Asynch Documentation

Release 1.0

Scott Small, Samuel Debionne

Jul 29, 2018

About Asynch

Asynch is numerical library for solving differential equations with a tree structure. Emphasis is given to hydrological river basin models.

The main documentation for Asynch is organized into a couple of sections:

CHAPTER 1

About

ASYNCH is a software package for solving large systems of ordinary differential equations with a tree structure. At its heart, ASYNCH is a library that can be mixed with other software to allow users greater fexibility. It is written entirely in the C programming language. ASYNCH currently supports an interface for programs written in C/C++ or Python.

Although ASYNCH is a library, it does come prepackaged with simple programs for performing basic simulations. These programs can be easily modified to increase their fexibility. ASYNCH is designed for a distributed memory computer. The Message Passing Interface (MPI) is used for inter-process communication. ASYNCH also contains routines for downloading and uploading data from and to PostgreSQL databases.

ASYNCH was designed with hydrological applications in mind. Much of the naming conventions and sample models reflect this. However, there is no reason why ASYNCH cannot be applied to other problems with ordinary differential equations having tree structures.

The novelty of this solver is that it uses an asynchronous integrator to solve the differential equations. Further, the communication between processes occurs asynchronously. Details about how this works can be found in *Small, et. al. An Asynchronous Solver for Systems of ODEs Linked by a Directed Tree Structure, Advances in Water Resources, 53, March 2013, 23-32*.

CHAPTER 2

Releases Notes

ASYNCH release notes provide information on the features and improvements in each release. This page includes release notes for major releases and minor (bugfix) releases. If you are upgrading from an earlier version of ASYNCH, you will find essential information in the Breaking Changes associated with the relevant release notes.

2.1 Version 1.4

2.1.1 Breaking Changes

The signature of the function that implements the differential equations has an extra parameter `max_num_dim`. This is required because data assimilation uses a variable number of equations at links, the maximum number of degree of freedom beeing known at runtime.

```
typedef void (DifferentialFunc) (
  double t,
  const double * const y_i, unsigned int num_dof,
  const double * const y_p, unsigned short num_parents, unsigned int max_num_dof,
  const double * const global_params,
  const double * const params,
  const double * const forcing_values,
  const QVSData * const qvs,
  int state,
  void *user,
  double *ans);
```

2.1.2 New Features

This release introduces *Data Assimilation*.

An additional, more compact HDF5 format for the time series is available, see option `6` of *Time Series Location*.

Three new models are available:

Constant Runoff Model `195`:

- based on `190` with precipitation forcing replaced by runoff and infiltration forcings to describe spatial variability of such processes, and including an additional state for rainfall accumulation.

Top Layer Model `256`:

- based on `254`;
- has a new state variable `ans[7]` : accumulated evapotranspiration;
- has one more toplayer storage to link parameter `k_tl = global_params[12]`;
- has a flux component from toplayer storage to channel that represents the interflow $q_{tl} = k_{tl} s_t$.

Top Layer Model `257`:

- based on `256` but channel velocity is spatialized as a function of the Horton order;
- the Horton order is an additional local parameter.

2.2 Version 1.3

This release is the result of a loong run profiling and optimization work.

Memory footprint improvements, better data structure reduce the memory usage typically by a factor two. If you have memory available you may want to increase the number of buffers, see *Buffer Sizes*.

Performance improvements, ASYNCH runs about 30% faster.

2.3 Version 1.2

2.3.1 Breaking Changes

The global file structure has changed. The time of the simulation is now given in absolute time, see *Simulation period*.

The snapshots in HDF5 format has changed, see *Ini HDF5 Files*.

2.3.2 New Features

Time series outputs can be written in HDF5 format, see *Time Series Location*.

CHAPTER 3

Terminology and Definitions

In this section, we define terminology and conventions used throughout this documentation.

3.1 Ordinary differential equation

By ordinary differential equation (ODE), we mean a system of differential equations with all unknowns dependent upon time. These independent variables are referred to as *states*. The ASYNCH solvers can only be used for ODEs with the time derivatives of the states isolated. See Section [sec: example models] for examples. Partial differential equations (PDEs) have states that depend upon multiple independent variables, and cannot be outright solved with the ASYNCH solvers.

3.2 Algebraic equations

Algebraic equations are equations which do not possess derivatives or integrals of states with respect to independent variables. If a system of equations consists of a mixed collection of ODEs and algebraic equations, then the system is referred to as a system of *differential-algebraic equations (DAEs)*. See Section [sec: linear reservoir hydrological model] for an example.

3.3 Systems of ODEs

The ASYNCH solvers are intended for solving systems of ODEs (and limited DAEs) with a tree structure. By tree structure, we mean that the system of equations can be clustered together in a way so that dependence upon clusters is

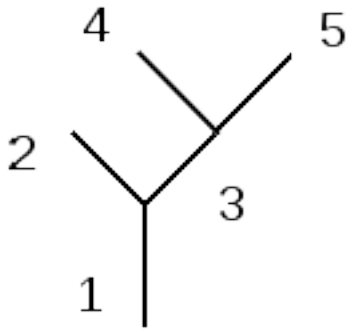
one way. As an example, consider the system of differential equations

$$\begin{aligned}
\frac{dy_1}{dt} &= -y_1 + y_2 + y_3 + y_4 + y_5 + y_6 \\
\frac{dy_2}{dt} &= -y_2 + y_3 + y_4 + y_5 + y_6 \\
\frac{dy_3}{dt} &= -y_3 + y_4 \\
\frac{dy_4}{dt} &= -y_4 \\
\frac{dy_5}{dt} &= -y_5 + y_6 + y_7 + y_8 + y_9 + y_{10} \\
\frac{dy_6}{dt} &= -y_6 + y_7 + y_8 + y_9 + y_{10} \\
\frac{dy_7}{dt} &= -y_7 + y_8 \\
\frac{dy_8}{dt} &= -y_8 \\
\frac{dy_9}{dt} &= -y_9 + y_{10} \\
\frac{dy_{10}}{dt} &= -y_{10}
\end{aligned}$$

This system can be clustered into the pairs:

$$1 : [y_1, y_2] \quad 2 : [y_3, y_4] \quad 3 : [y_5, y_6] \quad 4 : [y_7, y_8] \quad 5 : [y_9, y_{10}]$$

The dependency of these clusters is one way in the sense that the equations for the variables in cluster 3 depend only upon the variables in clusters 4 and 5, the equations for the variables in cluster 1 depend only upon clusters 2 and 3, and the equations for the variables in clusters 2, 4, and 5 do not depend upon any other cluster. Graphically, this system of equations could be viewed to have the tree structure given in:



3.4 Links

In the ASYNCH source code and in this document, the clusters of variables are referred to as **links**. Every link has a positive integer attached to it that identifies the link uniquely. This number is referred to as the **link ID**. The unknowns in the clusters are referred to as the **states of the link**.

3.5 Peakflow

The term **peakflow** is used frequently with ASYNCH. This is a hydrological term, but it can take meaning in other applications. A peakflow of a state is the maximum value of that state over a period of time.

3.6 Model

The term **model** is used frequently in technical disciplines to refer to a (usually mathematical) procedure or computer program for obtaining a description for how a natural process works. In this documentation, a model refers to the collection of equations, **but not their solution**. The entire purpose of ASYNCH is to solve ODEs efficiently, thus producing their solution. No model created for ASYNCH should ever contain time steps. The equations (_explicit-example) are an example of a model.

3.7 State vector

Each model has a number of states that are to be determined at each link. At a specific time, these states are stored in a vector, known as a **state vector**. Similarly, the value of the differential and algebraic equations at a particular time and state are stored in an **equation-value vector**. There is a correspondence between state and equation-value vectors at each link. For example, if the underlying model is a system of differential equations, then the derivative of the first state in a state vector is stored in the first entry of the equation-value vectors, the derivative of the second state in a state vector is stored in the second entry of the equation-value vectors, etc.

As an example, a state vector for equations ([eq: explicit example]) will look like

$$[y_1,\ y_2,\ y_3,\ y_4,\ y_5,\ y_6,\ y_7,\ y_8,\ y_9,\ y_{10}]^T$$

while an equation-value vector will look like

$$\left[\frac{dy_1}{dt},\ \frac{dy_2}{dt},\ \frac{dy_3}{dt},\ \frac{dy_4}{dt},\ \frac{dy_5}{dt},\ \frac{dy_6}{dt},\ \frac{dy_7}{dt},\ \frac{dy_8}{dt},\ \frac{dy_9}{dt},\ \frac{dy_{10}}{dt}\right]^T$$

3.8 Discontinuities

Some models contain discontinuities in their equations. This can include not only the equation themselves, but also in their derivatives. ASYNCH supports intelligent handling of these abrupt changes when the equations can be described piecewise. For example, the differential equation

$$\frac{dy}{dt} = \begin{cases} -(y-5) + f(t), \text{for } y < 5 \\ -(y-5)^2 + f(t), \text{for } y \geq 5 \end{cases}$$

with some forcing function f, has a discontinuity in the derivative of y. Each component of the piecewise function is referred to as a **discontinuity state**.

3.9 Computing

A few computer related terms are thrown around frequently in this document (and in parallel computing in general) that are worth describing.

- A **node** is a physical computer. This includes any related hardware inside the computer (cores, memory, hard disks, etc). The term **cluster** is used to refer to an interconnected group of nodes.
- A **core**, **processor**, or **slot** (in the case of Iowa HPC resources) are the physical processing units in computers. These are the components that actually perform computations.
- A **process** is an instance of a running program. ASYNCH uses MPI processes to achieve parallelism. This means several instances of ASYNCH are run simultaneously, each able to communicate with each other. Because this document refers to ASYNCH, the phrase **MPI process** is interchangeable with process. It is simply used to emphasize that communication occurs with MPI.
- A **thread** is a sequence of instructions (code) to a processor. **Multithreading** is when many threads are created by a program and potentially executed simultaneously on a single node. ASYNCH does not currently support multithreading explicitly (it may occur "behind the scenes" in MPI, however).

