions  [];
      displayNetwork ["Meters ", 16, "Comic  Sans  MS", 0.01];
]
```

Display selected

In[ ]:=
```
(* this  will  pop  up  a  dialog  notebook  that  constantly  displays  the  selected  *)
displaySelected [] := DynamicModule [{},
     selectedScreen  =
     CreateDialog [{TextCell [Dynamic [Style ["Selected : " <> selected , FontFamily  → "Comic  Sans
]
```

Create Build Options

In[ ]:=
```
(* creates  the  proper  options  screen  based  on what  is  selected  *)

createBuildOptions  [] :=
Module [
{
numOfComps ,
hasComps ,
optionsHeight ,
optionsBase  = 0,
nextColor  = White ,
buttonWidth  = windowSizeX  / 8,
buttonHeight  = windowSizeY  / 10,
counterStr
},

    (* closes  the  current  options  screen  if  there  is  one  *)
    NotebookClose [currentOptionsScreen  ];

        (* else  (if  selected  is  a  sequence )*)
        numOfComps  = Length [masterPathList [selected , "parts "]];

        (* this  is  used  to  determine  heights  of  the  buttons  and  if  there  is  a  select  comps
        If[numOfComps  ≥ 1,
            hasComps  = 1,
            hasComps  = 0
        ];

        (* this  makes  the  options  screen 's height  high  enough  depending  on  if  there  are  co
```

```
        and if the sequence already has an end *)
        If[SameQ[masterPathList [selected , "end"], Null],
            optionsHeight  = hasComps  * buttonHeight  + 7 * buttonHeight ,
            optionsHeight  = hasComps  * buttonHeight  + 4 * buttonHeight
        ];

        (* makes the options screen for a sequence *)
        currentOptionsScreen  =
            CreatePalette [
                {Graphics [

                    (* joins the "attach" buttons with the other buttons *)
                    Join[

                        (* if the selected sequence has no end, these "attach" buttons wil
                        If[SameQ[masterPathList [selected , "end"], Null],

                            (* joins the proper output reservoir buttons with the other bu
                            Join[
                                {
                                (* attaches another sequence to the selected *)
                                Inset[Button[Style["Attach sequence", FontFamily → "Comic

                                (* attaches a split to the selected *)
                                Inset[Button[Style["Attach split", FontFamily → "Comic San
                                },

                                (* determines which output reservoir button to include *)
                                If[isInfinite ,

                                    (* makes an infinite output reservoir *)
                                    {
                                    Inset[Button[Style["Attach output reservoir", FontFamil
                                    },

                                    (* makes a general output reservoir *)
                                    {
                                    Inset[Button[Style["Attach output reservoir", FontFamil
                                    }
                                ]
                            ], (* else *)
                            {Null}
                        ],

                        (* joins the select comp button with the add comp buttons *)
```

```
                    Join[

                        (* if there are components , this will make the select comp but
                        If[hasComps == 1,
                            {
                            Inset[Button[Style["Delete Comp", FontFamily → "Comic Sans
                            }, (* else *)
                            {Null}
                        ],
                        {
                        (* calls the prompt to add pipes *)
                        Inset[Button[Style["Add Pipes", FontFamily → "Comic Sans MS",

                        (* calls the prompt to add a valve *)
                        Inset[Button[Style["Add Valve", FontFamily → "Comic Sans MS",

                        (* calls the prompt to add a gate *)
                        Inset[Button[Style["Add Gate", FontFamily → "Comic Sans MS", F

                        (* pressed when the user does not want to add any more sequenc
                        Inset[Button[Style["Done", FontFamily → "Comic Sans MS", FontS
                            If[
                                allSeqsComplete [],
                                isDone = True;
                                processSysMain [];
                                NotebookClose [currentOptionsScreen ];
                                createRunOptions [],
                                (* else *)
                                DialogNotebook [{
                                    TextCell["Not all of your sequences have component
                                    DefaultButton [DialogReturn []]
                                }]
                            ],
                            ImageSize → {buttonWidth , buttonHeight }, Appearance → Non
                        }
                    ]
                ],