Generally with ASYNCH, a one-to-one correspondence between the number of cores and processes is desired. More processes than cores means some cores must run more than one process, creating computational bottlenecks. More cores than processes means some cores will have no work to complete.

CHAPTER 4

Getting started

In this section, we will give the steps and data needed for running basic simulations. This section is intended for getting a new user up and running quickly, and to provide links in this documentation for more information when needed.

If you are not using the UoI HPC infrastructures, be sure to follow the steps in Section *Installation* for installing the ASYNCH solvers.

Tip: For Argon users. For those using Iowa HPC resources, downloading and compiling the source code is not necessary. But to be able to use the prebuilt executables, setting up your environment is necessary. The prefered way is to edit your `~/.bash_profile`:

```
# User specific environment and startup programs for Argon

export PATH=$PATH:$HOME/.local/bin:/Dedicated/IFC/.argon/bin

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Dedicated/IFC/.argon/lib

# Load module OpenMPI and HDF5
module load zlib/1.2.11_parallel_studio-2017.1
module load hdf5/1.8.18_parallel_studio-2017.1
module load openmpi/2.0.1_parallel_studio-2017.1
```

Tip: For Neon users. For those using Iowa HPC resources, downloading and compiling the source code is not necessary. But to be able to use the prebuilt executables, setting up your environment is necessary. The prefered way is to edit your `~/.bash_profile`:

```
# User specific environment and startup programs for Neon

export PATH=$PATH:$HOME/.local/bin:/Dedicated/IFC/.local/bin:/Dedicated/IFC/.neon/bin

# Load module OpenMPI and HDF5
module load openmpi/intel-composer_xe_2015.3.187-1.8.8
module load hdf5/1.8.17
```

The ASYNCH directory contains a folder called `examples`, which contains several data files for starting sample simulations, as well as sample outputs for comparision. The `examples` directory should be copied to a location where the user has write access (for example, the home directory). On Neon or Argon, this can be done with

```
cp -r /Dedicated/IFC/asynch/examples/ ~/
```

For the first example, we will produce output for a small basin with 11 links using a hydrological model with constant runoff. The global file to setup this simulation is `test.gbl`. This uses the model given in Section *Constant Runoff Hydrological Model*. If using your own machine, the simulation can be run with the command

```
mpirun -np 1 <bin path>/asynch test.gbl
```

As calculations are performed, you will see output produced to the terminal window. If using Neon or Argon (or any system using the Sun Grid Engine), the submit script test.sh can be used to run the simulation. Use the command

```
qsub test.sh
```

while in the examples directory to launch the job. Output from the program will be produced in a log file with a name like test run.o###### Try using the command

```
qstat -u <username>
```

to monitor the progress of your job.

Warning: A submit script is needed to run a job on multiple machines of Iowa HPC resources. If you attempt to run an ASYNCH simulation using just `mpirun` at a terminal window, you are probably running ASYNCH on a login node. Doing this limits the number of cores available to 12, slows down all other users's connections, and is an easy way to be reported to the HPC admins for misuse of resources!

When the program is complete, the output results are written to the folder `examples` The global file causes the production of three output files: `test.dat`, `test.pea`, and `test.rec`. These files should be identical to those found in `examples/results`. The `.dat` file contains the output hydrograph for links with link ids 3 and 80. The pea file contains the peakfow information for every link. The rec file contains the fnal value of every state of every link at the end of the simulation. For this simulation, all output files are small enough to view in a text editor.

The simulations performed will use only 1 MPI process. To increase this number, use, for example,

```
mpirun -np 2 <ASYNCH directory>/asynch test.gbl
```

or modify `test.sh` to use more processes. This can be done by modifying the environment

```
#$ -smp 1
```

to use 2 processes instead of 1. Also be sure to modify the last line with mpirun so MPI looks for 2 processes. When using more than 1 process, your results may difer slightly from those in `examples/results`. In fact, the results may vary slightly from simulation to simulation, even if nothing changed in the global file. This is a result from the asynchronous communication used by ASYNCH for MPI processes and is an expected behavior.

As a second example, try the same procedure as before using the global file `clearcreek.gbl`. If using an Iowa HPC resource, the submit script `clearcreek.sh` can be used. The model for this simulation is the toplayer hydrological model using the Clear Creek river basin See Section *Top Layer Hydrological Model*. Results for the output discharge and basefow are given in *Output from the toplayer model sample simulation.*. This basin is larger than in the previous simulation as it contains about 6,000 links. This is a good example to experiment with the number of processes used.

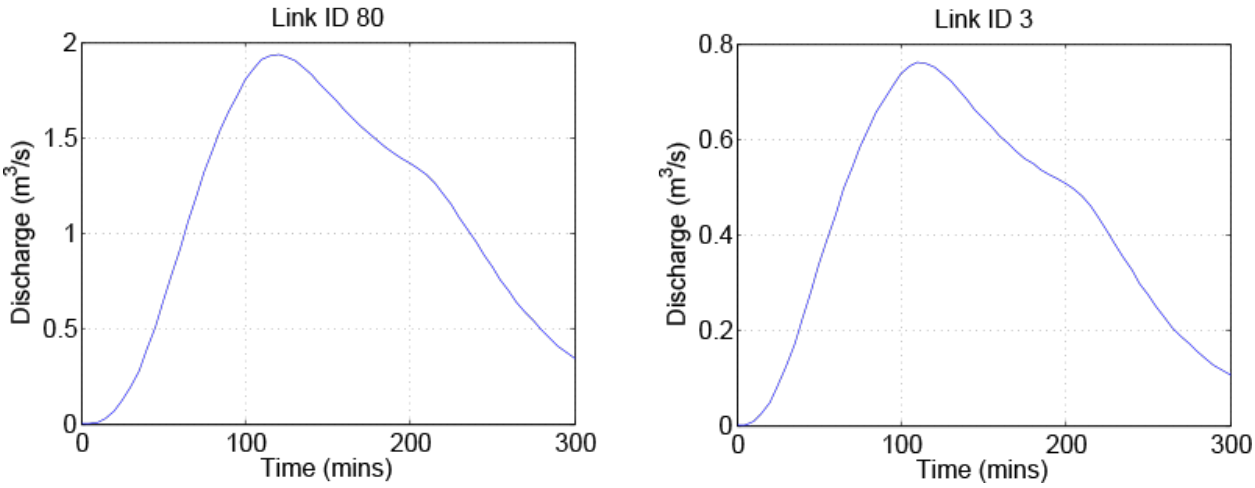


Fig. 1: Output from the sample simulation.

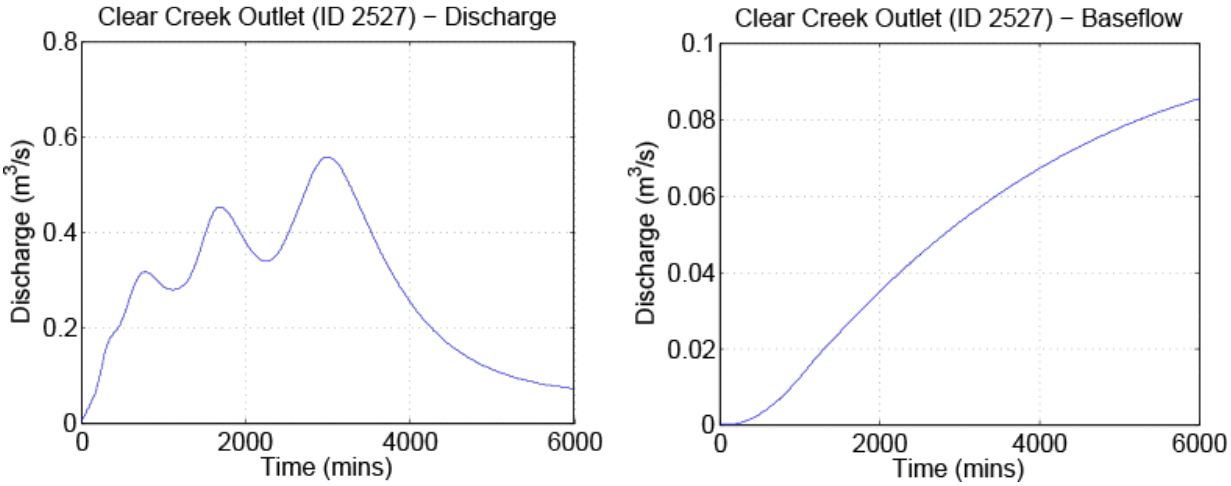


Fig. 2: Output from the toplayer model sample simulation.

CHAPTER 5

Input/Output Formats

Numerous inputs are required to construct and solve the model equations. This includes link connectivity, link parameters, numerical error tolerances, precipitation forcings, a mathematical model, etc. Most inputs are specified on a link-by-link basis, and can be given as either a file or a database table.

In addition, ASYNCH must know information about data outputs. This includes where to store the outputs, what the outputs are, at which links outputs will be given, etc.

The format for each input and output is described below. For database inputs/outputs, ASYNCH can access PostgreSQL through the libpq libraries. Obviously, the user must have read or write permissions to a PostgreSQL database to utilize database features.

5.1 Global File Structure

Global files (.gbl) are used to specify ALL inputs for the solvers. This includes river network topology files, database connections, initial model states, what information is printed to output files, model forcings, etc. Global files have a very rigid structure, unlike XML files, and information must be specified in a particular order. **The description of each input of the global files below are given in the order in which they are to be specified in the actual file**.

Global files are always ASCII files, assumed to be in UNIX format. The percent sign `%` is used for single line comments As described below, certain inputs are expected to be given within a single line Other than this restriction, white space is ignored Arguments surrounded by `{ }` below are mandatory, while those surrounded by the square brackets `[ ]` are dependent upon values specified by other parameters.

Although each setting in a global file modifies values for the ASYNCH solvers, their exact use can be modified by the user altering the underlying source code. This can be done with calls to routines described in *C API*.

Here is a typical global file taken from the examples folder:

```
%Model UID
190

%Begin and end date time
2017-01-01 00:00
```

(continues on next page)

(continued from previous page)

```
2017-01-02 00:00

%Parameters to filenames
0

%Components to print
3
Time
LinkID
State0

%Peakflow function
Classic

%Global parameters
%6 v_r  lambda_1  lambda_2  RC    v_h  v_g
6  0.33  0.20      -0.1     0.33  0.1  2.2917e-5

%No. steps stored at each link and
%Max no. steps transfered between procs
%Discontinuity buffer size
30 10 30

%Topology (0 = .rvr, 1 = database)
0 test.rvr

%DEM Parameters (0 = .prm, 1 = database)
0 test.prm

%Initial state (0 = .ini, 1 = .uini, 2 = .rec, 3 = .dbc, 4 = .h5)
1 test.uini

%Forcings (0 = none, 1 = .str, 2 = binary, 3 = database, 4 = .ustr,
%          5 = forecasting, 6 = .gz binary, 7 = recurring)
2

%Rain
1 test.str

%Evaporation
7 evap.mon
1398902400 1588291200

%Dam (0 = no dam, 1 = .dam, 2 = .qvs)
0

%Reservoir ids (0 = no reservoirs, 1 = .rsv, 2 = .dbc file)
0

%Where to put write hydrographs
%(0 = no output, 1 = .dat file, 2 = .csv file, 3 = database, 5 = .h5 packet, 6 = .h5␣
↪array)
5 5.0 outputs.h5

%Where to put peakflow data
%(0 = no output, 1 = .pea file, 2 = database)
1 test.pea
```

(continues on next page)

(continued from previous page)

```
%.sav files for hydrographs and peak file
%(0 = save no data, 1 = .sav file, 2 = .dbc file, 3 = all links)
1 test.sav %Hydrographs
3 %Peakflows

%Snapshot information (0 = none, 1 = .rec, 2 = database, 3 = .h5, 4 = recurrent .h5)
4 60 test.h5

%Filename for scratch work
tmp

%Numerical solver settings follow

%facmin, facmax, fac
.1 10.0 .9

%Solver flag (0 = data below, 1 = .rkd)
0
%Numerical solver index (0-3 explicit, 4 implicit)
2
%Error tolerances (abs, rel, abs dense, rel dense)
1e-3 1e-3 1e-3
1e-6 1e-6 1e-6
1e-3 1e-3 1e-3
1e-6 1e-6 1e-6

# %End of file
--------------------------------
```

In the following sections, we will go into details about the meaning and options for each entry.

5.1.1 Model Type

Format:

```
{model id}
```

This value specifies the id for the model to be used. This is a non-negative integer value which corresponds to a particular system of ordinary-differential equations (or possibly DAEs). Examples of built-in models is given in *Built-in Models*. If you are using the API to use custom models, this model id is ignored, see *Custom Models*.

5.1.2 Simulation period

Format:

```
{begin datetime}
{end datetime}
```

The begin and end datetimes are given in `YYYY-MM-DD HH:MM` format using the UTC timezone or in unix_time format.

5.1.3 Parameters on Filenames

Format:

```
{parameter on output filename flag}
```

This is a boolean value (`0` or `1`) that indicates whether all output filenames should be postfixed with the uniform in space and time (global) parameters. `0` indicates no, `1` indicates yes. This feature can be useful for keeping track of output files from multiple simulations.

5.1.4 Solver Outputs

Format:

```
{number of outputs}
[output1]
[output2]
[...]
```

This set of input parameters specifies the names of all outputs from the solvers. Several built in outputs exist, and the user is able to construct his own outputs. Built in outputs are given in *Built-In Output Time Series*. Output names are case sensitive. The first required value is the number of outputs (>= 0), followed by the names of each output, on separate lines.

5.1.5 Peakflow Statistics Function Name

Format:

```
{function name}
```

This sets the function for outputting peakflow information. The built in peakflow function "Classic" is one option, and the user is free to construct his own. A function name must be specified here, even if peakflow information is not requested for any links.

5.1.6 Global Parameters

Format:

```
{number of parameters} [parameter 1] [parameter 2] ... [parameter n]
```

This is where model parameters which are constant in space and time are specified. The first value is a nonnegative integer specifying the number of global parameters to follow. Every model requires a certain number of global parameters. If the number given in the global file is less than expected for a particular model, an error occurs. If the number is greater than expected, a warning is given. These "extra" parameters are available to the model for use. This can sometimes be useful for quick tests, but should be avoided normally.

The parameter meanings depend upon the model used. The units of these parameters is also model dependent.

5.1.7 Buffer Sizes

Format:

```
{steps stored at each link} {max number of steps transferred} {discontinuity buffer␣
↪size}
```

These nonnegative integer values allow the user to specify sizes of internal buffers. In general, as these numbers are increased, the solvers run faster, but more memory is required. A good starting point that works in most cases is the set `30 10 30`. Typically, if these values need to be less than 10 to run the solvers, a deeper issue with memory constraints should be addressed.

5.1.8 Topology

Format:

```
{topology flag} [output link id] {.rvr filename or .dbc filename}
```

This is where connectivity of the river network is specified. This can be done in one of two ways If the topology flag is `0`, a river topology file (.rvr) is used. If the topology flag is `1`, then topology is downloaded from the database specified with the database file (.dbc). The database connection allows for one additional feature: a subbasin can be specified If the output link id is taken to be 0, all link ids found in the database are used. Otherwise, the link with link id specified and all upstream links are used. Pulling subbasins from a topology file is not currently supported.

5.1.9 Link Parameters

Format:

```
{parameter flag} {.prm filename or .dbc filename}
```

This specifies where parameters which vary by link and not time, are specified If the parameter flag is `0`, the parameters are given in a parameter (.prm) file. If the flag is `1`, then the parameters are downloaded from the database specified by the database connection file (.dbc). The number, order, meaning, and units of these parameters varies from model to model.

5.1.10 Initial States

Format:

```
{initial state flag} {.ini, .uini, .rec, .dbc or .h5 filename} [unix time]
```

This section specifies the initial state of the model. The values for the initial state flag can be `0`, `1`, `2`, `3` or `4` corresponding, respectively, to a ini, uini, rec, dbc, h5 file. The unix time argument is used for database connections only. This value is available in the query of the database connection file and can be used for selecting values from tables.

5.1.11 Forcings

Format:

```
{number of forcings}
[forcing1 flag] [forcing1 information]
[forcing2 flag] [forcing2 information]
[...]
```

Information about time dependent forcings is specified here. Each model has an expected number of forcings. If the number of forcings specified here is less than expected, an error is thrown. If the number of forcings is greater than expected, a warning is given. This warning allows for tests to be performed and implemented quickly. In general, this feature should be avoided.

Forcing information varies considerably based upon the corresponding forcing flag. Several forcing types require unix times to determine what forcing data to use. If a model requires multiple forcings with unix times, the times do not need to be consistent, i.e., one forcing could start on July 1st 2014 at midnight, while another forcing starts at April 5th 2008.

No Forcing

Format:

```
0
```

A forcing flag of `0` specifies no forcing input. This is the same as a forcing value of *0.0* for all links and all time.

Storm File

Format:

```
1 {.str filename}
```

A forcing flag of `1` indicates the forcing is specified by a .str file. The filename and path of a valid storm (.str) file is required.

Binary Files

Format:

```
2 {binary file identifier}
{chunk size} {time resolution} {beginning file index} {ending file index}
```

A forcing flag of `2` indicates the forcing is specified by a collection of binary forcing files. The identifier can be adorned with a path to the binary files. The chunk size is a positive integer that indicates the number of binary files kept in memory at once. The time resolution indicates the amount of time between successively indexed binary files. This value is a floating point number with units equal to those of the time variable of the model used The beginning and ending file indices indicate the range of the binary files to be used. The indices are integer valued. The simulation will begin using the binary file with index given by the beginning file index. If the total simulation time would require binary files with index greater than the ending file index, the forcing values are taken to be 0.0 for all such binary files.

Forcings from Databases

Format:

```
3 {.dbc filename}
{chunk size} {time resolution} {beginning unix time} {ending unix time}
```

A forcing flag of `3` indicates the forcing data will be pulled from a PostgreSQL database. The database connection filename can include a path. The chunk size is a positive integer representing the number of forcing values pulled from the database at once from each link. A chunk size of 10 tends to work well. A larger chunk size requires more memory and larger datasets returned from the database, but a small number of queries. The time resolution is a floating point

number with units in minutes. This represents the time resolution of the data in the accessed table. The integrity of the database table is not thoroughly checked by the solvers.

The simulation will begin using the data from the database with unix time given by the beginning unix time. If the total simulation time would require data from the database with unix time greater than the ending unix time, the forcing values are taken to be 0.0 for times greater than the ending unix time.

Uniform Forcings

Format:

```
4 {.ustr filename}
```

A forcing flag of `4` indicates a forcing that is uniform in space. The forcings are given by a uniform storm file (.ustr).