                (* makes the proper image size *)
                 ImageSize → {buttonWidth , optionsHeight - optionsBase }, PlotRange → {
            ]},

        (* makes the window the proper size *)
        WindowSize → {buttonWidth , optionsHeight - optionsBase },
        Saveable → False ,
```

```
            WindowTitle  → "Options ",
            WindowFloating  → False
            ];
]
```

In[ ]:=

```
(* prompts  the user  to select  a component  based  on index  *)
askAndDeleteComp [] :=
Module [
{
numOfComps  = Length [masterPathList [selected , "parts "]]
},
    (* resets  the  input  dynamic *)
    resetInputDynamic [];

    (* creates  the  dialog  notebook  *)
    CreateDialog [{

        (* user  instructions  *)
        TextCell [Style ["Please  insert  the  index  of the  component  you would  like  to delete "
        TextCell [Style ["(you  currently  have  " <> ToString [numOfComps ] <> " components  in thi

        (* where  the  user  inputs  the  index  *)
        InputField [Dynamic [inputDynamic ]],

        (* the button  the  user  presses  when  they  are  done  *)
        Button [Style ["Done ", FontFamily  → "Comic  Sans  MS"],

            (* only  works  if the  input  is in the  correct  format  *)
            If[NumberQ [inputDynamic ] && inputDynamic  ≤ Length [masterPathList [selected , "part

                (* changes  selected  to the  correct  form  and  resets  the  options  screen  *)
                DialogReturn [
                    deleteComp [inputDynamic ];
                    createBuildOptions  [];
                ]
            ], Method  → "Queued "
        ]
    }]
]
```

In[ ]:=
```
(* deletes the selected component *)
deleteComp[compIndex_] :=
Module[
{
(* sets newMap to the current parts map *)
newMap = masterPathList[selected, "parts"]
},
    (* drops the selected component from newMap *)
    newMap = Drop[newMap, {compIndex, compIndex}];

    (* re-maps the selected to the newMap *)
    masterPathList[selected, "parts"] = newMap;
]
```

In[ ]:=

```
(* prompts the user to create a new split *)
askAndCreateSpl [] :=
Module[{newSplKey},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Please insert the following in the form {{t, m, b}, merge pairs list
        TextCell[Style["t is the number of sequences above the starting sequence ", FontFami
        TextCell[Style["m is 1 if the starting sequence continues and 0 if not", FontFamily
        TextCell[Style["b is the number of sequences below the starting sequence ", FontFami
        TextCell[Style["Please refer to the merge pairs help below if you have questions ",

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic ]],

        (* calls the merge pairs help *)
        Button[
            Style["Merge Pairs Help", FontFamily → "Comic Sans MS"],
            mergePairsHelp []
        ]

        (* the button the user presses when they are done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* only works if the input is in the correct format *)
            If[ListQ[inputDynamic ] && Length[inputDynamic ] == 2 && Length[inputDynamic [[1]]] ==

                (* creates a split and resets the options screen *)
                DialogReturn [
                    createSpl[selected , inputDynamic [[1]], inputDynamic [[2]]];
                    checkHeights [];
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* prompts the user for how many pipes they want to add and the diameter of them *)
askAndCreatePipes [] :=
Module[{},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Please insert the number of pipes and the desired diameter in meter
        TextCell[Style["{number of pipes, diameter of pipes}", FontFamily → "Comic Sans MS"

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic ]],

        (* the button they press when they are done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* only works if the input is in the proper format *)
            If[ListQ[inputDynamic ] && Length[inputDynamic ] == 2,

                (* calls the add pipes function and resets the options screen *)
                DialogReturn [
                    addPipes[selected , inputDynamic [[1]], Quantity[inputDynamic [[2]], "Meters"
                    createBuildOptions [];
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* prompts the user to input the diameter of a new valve *)
askAndCreateValve [] :=
Module[{},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{
        (* user instructions *)
        TextCell[Style["Please insert the desired diameter in meters", FontFamily → "Comic

        (* where the user inputs the diameter *)
        InputField[Dynamic[inputDynamic ]],

        (* the button they press when they're done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* only works if the input is in the correct format *)
            If[NumberQ[inputDynamic ],