Irregular Binary Files

Format:

```
5 {binary file prefix identifier}
{chunk size} {time resolution in minutes} {beginning unix time} {ending unix time}
```

A forcing of flag `5` indicates a collection of irregular binaries forcing files. The files are expected to be named ending by the *unix timestamp* of the time when they start to take affect, without any file extension. The *identifier* can be adorned with a relative or and absolute path to the binary files. The *chunck size* indicates the maximum number of files that will be read by Asynch on every moment the program stops to read the input files. As it happens with the database forcing (number `3`), a *chunck size* of 10 tends to work well. The *time resolution* indicates the amount of time, in minutes, between successively irregular binary files. The beginning and ending *unix times* indicate the limits, in the simulation time, of the forcing influence.

More information about the internal structure of those irregular binary files *Irregular Binary Storm Files*.

GZipped Binary

Format:

```
6 {gzipped binary file identifier}
{chunk size} {time resolution} {beginning file index} {ending file index}
```

A forcing flag of `6` indicates the forcing is specified by a collection of binary forcing files that have been gzipped (compressed as .gz files). All parameters for this flag are identical to that of using binary files with forcing flag `3`.

Monthly Forcings

Format:

```
7 { mon filename}
{beginning unix time} {ending unix time}
```

A forcing flag of `7` indicates a uniform in space forcing that recurs monthly. When the end of the calendar year is reached, the monthly forcing file (.mon) is read again from the beginning The beginning unix time is used to determine the month the simulation begins (for this forcing). If the total simulation time takes the simulation past the ending unix time, the forcing is assumed to be `0.0` for all locations and times beyond the ending unix time

Grid Cell

Format:

```
8 {index filename}
{chunk size} {beginning file index} {ending file index}
```

A forcing flag of `8` indicates the forcing is specified by a collection of grid cell forcing files. The index filename can be adorned with a path to the index file. The chunk size is a positive integer that indicates the number of grid cell files kept in memory at once. The beginning and ending file indices indicate the range of the grid cell files to be used. The indices are integer valued.

The simulation will begin using the grid cell file with index given by the beginning file index. If the total simulation time would require grid cell files with index greater than the ending file index, the forcing values are taken to be `0.0` for all such grid cell files. In addition, if a grid cell file is missing, all values at each cell are assumed to be `0.0`.

5.1.12 Dams

Format:

```
{dam flag} [.dam or .qvs filename]
```

This section specifies whether dams will be used. A dam flag of `0` means no dams are used. A flag of `1` indicates a dam file (.dam) will be used, and a flag value of `2` indicates a discharge vs storage file (.qvs) will be used. Some models do not support dams. For these models, the dam flag must be set to `0` or an error occurs.

5.1.13 State Forcing Feeds

Format:

```
{reservoir flag} [.rsv or .dbc filename] [forcing index]
```

This section specifies whether a provided forcing (see *Forcings*) is to be used as a forcing of the states of differential or algebraic equations at some links. A reservoir flag of `0` indicates no forcing will by applied to system states. A flag of `1` indicates state forcings will be applied to all link ids in the specified .rsv file. A reservoir flag of `2` indicates state forcing will be applied to all link ids pulled from the database the given .dbc file. If the reservoir flag is not `0`, then the index of the forcing must be specified.

5.1.14 Time Series Location

Format:

```
{time series flag} [time resolution] [.dat or .csv or .h5 or .dbc filename] [table␣
↪name]
```

This section specifies where the final output time series will be saved. A time series flag value of `0` indicates no time series data will be produced. Any flag with value greater than `0` requires a time resolution for the data. This value has units equal to the units of total simulation time (typically minutes). A value of `-1` uses a resolution which varies from link to link based upon the expression:

$$\left(0.1 \cdot \frac{A}{1\ km^2}\right)^{\frac{1}{2}}\ min \tag{5.1}$$

where A is the upstream of the link, measured in km2.

A time series flag of `1` indicates the results of the simulation will be saved as a .dat file. The filename complete with a path must be specified. If a file with the name and path given already exists, it is overwritten. A time series flag of `2` indicates the results will be stored as a .csv file. A time series flag of `3` indicates the results will be uploaded into the database described by the given .dbc file. In this case, a table name accessible by the queries in the .dbc file must be specified. A time series flag of `5` indicates the results will be stored as a .h5 HDF5 file with a packet layout compatible with PyTable. A time series flag of `6` indicates the results will be stored as a .h5 HDF5 file with an 3D array layout. Time, link id and output indexes are given as additional 1D "dimension" arrays. Selected outputs in *Solver Outputs* must have the same type (ASYNCH_FLOAT).

This section is independent of the section for Link IDs to Save described below (see *Global Parameters*) For example, if link ids are specified in the Link IDs to Save section and the time series flag in the Time Series Locations set to `0`, no output is generated. Similarly, if *the time series id flag* is set to `0` in the Link IDs to Save section and the time series flag is set to `1`, a .dat file with `0` time series is produced.

Note: The time resolution is entirely independent of the time step used by the numerical integrators. Reducing this value does NOT produce more accurate results. To improve accuracy, reduce the error tolerances described in *Numerical Error Tolerances*. There is no built-in way to produce results at every time step, as this is a very easy way to crash a compute node or file system.

5.1.15 Peakflow Data Location

Format:

```
{peakflow flag} [.pea / .dbc filename] [table name]
```

This section specifies where the final peakflow output will be saved. A peakflow flag of `0` indicates no peakflow data is produced. A peakflow flag of `1` indicates the peakflow results of the simulation will be saved as a .pea file. The filename complete with a path from the binary file must be specified. A peakflow flag of `2` indicates the results will be uploaded into the database described by the given .dbc file. In this case, a table name accessible by the queries in the dbc file must be specified.

This section is independent of the section for Link IDs to Save described below (see *Link IDs to Save*). For example, if link ids are specified in the Link IDs to Save section and the peakflow flag in the peakflow data location is set to `0`, no output is generated. Similarly, if the peakflow id flag is set to `0` in the Link IDs to Save section and the peakflow flag is set to `1`, a .pea file with `0` peakflows is produced.

5.1.16 Link IDs to Save

Format:

```
{time series id flag} [.sav / .dbc filename]
{peakflow id flag} [.sav / .dbc filename]
```

This section provides the list of link ids in which data is produced. The first line is for the time series outputs, while the second is for the peakflow outputs. The time series ID flag and the peakflow ID flag take the same list of possible values. A flag of `0` indicates no link IDs for which to produce data. A flag of `1` indicates the list of link IDs is provided by the corresponding save file (.sav). A flag of `2` indicates the list of link IDs is provided by the database specified in the given database connection file (.dbc). A flag of `3` indicates that all links will have data outputted.

Warning: A time series ID flag of `3` can easily wreak havoc on a file system for simulations with a large number of links. At the very least, extremely large output files and database tables will occur. Be very careful with this!

Typically, using a flag value of 3 for peakflow link ids, or for the time series ID flag for a very small basin (< 500 links) will not create any problems.

This section is independent of the sections for Time Series Location and peakflow data location above (see *Time Series Location* and *Peakflow Data Location*). For example, if link ids are specified in the Link IDs to Save section and the time series flag in the Time Series Location set to 0, no output is generated. Similarly, if the time series id flag is set to 0 in the Link IDs to Save section and the time series flag is set to 1, a .dat file with zero time series is produced.

5.1.17 Snapshot Information

Format:

```
{snapshot flag} [time step of periodical snapshots] [.rec / .dbc / .h5 filename]
```

This section specifies where snapshot information is produced. A snapshot is a record of *every state* at *every link* in the network. Snapshots can be produced at the end of simulations or periodically. This is useful for beginning a new simulation where an old one ended. A snapshot flag of 0 indicates no snapshot is produced. A snapshot flag of 1 indicates the snapshot will be produced as a recovery (.rec) file with path and filename specified. A snapshot flag of 2 indicates the snapshot will be uploaded to the database specified by the database connectivity (.dbc) file.

A snapshot flag of 3 indicates the snapshot will be produced as a HDF5 (.h5) file with path and filename specified. A snapshot flag of 4 generates periodical snapshots in which case an addition parameter gives the interval between two snapshots and the second parameter is the output basename. For example:

```
%Snapshot information (0 = none, 1 = .rec, 2 = .dbc, 3 = .h5, 4 = periodical .h5)
4 60 filename.h5
```

generates

```
filename_1480000000.h5
filename_1480003600.h5
filename_1480007200.h5
...
```

5.1.18 Scratch Work Location

Format:

```
{filename}
```

This section specifies the location of temporary files. These files are used to store intermediate calculations. The filename can include a path name. If the file already exists, the contents are overwritten. If a simulation is aborted, these files may not be removed. Otherwise, they are deleted at the end of the simulation.

5.1.19 Error Control Parameters

Format:

```
{facmin} {facmax} {fac}
```

This section specifies parameters related to the error control strategy of the numerical integrators. The value facmin represents the largest allowed decrease in the stepsize of the integrators as a percent of the current step Similarly, facmax represents the largest allowed increase. The value fac represents the safety factor of the integrators. Any accepted stepsize is multiplied by this value Good values of facmin, facmax, and fac to use are 0 1, 10 0, and 0 9, respectively

5.1.20 Numerical Error Tolerances

Format:

```
{solver flag} [ rkd filename]
[rk solver index]
[absolute error tolerance 1] [absolute error tolerance 2]
[relative error tolerance 1] [relative error tolerance 2]
[dense absolute error tolerance 1] [dense absolute error tolerance 2]
[dense relative error tolerance 1] [dense relative error tolerance 2]
```

This section specifies error tolerances for the numerical integrators. A solver flag of 0 indicates the same tolerances will be used for all links. A solver flag of 1 indicates the tolerance info will be specified in the given RK data (.rkd) file. If solver flag is 0, than an rk solver index must be specified. A list of Runge-Kutta methods is given in *Built-In Runge-Kutta Methods*. Each error tolerance must have a value for each state of the system. The order of the tolerances must match the order of the states in the state vectors. The absolute and relative error tolerances are those typically used for RK methods. The dense tolerances are for the numerical solution produced between time steps. A numerical solution is rejected if either the error tolerances or dense error tolerances for any state is believed to be violated.