                (* adds the valve and resets the options screen *)
                DialogReturn [
                    addValve[selected , Quantity[inputDynamic , "Meters"]];
                    createBuildOptions [];
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* prompts the user to input the diameter of a new gate *)
askAndCreateGate [] :=
Module [{},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{
        (* user instructions *)
        TextCell [Style ["Please insert the desired diameter in meters ", FontFamily → "Comic

        (* where the user inputs the diameter *)
        InputField [Dynamic [inputDynamic ]],

        (* the button they press when they're done *)
        Button [Style ["Done ", FontFamily → "Comic Sans MS"],

            (* only works if the input is in the correct format *)
            If[NumberQ [inputDynamic ],

                (* adds the gate and resets the options screen *)
                DialogReturn [
                    addGate [selected , Quantity [inputDynamic , "Meters "]];
                    createBuildOptions [];
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* this is called if the user chooses general at the beginning and tries to create an outpu
(* prompts the user for the output reservoir options and creates one *)
askAndCreateGenOutput [] := Module [{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell [Style ["Create an output reservoir !", FontFamily → "Comic Sans MS"]],
        TextCell [Style ["Please insert the following in the form {d, p, bv, mv, du} where ",
        TextCell [Style ["d is the distance from the free surface of the liquid to the top i
        TextCell [Style ["p is the gauge pressure of the reservoir in pascals (0 if unpressur
```

```
        TextCell[Style["bv is the volume of the lower liquid in meters^3", FontFamily → "C
        TextCell[Style["mv is the max volume of the reservoir in meters^3", FontFamily → "
        TextCell[Style["du is the density of the upper fluid/gas in kilograms/meter^3 (0 if u

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 5,
                DialogReturn[

                    (* creates the reservoir *)
                    mapNewGenOutputReservoir[
                        selected,
                        Quantity[0, "Meters"],
                        Quantity[inputDynamic[[1]], "Meters"],
                        Quantity[inputDynamic[[2]], "Pascals"],
                        Quantity[inputDynamic[[3]], "Meters"^3],
                        Quantity[0, "Meters"^3],
                        Quantity[inputDynamic[[4]], "Meters"^3],
                        Quantity[inputDynamic[[5]], "Kilograms"/"Meters"^3]
                    ];
                    checkHeights[];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
(* this is called if the user chooses infinite at the beginning and tries to create an outp
(* prompts the user for the output reservoir options and creates one *)
askAndCreateInfOutput [] := Module[{},

    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Create an output reservoir!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {p, v} where", FontFamily →
        TextCell[Style["p is the pressure of the reservoir in pascals (0 if unpressurized )"
        TextCell[Style["v is the volume of the reservoir in meters^3", FontFamily → "Comic

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 2,
                DialogReturn [

                    (* creates the reservoir *)
                    mapNewUnboundedOutputReservoir [
                        selected ,
                        Quantity[1, "Meters"],
                        Quantity[inputDynamic[[1]], "Pascals"],
                        Quantity[inputDynamic[[2]], "Meters"^3]
                    ];
                    checkHeights [];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
(* This goes through the masterPathList and checks if any sequences don't have heights *)
checkHeights [] :=
Module[
{
counter,
nextKey,
noHeightFound = False
},
    (* goes through masterPathList and looks for any sequences without a height *)
    For[counter = 1, counter ≤ Length[masterPathList], counter++,
        nextKey = Keys[masterPathList][[counter]];

        (* if it finds one, it asks the user to set the height *)
        If[masterPathList[nextKey]["attr"]["height"] === Null,
            askSetHeight[nextKey];
            noHeightFound = True;
            Break[];
        ]
    ];
    If[!noHeightFound,
        startDisplay[];
    ]
]
```

In[ ]:=
```
(* prompts the user to input a height for a sequence *)
askSetHeight[key_] :=
Module[
{},
    (* resets the input dynamic *)
    resetInputDynamic[];

    (* returns true if it is supposed to continue the previous sequence *)
    If[findHighMidLow[key] == 2,
        masterPathList[key, "attr", "height"] = masterPathList[masterPathList[masterPathList
        checkHeights[],

        (* creates the dialog notebook *)
        CreateDialog[{

            (* user instructions *)
            TextCell[Style["Please insert the desired height of " <> key <> " in meters",

            (* where the user inputs the height *)
            InputField[Dynamic[inputDynamic]],
```

```
            (* the button they press when they're done *)
            Button[Style["Done", FontFamily → "Comic Sans MS"],

                (* returns true if the height is supposed to be be high and the input is *
                If[NumberQ[inputDynamic] && findHighMidLow[key] == 1 && inputDynamic > Quant
                    setHeight[key, Quantity[inputDynamic, "Meters"]];
                ];

                (* returns true if the height is supposed to be be low and the input is *)
                If[NumberQ[inputDynamic] && findHighMidLow[key] == 3 && inputDynamic < Quant
                    setHeight[key, Quantity[inputDynamic, "Meters"]];
                ];

                (* returns true if it doesn't matter what the height is *)
                If[NumberQ[inputDynamic] && findHighMidLow[key] == 4,
                    setHeight[key, Quantity[inputDynamic, "Meters"]];
                ];

                (* checks the heights and closes the dialog window *)
                checkHeights[];
                DialogReturn[Null],
                Method → "Queued"
            ]
        }]
    ]
]
```

In[ ]:=
```
(* returns an index of sorts that is used in askSetHeight *)
(* a 1 cooresponds to a sequence that is intended to be above the previous sequence *)
(* a 2 cooresponds to a sequence that is intended to be a continuation of the previous sequ
(* a 3 cooresponds to a sequence that is intended to be below the previous sequence *)
(* a 4 cooresponds to a sequence that is not restricted *)
findHighMidLow [key_] :=
Module [
{
startKey = masterPathList [key, "start"]
},
    If[key == "seq1",
        Return[4]
    ];

    (* determines if the key before the input is a split *)
    If[getPathType [startKey] == "spl",

        (* returns 1 if above, 2 if continue, and 3 if below *)
        Return[Position [masterPathList [masterPathList [key, "start"], "branches"], key]][[1, 1,

        (* else *)
        Return[4]
    ]
 ]
```

Create New Run Options

In[ ]:=
```
(* creates the run options menu *)
createRunOptions [] :=
Module [
{
buttonWidth = windowSizeX / 8,
buttonHeight = windowSizeY / 5,
windowHeight
},
    (* closes the current Run Options screen if there is one *)
    NotebookClose [currentRunOptionsScreen ];

    (* there will be at least 4 buttons *)
    windowHeight = 2 * buttonHeight ;

    (* there will be two more buttons if the user chose to create a general setup *)
    If[!isInfinite ,
        windowHeight = windowHeight + buttonHeight
    ];
```

```
    (* creates the run options screen *)
    currentRunOptionsScreen   =
        CreatePalette [
            {Graphics [
                Join[
                    If[!isInfinite ,
                        (* these are the extra buttons if the user chose a general setup *
                        {
                        (* this prompts the user to modify the pressure of a reservoir *)
                        Inset[Button[Style["Modify gauge pressure ", FontFamily →"Comic Sans

                        (* this creates a dialog notebook that shows the required head val
                        Inset[Button[Style["Calculate required head value ", FontFamily →"Co
                        },
                        {Null}
                    ],
                    {
                    (* this calls the modify pump functions for the user to change their p
                    Inset[Button[Style["Modify pump ", FontFamily →"Comic Sans MS", FontSize

                    (* this allows the user to calculate the differential pressure between
                    Inset[Button[Style["Transfer fluid ", FontFamily →"Comic Sans MS", FontSt

                    (* this allows the user to calculate the differential pressure between
                    Inset[Button[Style["Calculate differential pressure ", FontFamily →"Comic

                    (* this allows the user to change a valve or gate *)
                    Inset[Button[Style["Change comp diameter ", FontFamily →"Comic Sans MS",