5.2 Database Connection Files

Database connection files are ASCII text files with a .dbc extension which specify how to connect to a database, and the queries to pull/push data from/to the database. Although the format of database connection files is the same, the specific requirements of the queries varies with how the file is used. For instance, queries for pulling link connectivity information is very different from queries for uploading peakflows to a database table.

Format:

```
dbname={db} host={host} user={user} password={pass}
{number of queries}
[query 1]
[query 2]
...
```

The first line of every database connection file specifies the information needed to make a connection. A user must have permission to read or write from the database at the given host; otherwise, queries sent to the database will fail. The number of queries will vary depending upon how the database connection file is used. The appropriate number of queries and what they should return is documented in the remainder of *Input/Output Formats*. The number of queries may be zero.

Any queries listed **MUST** be ended with a semicolon (;). For some queries, further information outside the database connection file may be available, depending upon how the query is to be used. This additional information is explained in the appropriate section below for input formats. Such information includes link ids and unix times. To denote in a query where this information should be placed, use the symbol "%u" for integers and "%s" for names.

5.3 Link Connectivity Input

Link connectivity information is used to specify how links in a network are connected. Topology can be provided through either a river network file (.rvr) file or through a database table. When a river network file is used, every link in the file is used (i.e. no subnetworks) then pulling connectivity data from a database, a subset of the network can be used.

Regardless of the format, all link ids must be given in the same order in the link connectivity, link parameter, and initial state inputs.

5.3.1 Rvr Files

River network files are ASCII text files with the following format:

```
{number of links}
{link id 1}
{number of parents} [parent id 1] [parent id 2]
{link id 2}
{number of parents} [parent id 1] [parent id 2]
```

White space can be used freely throughout the file. The layout in the above specification is purely optional; the order of the information is what is important. The file begins with the total number of links in the file. Then each link id is specified, followed by the number of parents for the link and each of their ids. A link id can appear in a list of parent link ids at most once. If a link does not have parents, it must still appear in this file with a `0` for the number of parents.

5.3.2 Topology Database Queries

If the connectivity is pulled from a database, a corresponding database connection file is used This file requires three queries:

1. Query to pull all link ids from a table
   - Inputs: none
   - Returned tuples: (link id)
2. Query to pull all link id, parent link id pairs
   - Inputs: none
   - Returned tuples: (link id, parent link id)
3. Query to pull all link id, parent link id pairs upstream from a given outlet link id
   - Inputs: outlet link id
   - Returned tuples: (link id, parent link id)

The last two queries must return a tuple for each link id for each parent link. So a link with two parents should appear twice in the returned tuples, once for each parent link. The returned tuples must be grouped by the link id so all parent information appears consecutively.

5.4 Link Parameter Input

Link parameter input specifies the parameters for the model that vary link to link This information can be provided in a parameter file (prm) or through a database table The number of parameters for each link, their meaning, and

their order depends upon the model used In particular, the value of disk params determines the number of parameters expected at each link See *SetParamSizes*.

Regardless of the format, all link ids must be given in the same order in the link connectivity, link parameter, and initial state inputs.

5.4.1 Prm File

A parameter file is an ASCII text file with the following format:

```
{number of links}
{link id 1} {parameter 1} {parameter 2} {parameter 3}
{link id 2} {parameter 1} {parameter 2} {parameter 3}
```

White space can be used freely throughout the file. The layout in the above specification is purely optional; the order of the information is what is important. The file begins with the total number of links. Then each link id is specified, followed by the parameters for that link.

5.4.2 Param Database Queries

If the parameters are pulled from a database, a corresponding database connection file is used This file requires two queries:

1. Query to pull all parameters
   - Inputs: none
   - Returned tuples: (link id, parameter 1, parameter 2, . . .)
2. Query to pull all parameters above an outlet
   - Inputs: outlet link id
   - Returned tuples: (link id, parameter 1, parameter 2, . . .)

5.5 Initial Values Input

The link initial values input specifies the initial values for the states of the differential and algebraic model equations This information can be provided in several different formats: an initial value file (.ini), a uniform initial value file (.uini), a recovery file (.rec), and through a database table.

5.5.1 Ini Files

An initial value file is an ASCII text file that lists the initial values for each link. The format is:

```
{model type}
{number of links}
{initial time}
{link id 1}
{initial value 1} {initial value 2}
{link id 2}
{initial value 1} {initial value 2}
```

The model type is the number of the model to be used. This determines how many initial values are expected for the model. Initial states must be provided only for those states determined by differential equations, and only for those which require an initial condition. These are the states with index between `diff_start` and `no_ini_start` in the state vectors See *SetParamSizes*.

5.5.2 Uini Files

A uniform initial value file is similar to an initial value file, but the initial values, when required, are the same at every link The format is given by:

```
{model type}
{initial time}
{initial value 1} {initial value 2}
```

The model type is the number of the model to be used. This determines how many initial values are expected for the model. Initial values must be provided only for those states determined by differential equations, and only for those which require an initial condition. These are the states with index between `diff_start` and `no_ini_start` in the state vectors. See *SetParamSizes*. Notice that unlike an initial value file, no link ids are given, and only one set of initial values are given.

5.5.3 Rec Files

A recovery file is an ASCII text file that lists the initial values for each link. The format is:

```
{model type}
{number of links}
{initial time}
{link id 1}
{initial value 1} {initial value 2}
{link id 2}
{initial value 1} {initial value 2}
```

The format is identical to that of an initial value file, with one important exception The initial value of EVERY state must be provided at each link. For models with `diff_start` set to 0 and `no_ini_start` set to dim, a recovery file is identical to an initial value file. See *SetParamSizes*.

Warning: For the initial values of algebraic equations, no checks on the input data are performed to ensure the solution is consistent.

5.5.4 Ini Database Queries

If the initial values are pulled from a database, a corresponding database connection file is used. This file requires one query:

1. Query to pull all initial states for every link:

- Inputs: integer value
- Returned tuples: (link id, initial value 1, initial value 2,)

The query allows for one input to be used to obtain the needed information. This value could be, for example, an outlet link id or a unix time. Similar to recovery files, initial values must be provided for every link.

5.5.5 Ini HDF5 Files

H5 Ini files are H5 that contains a single resizable Packet Tables or PyTable *snapshot*. Two HDF5 tools can be used to get the structure of the snapshots, `l5hs` and `h5dump`:

```
>h5ls -v test_1483228800.h5
snapshot                 Dataset {11/Inf}
    Location:  1:1024
    Links:     1
    Chunks:    {512} 14336 bytes
    Storage:   308 logical bytes, 80 allocated bytes, 385.00% utilization
    Filter-0:  deflate-1 OPT {5}
    Type:      struct {
                   "link_id"          +0    native unsigned int
                   "state_0"          +4    native double
                   "state_1"          +12   native double
                   "state_2"          +20   native double
               } 28 bytes
```

```
>h5dump -H test_1483228800.h5
HDF5 "test_1483228800.h5" {
GROUP "/" {
   ATTRIBUTE "model" {
      DATATYPE  H5T_STD_U16LE
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
   ATTRIBUTE "unix_time" {
      DATATYPE  H5T_STD_U32LE
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
   ATTRIBUTE "version" {
      DATATYPE  H5T_STRING {
         STRSIZE 4;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
      DATASPACE  SCALAR
   }
   DATASET "snapshot" {
      DATATYPE  H5T_COMPOUND {
         H5T_STD_U32LE "link_id";
         H5T_IEEE_F64LE "state_0";
         H5T_IEEE_F64LE "state_1";
         H5T_IEEE_F64LE "state_2";
      }
      DATASPACE  SIMPLE { ( 11 ) / ( H5S_UNLIMITED ) }
   }
}
}
```

Three global attributes are available :

Name	Description
version	The version of ASYNCH used to generate this file
model	The model id used to generate this file
issue_time	The unix time at the beginning of the time serie

5.6 Forcing Inputs

Numerous and diverse formats are implemented for feeding forcing inputs into a model. These formats vary considerably, and can have different impacts on performance.

5.6.1 Storm Files

Storm files (.str) provide an ASCII text format for setting a forcing at each link. The format of these files is:

```
{number of links}
{link id 1} {number of changes}
{time 1} {value 1}
{time 2} {value 2}
{time 3} {value 3}
{link id 2} {number of changes}
{time 1} {value 1}
{time 2} {value 2}
{time 3} {value 3}
```

The format requires a time series to be provided for every link. The number of values can vary from link to link, and the time steps do not need to be uniformly spaced or even the same for each link. The first time at each link must be the same however, and correspond to the beginning of the simulation (typically 0). The forcings are assumed to be constant between time steps. After the final time step, the forcing value is held at the last value for the remainder of the simulation. The data provided by a storm file is entirely read into memory at the beginning of a run. As such, this format may not be suitable for large or long simulations.