                    }
                ]
                ,

                (* makes the graphics the proper size *)
                 ImageSize  → {2 * buttonWidth , windowHeight }, PlotRange  → {{0, 2 * buttonWid
            ]},
            (* palette options *)
            WindowSize  → {2 * buttonWidth , windowHeight },
            Saveable  → False ,
            WindowTitle  → "Run Options ",
            WindowFloating  → False
        ]
]
```

In[ ]:=

```
(* prompts the user for which component they would like to change *)
askComp [] :=
Module [{},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell [Style ["Please input the index of the component you would like to change",

        (* where the user inputs the diameter *)
        InputField [Dynamic [inputDynamic ]],

        (* the button the user presses when done *)
        Button [Style ["Done", FontFamily → "Comic Sans MS"],
        DialogReturn [
            inputDynamic = {masterPathList [selected , "parts"][[inputDynamic , 1]], inputDynamic
                (* only works if input is in the correct format *)
                If [NumberQ [inputDynamic [[2]]] && inputDynamic [[2]] ≤ Length [masterPathList [selec

                    Switch [inputDynamic [[1]],

                        "V",
                        (* prompts the user to change the valve diameter *)
                        askChangeValveDiameter [inputDynamic [[2]]],

                        "G",
                        (* if it is a gate, negates it *)
                        negateGate [selected , inputDynamic [[2]]];
                        processSysMain [];
                    ]
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* if the comp inputted is a valve, this asks the user what they want to change the diamete
askChangeValveDiameter [valveIndex_] :=
Module[{},
    (* resets the input dynamic *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Please insert the new valve diameter in meters", FontFamily → "Com

        (* where the user inputs the diameter *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* only works if input is in the correct format *)
            If[NumberQ[inputDynamic],

                (* changes the valve diameter *)
                DialogReturn [
                    changeValveDiameter [selected, valveIndex, Quantity[inputDynamic, "Mete
                    processSysMain []
                ]
            ]
        ]
    }]
]
```

In[ ]:=
```
(* finds and displays the required head value *)
showRequiredHeadValue []:=
CreateDialog [{
    TextCell["The required head value is " <> ToString@findHeadVal []],
    DefaultButton [DialogReturn []]
}]
```

In[ ]:=
```
(* asks the user which reservoir they want to change the pressure of, and what they want t
askModifyPressure [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Modify the gauge pressure ", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {sequence , p} where ", FontFa
        TextCell[Style["sequence is the sequence that the desired reservoir is attached to
        TextCell[Style["p is the new pressure ", FontFamily → "Comic Sans MS"]],

        (* where the user inputs their specifications *)
        InputField[Dynamic[inputDynamic ]],

        (* the button the user presses when they're finished *)
        Button[Style["Done ", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic ] && Length[inputDynamic ] == 2,
                DialogReturn [

                    (* changes the pressure of the attached reservoir *)
                    changeGaugePressureGen [
                        inputDynamic [[1]],
                        Quantity[inputDynamic [[2]], "Pascals "]
                    ];
                    processSysMain [];
                ]

            (* required for proper button action *)
            ], Method → "Queued "
        ]
    }
    ];
]
```

In[ ]:=
```
(* calls the correct water transfer function if the setup is infinite *)
pickTransferFluid []:=
If[isInfinite ,
    askInfWaterTransfer [],
    askGenWaterTransfer []
]
```

In[ ]:=
```
(* asks the user for what time in seconds they want to shift the simulation to *)
askInfWaterTransfer [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Water Transfer!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert a time in seconds", FontFamily → "Comic Sans MS"]],

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic ]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[NumberQ[inputDynamic ],
                DialogReturn [

                    (* automatically fills the outputs *)
                    fillUnOutputs [Quantity[inputDynamic , "Seconds"]]
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
(* asks the user for what time in seconds they want to shift the simulation to *)
askGenWaterTransfer [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Water Transfer!", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert a time in seconds", FontFamily → "Comic Sans MS"]],

        (* where the user inputs their options *)
```

```
        InputField [Dynamic [inputDynamic ]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[NumberQ [inputDynamic ],
                DialogReturn [

                    (* determines if the pump is strong enough for the system to transfer
                    If[headValCheck [],

                        (* if true, it transfers fluid based on how many seconds the user
                        fillGenOutputs [Quantity [inputDynamic , "Seconds "]],