5.6.2 Uniform Storm Files

Uniform storm files (.ustr) provide an ASCII text format for setting a forcing uniformly at each link. The format of these files is:

```
{number of changes}
{time 1} {value 1}
{time 2} {value 2}
{time 3} {value 3}
```

The format is similar to that of a storm file, but only one time series is specified, and is applied at every link. The time steps do not need to be uniformly spaced. The first time must correspond to the beginning of the simulation (typically 0) The forcing is assumed to be constant between time steps. After the fnal time step, the forcing value is held at the last value for the remainder of the simulation. The data provided by a uniform storm file is entirely read into memory at the beginning of a run. As such, this format may not be suitable for extremely long simulations.

5.6.3 Binary Storm Files

Binary storm files (no extension) may also be used. Instead of a single file, these are a collection of files providing forcing values at different times. The format of these files is:

```
{link id 1 value}
{link id 2 value}
{link id 3 value}
```

Each file is simply a list of forcing values for each link. Because link ids are not present, the values are assumed to be in the same order as the link ids from the link connectivity input. Further, a value must be specified for every link. The

filename of each binary file should end with an integer value. The index of each file should be numbered consecutively. The time step size between files, as well as the range of files to use, are specified in the global file (see 6 1 10). If the simulation goes beyond the last file, all further forcing values are assumed to be `0` for all links. The values in binary storm files are stored as single precision floating point numbers, with endianness different from the native machine. These files are read into memory chunk by chunk. This allows for a ceiling on the memory used, independent of the number of files.

5.6.4 Irregular Binary Storm Files

Irregular binary storm files are designed to be a more flexible and compact file format than the simple binary storm files. Also a collection of files providing forcings at different times, they are named having the respective unix time as a sufix and no file extensions. Each file may contain a variable number of links and should present the following structure:

```
{number of links}
{link id 1}
{value 1}
{link id 2}
{value 2}
```

The number of links and the link ids are in *unsigned integers*, while values in *floats*. Each of these values is written in 4 bytes. Asynch does not try to distinguish between little endian or big endian, but little endian is recomended as it has already been tested on both Windows desktop and Linux-based server environments.

This forcing structure is named *irregular* because forcing values of *zero* for a link do not need to be represented withing the file, causing the foncings that are naturally sparse present irregular, minimal file sizes. Also, in the absence of a single irregular binary file in the sequence, it is assumed a forcing value of *zero* for all links for the missing time period.

For more information about referencing the irregular binary files in the global files, see *Irregular Binary Files*.

5.6.5 Gzipped Binary Storm Files

Gzipped binary storm files (.gz) are also supported. The format of these files is identical to that of the binary storm files, as they are simply gzipped binary storm files. The files are uncompressed before use, making this format slower than the regular binary storm files.

5.6.6 Grid Cell Files

Grid cell files group link ids together, and specify forcing values for each group (referred to as a cell). Although similar to binary files, this format requires two accompanying text files: an index file and a lookup file.

The index file specifies meta data for the grid cell data. The format for this ASCII file is:

```
{time resolution (mins)}
{conversion factor}
{number of cells}
{grid cell data file name prefix}
{lookup filename}
```

The time resolution specifies the amount of time between grid cell files. The resolution is typically given in minutes. The conversion factor is a floating point number. Each forcing value in the grid cell files is multiplied by this factor. The number of cells is an integer value. Each grid cell filename has a prefix, followed by a number. The prefix is specified in the index file. The prefix may include a path. If the path is relative (i e , does not begin with a '/'), the path

is taken relative to the location of the index file. Lastly, the index file includes the filename for the lookup file. A path is allowed, but is taken relative to the location of the index file, unless given as an absolute path.

The lookup file specifies which link ids are located in each cell. The format for this text file is:

```
{link id 1} {cell id 1}
{link id 2} {cell id 2}
```

The cell ids are indexed starting from 0, and the cell index cannot be larger than the number of cells specified in the accompanying index file.

The grid cell files are binary files. Each gives forcing values for each cell at a moment of time. If a cell is omitted from a file, then the forcing value is assumed to be 0. The filename for each grid cell file begins with the same prefix, followed by an integer. This integer indicated the order in which the grid cell files should be used. Although the number on these files should be consecutive, a missing file indicates all cells take a forcing value of 0. The format of the binary grid cell files is:

```
{1} {cell id 1} {forcing value 1}
{cell id 2} {forcing value 2}
```

The grid cell files are binary, so the spacing above is purely for readability. Each file begins with the integer value 1, stored as a 32-bit integer. This is used for checking the file format. Each cell id is given as a 32-bit integer and each forcing value is given as a 16-bit integer. Before the forcing values are loaded into memory, they are multiplied by the conversion factor given in the index file. Again, every cell does not need to be given in every grid cell file; only when the forcing value is nonzero does a value need to be given.

5.6.7 Monthly Recurring Forcing

Monthly recurring forcing files (.mon) allow forcings to be set monthly. These files are given as ASCII text files in the format:

```
{value for January}
{value for February}
{value for March}
...
{value for December}
```

A value must be given for each month. Over each simulated month, the forcing value is held constant, and is uniform over all links.

5.6.8 Database Forcing

If the forcing data is pulled from a database, a corresponding database connection file is used. This file requires three queries:

1. Query to pull rainfall data for all link ids present in the database table
   - Inputs: lower unix time, upper unix time
   - Returned tuples: (unix time, forcing value, link id)
2. Query to pull rainfall data for all link ids upstream from an outlet link
   - Inputs: outlet link id, lower unix time, upper unix time
   - Returned tuples: (unix time, forcing value, link id)
3. Query to pull a single forcing intensity from the database table

- Inputs: none
- Returned tuple: (unix time)

The first and second queries are similar, except that the second pulls only a subset of links from the database table. Forcing values are read in blocks from the table, with forcing values coming between the lower (inclusive) and upper (exclusive) unix times available to the queries. If a value for a link is not pulled from the database, the value is assumed to be 0.

The last query is used to find an actual valid timestamp in the database table. It needs to return only one unix time.

5.7 Dam Parameters Input

Two formats currently exist for setting parameters at links with dams: dam parameter files (.dam) and discharge vs storage files (.qvs).

5.7.1 Dam Files

The format of dam parameter files is similar to that of parameter files:

```
{number of links with dams}
{link id 1}
{parameter 1} {parameter 2} {parameter 3} ...
{link id 2}
{parameter 1} {parameter 2} {parameter 3} ...
...
```

The number of parameters needed for each link is model dependent and determined by the value dam params size. See *SetParamSizes*. For dam parameter files, only the links with dams must be listed here. Only links with id appearing in this file will have dams.

5.7.2 QVS Files

Discharge vs storage files take a series of discharge values and a corresponding series of storage values to decide the relationship between two states. The format of these files is similar to storm files (see *Forcing Inputs*):

```
{number of links with dams}
{link id 1} {number of points for link 1}
{storage value 1} {discharge value 1}
{storage value 2} {discharge value 2}
...
{link id 2} {number of points for link 2}
{storage value 1} {discharge value 1}
{storage value 2} {discharge value 2}
...
```

The number of points at each link can vary For dam parameter files, only links with dams are listed here. Only links with id appearing in this file will have dams. Internally, discharges and storages with values between two points are interpolated. This interpolation process is model dependent.

5.8 Time Series Output

Three formats are supported for outputting time series calculations: data files (.dat), comma-separated values (.csv), and a database table. The particular time series calculated is set in the global file (see *Time Series Location*). The structure of each format is considerably different.

5.8.1 Data Files

Data files are in ASCII text format. These files are designed to be generic and flexible so as to be easily read by whatever data analysis program the user prefers. Data files are created with the format:

```
{number of links}
{number of output values}
{link id 1} {number of points for link id 1}
{value 1 for series 1} {value 1 for series 2} {value 1 for series 3} ...
{value 2 for series 1} {value 2 for series 2} {value 2 for series 3} ...
...
{link id 2} {number of points for link id 2}
{value 1 for series 1} {value 1 for series 2} {value 1 for series 3}
{value 2 for series 1} {value 2 for series 2} {value 2 for series 3}
...
```

The series for the links appear in a column The number of points can vary from link to link, depending upon the user's selection in the global file The number of output values determines how many values appear in each line of the time series.

5.8.2 CSV Files

A CSV file is a typical format to make data easy to read in spreadsheet software. The structure of CSV files is:

```
{link id 1},, ... , {link id 2},
Output 1, Output 2,  , Output 1, Output 2,
{value 1,1,1}, {value 1,2,1},  , {value 1,1,2}, {value 1,2,2},
{value 2,1,1}, {value 2,2,1},  , {value 2,1,2}, {value 2,2,2},
{value 3,1,1}, {value 3,2,1},  , {value 3,1,2}, {value 3,2,2},
```

The series for the links appear in a row. Under link id 1, each requested series appears, followed by the series for link id 2, and so on.

5.8.3 Out Database Queries

A database connection file can be used to upload results into a database table This file requires only one query:

1. Query to create a table for uploading data

- Inputs: table name
- Returned tuples: none

The query should create the table where the series information is to be stored ASYNCH does NOT remove any existing data from the table, or check if the table exists already.

5.8.4 Out HDF5 Files

H5 outputs files are the prefered output format as it is both compact and efficient. There are also easy to read with third party software, see *Reading the HDF5 outputs with Python* for example.

H5 Output files are H5 that contains a single resizable Packet Tables or PyTable *outputs*. Two HDF5 tools can be used to get the strucutre of the outputs, `l5hs` and `h5dump`:

```
> l5hs -v outputs.h5
Opened "outputs.h5" with sec2 driver.
outputs                  Dataset {578/Inf}
    Location:  1:1024
    Links:     1
    Chunks:    {512} 8192 bytes
    Storage:   9248 logical bytes, 3455 allocated bytes, 267.67% utilization
    Filter-0:  deflate-1 OPT {5}
    Type:      struct {
                   "Time"             +0    native double
                   "LinkID"           +8    native int
                   "State0"           +12   native float
               } 16 bytes
```

```
>h5dump -H  outputs.h5
HDF5 "outputs.h5" {
GROUP "/" {
   ATTRIBUTE "issue_time" {
      DATATYPE  H5T_STD_U32LE
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
   ATTRIBUTE "model" {
      DATATYPE  H5T_STD_U16LE
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
   ATTRIBUTE "version" {
      DATATYPE  H5T_STRING {
         STRSIZE 4;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
      DATASPACE  SCALAR
   }
   DATASET "outputs" {
      DATATYPE  H5T_COMPOUND {
         H5T_IEEE_F64LE "Time";
         H5T_STD_I32LE "LinkID";
         H5T_IEEE_F32LE "State0";
      }
      DATASPACE  SIMPLE { ( 578 ) / ( H5S_UNLIMITED ) }
   }
}
}
```

Three global attributes are available :

Name	Description
version	The version of ASYNCH used to generate this file
model	The model id used to generate this file
issue_time	The unix time at the beginning of the time serie

5.9 Peakflow Output

Peakflow outputs can be created in two formats: peakflow files (.pea) and database tables.

5.9.1 PEA Files

Peakflow files created with the "Classic" peakflow function take the structure:

```
{number of link}
{model type}
{link id 1} {area} {time to peak} {peakflow value}
{link id 2} {area} {time to peak} {peakflow value}
{link id 3} {area} {time to peak} {peakflow value}
```

The time to peak is measured since the beginning of the simulation. The peakflow value for each link is the maximum value achieved over the simulation for the state with index `0` in the state vector. The area given is the parameter from the link parameters with index area idx. See *SetParamSizes*.

5.9.2 Peakflow Database Queries

Peakflow output may be written to a database table if a database connection file is specified. One query is required, and one additional query is optional:

1. Query to create a table for uploading data
   - Inputs: table name
   - Returned tuples: none
2. Query to delete contents from a table
   - Inputs: table name
   - Returned tuples: none

The first query should create the table where the peakflow information is to be stored ASYNCH does NOT remove any existing data from the table, or check if the table exists already. The second query is optional, and will be used to delete any existing contents from the table before uploading data. The particular values uploaded to the database are determined through the peakflow function defined in *Peakflow Statistics Function Name*.

5.10 Link IDs for Time Series and Peakflows

Link ids must be specified for time series output and peakflow outputs. This can be done in one of two formats: save files (.sav) and database tables. Each of these formats is effectively just a list of link ids.

5.10.1 SAV Files

The structure of save files is:

```
{link id 1}
{link id 2}
{link id 3}
...
```

If a link id is specified in a save file, but is not present in the network, a warning will be issued, and the link id is ignored.

5.10.2 Save Database Queries

For pulling links from a database table, only one query is required:

1. Query to pull link ids
   - Inputs: none
   - Returned tuples: (link id)

5.11 Snapshot Output

Snapshot outputs can take multiple formats: files and database tables. The format for recovery and hdf5 files is covered in *Initial Values Input* as an input.

For using a database table, a database connection file is specified. The database connection file has three optional queries:

1. Query to create a database table before the first upload
   - Inputs: table name
   - Returned tuples: none
2. Query to delete a table before the first upload
   - Inputs: table name
   - Returned tuples: none
3. Query to truncate a table before every upload
   - Inputs: table name
   - Returned tuples: none

In practice, snapshots are often applied periodically as a way to create check points for the program. The third query allows the user to limit the number of snapshots in a table to one.

5.12 Runge-Kutta Data Input

Runge-Kutta Data files (.rkd) allow information about the underlying numerical methods to be specified link by link. These files are ASCII. The structure is given by:

```
{number of links}
{number of states}
{link id 1}
[absolute error tolerance 1] [absolute error tolerance 2]
[relative error tolerance 1] [relative error tolerance 2]
[dense absolute error tolerance 1] [dense absolute error tolerance 2]
[dense relative error tolerance 1] [dense relative error tolerance 2]
{RK Index for link id 1}
{link id 2}
[absolute error tolerance 1] [absolute error tolerance 2]
[relative error tolerance 1] [relative error tolerance 2]
[dense absolute error tolerance 1] [dense absolute error tolerance 2]
[dense relative error tolerance 1] [dense relative error tolerance 2]
{RK Index for link id 2}
...
```

An error tolerance is specified for every state at every link. The order of the links must match with the order given by the topology input, and number of states must agree with what the model expects.

5.13 Temporary Files

In general, sufficient memory is not available to hold a history of states while further computations take place. Thus, temporary files are created by ASYNCH to store time series outputs. These files are aggregated (typically after the simulation has completed) into the final time series output files (see *Time Series Output*).

Most users will not need to concern themselves with the underlying structure of these files. However, some routines exist for manipulating how these files are used, and an understanding of the temporary file structure is necessary for using those routines.

Temporary files are always in binary format. Every MPI process has its own file, which contains time series outputs for links assigned to the process. The format of the data looks like:

```
{link id 1} {expected number of steps}
{step 0, value 0} {step 0, value 1}
{step 1, value 0} {step 1, value 1}
{step 2, value 0} {step 2, value 1}
{link id 2} {expected number of steps}
{step 0, value 0} {step 0, value 1}
{step 1, value 0} {step 1, value 1}
{step 2, value 0} {step 2, value 1}
...
```

Because these files are a binary format, the presentation above is simply for readability. No new lines or spaces are present in the actual files. Only link ids for which time series data has been requested will appear in the files. Before any data is written, dummy values are placed in the files when they are created to insure the files are large enough. The number of these dummy files for each link is given by the expected number of steps value. This number is determined based upon the values of `maxtime` and the time resolution of the output time series when the temporary files are created.

Warning: Modifcations to these values after creation of the temporary files could create a situation where the files are not large enough to contain every point in the time series. This generally happens when `maxtime` is increased. Therefore, when the temporary files are created, the largest expected value of `maxtime` should be set. If the temporary files are not large enough, a warning will be displayed during calculations, and further outputs will be lost.

While calculations are performed, processes will write output data to these files, replacing whatever dummy values are present. Modifying the behavior of these files is generally not needed, but can be performed through various routines. See *C API*. The link ids and expected number of steps in temporary files are represented by unsigned 32 bit integers. The data types and sizes for the time series data will vary depending upon the desired outputs. If a built-in output time series is used for the states, these will appear in the temporary files as 64 bit floating point numbers.

CHAPTER 6

Built-in Options

Through a global file, many options are selected, including which model to use, what time series to output, and how to calculate peakflows. Each of these can be customized by the user. However, several precreated options do exist. This section provides a list of these options.

6.1 Built-In Output Time Series

Figure [fig: built-in output time series] contains the names and a description of built-in output time series. These outputs are defined in the source file *modeloutputs.c*. Up to seven states can be outputted with the built-in output time series. In addition to these, users can create their own time series outputs. See Section *Custom Outputs*.

Table 1: Built-in output time series

Output Name	Description
Time	Simulation time
TimeI	Simulation time, truncated to an integer
State0	State 0 of the model
State1	State 1 of the model
...	
State6	State 6 of the model

6.2 Built-In Peakflow Functions

Two built-in peakflow functions exist: *Classic* and *Forecast*. The two are described in Figure [fig: built-in peakflow functions]. The peak discharges are the largest values obtained in the state with index 0 in the state vectors. The time to peak for the *Classic* function is given in simulation time. For *Forecast*, the time to peak is measured in unix time. The time period output is a parameter that can be altered by user programs to provide additional output information.

Table 2: Built-in peakflow functions

Function Name	Outputs
Classic	Link ID, upstream area, time to peak, peak discharge
Forecast	Link ID, time to peak, peak discharge, time period

6.3 Built-In Runge-Kutta Methods

The ASYNCH solver is based upon using Runge-Kutta methods at the link level. These methods are selected either in the input global file or in a Runge-Kutta data file by the *RK index* in Table *Built-in RK methods*.

Table 3: Built-in RK methods

RK index	Name	Local order / Dense order
0	Kutta's Method	3 / 2
1	The RK Method	4 / 3
2	Dormand and Prince's Method	5 / 4
3	RadauII 3A	3 / 2

The application of these methods is done through the *RKSolver* routine in the *UnivVars* structure. This is set with a call to the *InitRoutines* method. See Section [sec: initroutines]. Several choices exist for the *RKSolver*. They are given in Table *Built-in RK solvers*. Some solvers are only appropriate if the model uses ODEs, while others support DAEs. Similarly, some methods support discontinuity states, while others do not. Currently, only one method is equipped to handle stiff ODEs. Certainly, the routine *ExplicitRKIndex1SolverDam* could be used to solve any problem. However, using a more appropriate solver is significantly more efficient.