                        (* if false it give this error message *)
                        If[containsPump ["seq1"],
                            CreateDialog [{
                                TextCell [Style["Your pump's too weak! Get a better one!", F
                                DefaultButton [DialogReturn []]
                            }],
                            CreateDialog [{
                                TextCell [Style["Gravity can't overcome the system's resist
                                DefaultButton [DialogReturn []]
                            }]
                        ]
                    ]
                ]

            (* required for proper button action *)
            ], Method → "Queued "
        ]
    }
    ];
]
```

In[ ]:=

```
(* picks the right pump modification method based on if the setup is infinite *)
pickPumpAsk [] :=
If[isInfinite ,
    askModifyInfPump [],
    askModifyGenPump []
]
```

In[ ]:=
```
(* asks the user for the new parameters they want to change the pump to *)
askModifyGenPump [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Modify the pump", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the following in the form {e, w, h} where", FontFamily
        TextCell[Style["e is a number from 0-1 that describes how efficient the pump is",
        TextCell[Style["w is the maximum wattage of the pump", FontFamily → "Comic Sans M
        TextCell[Style["h is the head value of the pump", FontFamily → "Comic Sans MS"]],

        (* where the user inputs their options *)
        InputField[Dynamic[inputDynamic]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic] && Length[inputDynamic] == 3,
                DialogReturn [

                    (* creates the pump *)
                    modifyGenPump [
                        "seq1",
                        inputDynamic [[1]],
                        Quantity[inputDynamic [[2]], "Watts"],
                        Quantity[inputDynamic [[3]], "Meters"]
                    ];
                    processSysMain [];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
(* asks the user for the new parameters they want to change the pump to *)
askModifyInfPump [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    (* creates the dialog notebook *)
    CreateDialog [{

        (* user instructions *)
        TextCell[Style["Modify your pump", FontFamily → "Comic Sans MS"]],
        TextCell[Style["Please insert the desired flow rate of your pump", FontFamily → "

        (* where the user inputs the flow rate *)
        InputField[Dynamic[inputDynamic ]],

        (* what the user presses when they're done *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* will only work if the input is in the correct format *)
            If[NumberQ[inputDynamic ],
                DialogReturn [

                    (* creates the pump *)
                    modifyInfPump [
                        "seq1",
                        Quantity[inputDynamic , "Meters"^3/"Seconds"]
                    ];
                    processSysMain [];
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

In[ ]:=
```
(* asks the user which two points they want to calculate the differential pressure of, and
askFindDiffPress [] := Module[{},
    (* clears the input dynamic so nothing will be in the input box *)
    resetInputDynamic [];

    CreateDialog [{

        (* user instructions *)
```

```
        TextCell[Style["Differential pressure is the difference in pressure between two po
        TextCell[Style["Please insert the following in the form {sequence , sequence2 }", Fon
        TextCell[Style["For example : {\"seq1\", \"seq8\"}", FontFamily → "Comic Sans MS"]],

        (* where the user inputs the sequences *)
        InputField[Dynamic[inputDynamic ]],

        (* the button the user presses when they're finished *)
        Button[Style["Done", FontFamily → "Comic Sans MS"],

            (* this will not work if it is not in the correct format *)
            If[ListQ[inputDynamic ] && Length[inputDynamic ] == 2,
                DialogReturn [

                    (* finds the differential pressure and displays it in another dialog r
                    CreateDialog [{
                        TextCell["The differential pressure is "],
                        TextCell [
                            UnitConvert [
                                findDiffPress [
                                    getLiqDensity [isInfinite ],
                                    inputDynamic [[1]],
                                    inputDynamic [[2]]
                                ],
                                "Pascals"
                            ]
                        ],
                        DefaultButton [DialogReturn []]
                    }]
                ]

            (* required for proper button action *)
            ], Method → "Queued"
        ]
    }
    ];
]
```

Create All-Star Options

In[ ]:=

```
createAllStarOptions [] :=
Module[
{
buttonWidth = windowSizeX / 8,
```

```
buttonHeight  = windowSizeY  / 5,
windowHeight ,
windowWidth
},
     NotebookClose [currentAllStarScreen ];

     (* there  are  6 all  star  buttons , 3 rows , 2 columns  *)
     windowHeight  = 3 * buttonHeight ;
     windowWidth  = 2 * buttonWidth ;
     (* creates  the  run  options  screen  *)
     currentAllStarScreen   =
          CreatePalette [
               Graphics [
                       {
                       (* this  *)
                       Inset[Button[Style["Save ", FontFamily →"Comic  Sans  MS", FontSize  → 20],

                       (* this  *)
                       Inset[Button[Style["Load ", FontFamily →"Comic  Sans  MS", FontSize  → 20],

                       (* this  *)
                       Inset[Button[Style["Show  global  properties ", FontFamily →"Comic  Sans  MS

                       (* this  *)
                       Inset[Button[Style["Show  sequence  properties ", FontFamily →"Comic  Sans

                       (* this  *)
                       Inset[Button[Style["Main  Menu ", FontFamily →"Comic  Sans  MS", FontSize  →

                       (* this  *)
                       Inset[Button[Style["Reset ", FontFamily →"Comic  Sans  MS", FontSize  → 20]
                       }
                   ,

                   (* makes  the  graphics  the  proper  size  *)
                    ImageSize  → {windowWidth , windowHeight }, PlotRange  → {{0, windowWidth }, {0
               ],
               (* palette  options  *)
               WindowSize  → {windowWidth , windowHeight },
               Saveable  → False ,
               WindowTitle  → "Run  Options ",
               WindowFloating  → False
          ]
]
```

Create Properties- local+global

In[ ]:=
```
(* uses displaySeqProps  to retrieve  text  for  a given  sequence  that outlines  its properties
createPropertiesScreen  [seqKey_]:= Module[
{},
    (* close  the properties  screen  that  is currently  open  *)
    NotebookClose [currentPropertiesScreen  ];

    (* if seq, display  its properties  and  set  it as  the current  props  screen  *)
    If[StringContainsQ [seqKey , "seq"],
        currentPropertiesScreen   = CreateDialog [Join[displaySeqProps [seqKey], {DefaultButton [
    ]
]
```

In[ ]:=
```
(* Display  the global  properties  of the network  *)
createGlobalPropertiesScreen  [] := Module[
  {},

  (* close  the properties  screen  that  is currently  open  *)
    NotebookClose [currentPropertiesScreen ];

    currentPropertiesScreen  = CreateDialog [{
     TextCell["Gravity :  " <> ToString[globalConstants ["gravity "]]],
     TextCell["Temperature :  " <> ToString[globalConstants ["environmentTemp "]]],
     DefaultButton [DialogReturn []]
    }]
 ]
```

In[ ]:=
```
(* calls  the seq  from masterPathList  and  formats  the
 keys  and values  into legible  information  for display  *)
displaySeqProps [seqName_] := Module[
  {seqKey  = seqName ,
   flowRate ,
   avgVel ,
   avgDia ,
   height ,
   partsOut ,
   start ,
   name ,
   end ,
   outTexts  = {},
```

```
  partsList,
  formattedParts = ""(* makes parts list easier to read *)
 },

 partsList = masterPathList[seqKey, "parts"];
 Do[
      formattedParts = formattedParts <> "\n" <> ToString[part];
  ,{part, partsList}
 ];

 (* Make text cell for attributes *)
 name = TextCell["Selected Seq: " <> seqKey];
 flowRate =
  TextCell["Flow Rate: " <> ToString[masterPathList[seqKey, "attr", "flowRate"]]];
 avgVel = TextCell["Average Velocity: " <>
    ToString[masterPathList[seqKey, "attr", "avgVelocity"]]];
 avgDia = TextCell["Average Diameter: " <>
    ToString[masterPathList[seqKey, "attr", "avgDiameter"]]];
 height = TextCell["Height: " <> ToString[
     masterPathList[seqKey, "attr", "height"]]];
 partsOut = TextCell["Parts: " <> formattedParts];
 start = TextCell["Start: " <> ToString[masterPathList[seqKey, "start"]]];
 end = TextCell["End: " <> ToString[masterPathList[seqKey, "end"]]];

 (* Add the property read outs to the outTexts list *)
 AppendTo[outTexts, name];
 AppendTo[outTexts, flowRate];
 AppendTo[outTexts, avgVel];
 AppendTo[outTexts, avgDia];
 AppendTo[outTexts, height];
 AppendTo[outTexts, partsOut];
 AppendTo[outTexts, start];
 AppendTo[outTexts, end];

 (* return the text cells *)
 Return[outTexts]
]
```

Merge Pairs Help

In[ ]:=
```
mergePairsHelp [] := Module [{},

]
(* will open seperate notebook *)
```

Notebook functions

In[ ]:=
```
(* closes all notebooks *)
closeNotebooks [] := Module [{},
    NotebookClose [currentDisplayScreen ];
    NotebookClose [currentOptionsScreen ];
    NotebookClose [currentPropertiesScreen ];
    NotebookClose [currentRunOptionsScreen ];
    NotebookClose [currentAllStarScreen ];
    NotebookClose [selectedScreen ];
]
```

Network Display

In[ ]:=
```
(* Uses the functions from the "Controller"
 section to display a basic view of the network *)
(* Note- height is how high above the given sequence a button is *)
displayNetwork [units_, fontSize_, fontType_, lineThickness_] := Module[
  {seqLocs,
   layerLength = 5, (* value for layerLength doesn't matter because the height
    and yScale will always be the correct proportion using division *)
   seqPointSets,
   buttons,
   splPoints,
   merPoints,
   yScale,
   height,
   splGfx, (* graphics that show splits *)
   merGfx, (* graphics that show merges *)
   inputGfx, (* rectangle that represents the input reservoir *)
   pumpGfx = Null, (* disk that represents the pump -if present *)
   outputGfx (* disks that represent outputs *)
  },

  (* closes the current display screen *)
  NotebookClose [currentDisplayScreen ];
```

```
height = layerLength / 5 - 0.5; (* vertical offset of buttons *)
yScale = layerLength / 5; (* multiplier on y-coords*)

(* determine the start and end locs for each sequence *)
seqLocs = seqNameAndPos[layerLength, N[yScale], units];

seqPointSets = allSeqPoints[seqLocs];
buttons = getSeqButtons[seqLocs, fontSize, fontType, layerLength, height];

(* determine points for the vertical
 lines that connect sequences at splits and merges *)
splPoints = splPtsAll[seqLocs];
merPoints = merPtsAll[seqLocs];

(* get graphics to mark splits and merges *)
splGfx = Join[getSplRects[seqLocs, layerLength, Red],
  getSplLabels[seqLocs, layerLength, Black, fontType, fontSize]];
merGfx = Join[getMerRects[seqLocs, layerLength, Green],
  getMerLabels[seqLocs, layerLength, Black, fontType, fontSize]];

(* get rectangle for input reservoir *)
inputGfx = getInputRect[seqLocs, Darker[Blue], layerLength, 0.2];

(* get circle for pump if present *)
If[containsPump["seq1"],
     pumpGfx = getPumpCirc[seqLocs, Yellow, 5, 0.4];
];

(* get circles to represent outputs if present *)
outputGfx = createOutputCircs[seqLocs, Purple, 0.8];

(* output- opens up document. Allows for scrolling *)
currentDisplayScreen =
 CreateDocument[
  Graphics[{
    {Thickness[lineThickness], Line[seqPointSets]},
    {Thickness[lineThickness], Line[splPoints]},
    {Thickness[lineThickness], Line[merPoints]},
    buttons,
    splGfx,
    merGfx,
    inputGfx,
    pumpGfx,
```

```
      outputGfx
     },
     (* Doc options *)
     ImageSize → windowSizeX * 3 / 4],
    WindowSize → {windowSizeX * 3 / 4, windowSizeY * 3 / 4},
    WindowTitle → "Display",
    Saveable → False
   ];

]
```

Start-up

In[ ]:=
```
(* starts the program *)
bootUp[] :=
  initializeLists [];
    initializeUIglobals [];
    resetGlobalConstants [];
    startingMenu [];
```