Table 4: Built-in RK solvers

Name	DAEs	Discontinuities	Stiff
ExplicitRKSolver	No	No	No
ExplicitRKIndex1SolverDam	Yes	Yes	No
ExplicitRKIndex1Solver	Yes	No	No
ExplicitRKSolverDiscont	No	Yes	No
RadauRKSolver	No	No	Yes

CHAPTER 7

Built-in Models

Model Type	Description	States
21	Linear reservoir model, with dams	q, S, S_s, S_g
40	Linear reservoir model, with qvs dams	q, S, S_s, S_g
190	IFC linear model with constant runoff	q, s_p, s_s
191	IFC linear model with constant runoff	q, s_p, s_s, s_{precip}, V_r, q_b
192	IFC linear model with variable runoff	q, s_p, s_s, s_{precip}, V_r, q_b
195	IFC linear model, offline precip.	q, s_p, s_s, s_{precip}
196	IFC linear model, offline precip.	q, s_p, s_s, s_{precip}, s_{runoff}
252	IFC toplayer model	q, s_p, s_t, s_s
253	IFC toplayer model, with reservoirs	q, s_p, s_t, s_s
254	IFC toplayer model, with reservoirs	q, s_p, s_t, s_s, s_{precip}, V_r, q_b
256	IFC toplayer model with interflow	q, s_p, s_t, s_s, s_{precip}, s_{evap}, s_{runoff}, q_b
257	IFC toplayer model, variable velocity	q, s_p, s_t, s_s, s_{precip}, s_{et}, V_r, q_b
258	IFC toplayer model, offline precip.	q, s_p, s_t, s_s, s_{precip}, s_{evap}, s_{runoff}, q_b
259	IFC toplayer with offline, interflow	q, s_p, s_t, s_s, s_{precip}, s_{evap}, s_{runoff}, q_b

In this section, a description of a few different models is presented to demonstrate the features described in Section [sec: model descriptions]. These models are already fully implemented in `problems.c` and `definetype.c`, and may be used for simulations.

7.1 Main models

7.1.1 Constant Runoff Hydrological Model

This model describes a hydrological model with linear reservoirs used to describe the hillslope surrounding the channel. This is equivalent to a hillslope with a constant runoff. This model is implemented as model `190`.

Three states are modeled at every link:

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$s_p(t)$	Water ponded on hillslope surface $[m]$
$s_s(t)$	Effective water depth in hillslope subsurface $[m]$

where each state is a function of time (t), measured in $mins$.

These states are given as the solution to the differential equations

$$\begin{aligned}
\frac{dq}{dt} &= \frac{1}{\tau}\left(\frac{q}{q_r}\right)^{\lambda_1}(-q + (q_{pc} + q_{sc}) \cdot (A_h/60.0) + q_{in}(t)) \\
\frac{ds_p}{dt} &= p(t) \cdot c_1 - q_{pc} - e_p \\
\frac{ds_s}{dt} &= p(t) \cdot c_2 - q_{sc} - e_s.
\end{aligned}$$

Here, precipitation and potential evaporation are given as the time series $p(t)$ and $e_{pot}(t)$, measured in mm/hr and $mm/month$, respectively. The function $q_{in}(t)$ is the total discharge entering the channel from the channels of parent links, measured in m^3/s. A flux moves water from the water ponded on the surface to the channel, and another flux moves water from the subsurface to the channel. These are defined by:

$$\begin{aligned}
q_{pc} &= k_2 \cdot s_p \quad [m/min] \\
q_{sc} &= k_3 \cdot s_s \quad [m/min].
\end{aligned}$$

Further fluxes representing evaporation are given by:

$$\begin{aligned}
e_p &= Corr_{evap} \cdot C_p \cdot e_{pot}(t) \cdot u \quad [m/min] \\
e_s &= Corr_{evap} \cdot C_s \cdot e_{pot}(t) \cdot u \quad [m/min] \\
Corr_{evap} &= \begin{cases} \frac{1}{C_p + C_s}, & \text{if } C_p + C_s > 1, \\ 1, & \text{else} \end{cases} \\
C_p &= \frac{s_p}{e_{pot}(t)} \\
C_s &= \frac{s_s}{e_{pot}(t)}.
\end{aligned}$$

When potential evaporation is 0, the fluxes e_p and e_s are taken to be $0\ m/min$.

Some values in the equations above are constant in time, and are given by:

$$\begin{aligned}
u &= 10^{-3}/(30 \cdot 24 \cdot 60) \\
k_2 &= v_h \cdot L/A_h \cdot 60 \cdot 10^{-3} \quad [1/min] \\
k_3 &= v_g \cdot L/A_h \cdot 60 \cdot 10^{-3} \quad [1/min] \\
\frac{1}{\tau} &= \frac{60 \cdot v_r \cdot (A/A_r)^{\lambda_2}}{(1-\lambda_1) \cdot L \cdot 10^{-3}} \quad [1/min] \\
c_1 &= RC \cdot (0.001/60) \\
c_2 &= (1 - RC) \cdot (0.001/60) \\
q_r &= 1 \quad [m^3/s] \\
A_r &= 1 \quad [km^2].
\end{aligned}$$

Several parameters are required for the model. These are constant in time and represent:

Parameters	Description
A	Total area draining into this link $[km^2]$
L	Channel length of this link $[km]$
A_h	Area of the hillslope of this link $[km^2]$

Finally, some parameters above are constant in time and take the same value at every link. These are:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
RC	Runoff coefficient []
v_h	Velocity of water on the hillslope $[m/s]$
v_g	Velocity of water in the subsurface $[m/s]$

Let's walk through the required setup for this model. The above information for the model appears in three different source files: `definetype.c`, `problems.c`, and `problem.h` which is pretty bad and will be fix in the near future.

The function `SetParamSizes` contains the block of code for model `190`:

```
globals->dim = 3;
globals->template_flag = 0;
globals->assim_flag = 0;
globals->diff_start = 0;
globals->no_ini_start = globals->dim;
num_global_params = 6;
globals->uses_dam = 0;
globals->params_size = 8;
globals->iparams_size = 0;
globals->dam_params_size = 0;
globals->area_idx = 0;
globals->areah_idx = 2;
globals->disk_params = 3;
globals->num_dense = 1;
globals->convertarea_flag = 0;
globals->num_forcings = 2;
```

Each value above is stored into a structure called `GlobalVars`. Details about this object can be found in `GlobalVars`. Effectively, this object holds the values described in Section `SetParamSizes`. *dim* is set to `3`, as this is the number of states of the model (q, s_p, and s_s). This value is the size of the state and equation-value vectors. For the ordering in these vectors, we use:

States:	q	s_p	s_s
Index:	0	1	2

This ordering is not explicitly stated anywhere in code. Anytime a routine in `definetype.c` or `problems.c` accesses values in a state or equation-value vector, the routine's creator must keep the proper ordering in mind. *template_flag* is set to `0`, as no XML parser is used for the model equations. *assim_flag* is set to `0` for no data assimilation.

The constant runoff model consists entirely of differential equations (i.e. no algebraic equations), so *diff_start* can be set to the beginning of the state vector (index 0). *no_ini_start* is set to the dimension of the state vector. This means initial conditions for all 3 states must be specified by the source from the global file in the initial values section (see *Initial States*).

Six parameters are required as input which are uniform amongst all links. This value is stored in *num_global_params*. This model does use dams, so the *uses_dam* flag is set to `0` and *dam_params_size* is set to `0`.

Each link has parameters which will be stored in memory. Some of these values must be specified as inputs, while others can be computed and stored. For the constant runoff model, these parameters and the order in which we store them is

Parameters:	A	L	A_h	k_2	k_3	$invtau$	c_1	c_2
Index:	0	1	2	3	4	5	6	7

Each link has 8 parameters and no integer parameters. Thus *params_size* is set to 8 and *iparams_size* is set to `0`. The parameters A, L, and A_h are required inputs, while the others are computed in terms of the first three parameters and the global parameters. Therefore *disk_params* is set to `3`. The index *area_idx* is set to `0`, as `0` is the index of the upstream area. Similarly, *areah_idx* is set to `2` for the hillslope area. *convertarea_flag* is set to `0`, as the hillslope area will be converted to units of m^2, as shown below.

When passing information from one link to another downstream, only the channel discharge q is needed. So we set *num_dense* to `1`. Finally, two forcings are used in the constant runoff model (precipitation and evaporation), so *num_forcings* is set to 2.

In the `SetParamSizes` routine, an array *dense_indices* is created with a single element (the size is *num_dense*). For model `190`, the entry is set via:

```
globals->dense_indices[0] = 0;    //Discharge
```

Because the state q is passed to other links, its index in state vectors is put into the *dense_indices* array.

In the routine *ConvertParams*, two parameters are opted to receive a unit conversion:

```
params.ve[1] *= 1000;  //L: km -> m
params.ve[2] *= 1e6;   //A_h: km^2 -> m^2
```

The parameter with index 1 (L) is multiplied by 1000 to convert from km to m. Similarly, the parameter with index 2 (A_h) is converted to km^2 to m^2. Although these conversions are optional, the model differential equations contain these conversions explicitly. By converting units now, the conversions do not need to be performed during the evaluation of the differential equations.

In the routine `Precalculations`, each of the parameters for the constant runoff model are calculated at each link. The code for the calculations is:

```
else if(type == 190)
{
  double* vals = params.ve;
  double A = params.ve[0];
  double L = params.ve[1];
  double A_h = params.ve[2];
  double v_r = global_params.ve[0];
  double lambda_1 = global_params.ve[1];
  double lambda_2 = global_params.ve[2];
  double RC = global_params.ve[3];
  double v_h = global_params.ve[4];
  double v_g = global_params.ve[5];

  vals[3] = v_h * L / A_h * 60.0;    //k_2
  vals[4] = v_g * L / A_h * 60.0;    //k_3
  vals[5] = 60.0*v_r*pow(A,lambda_2) / ((1.0-lambda_1)*L); //invtau
  vals[6] = RC*(0.001/60.0);     //c_1
  vals[7] = (1.0-RC)*(0.001/60.0);  //c_2
}
```

Here, the array of parameters is named *vals* (simply as an abbreviation). The input parameters of the system are extracted (with the conversions from `ConvertParams`), and the remaining parameters are calculated, and saved into the corresponding index in *params*.

In the routine `InitRoutines`, the Runge-Kutta solver is selected based upon whether an explicit or implicit method is requested:

```
else if(exp_imp == 0)
  link->RKSolver = &ExplicitRKSolver;
else if(exp_imp == 1)
  link->RKSolver = &RadauRKSolver;
```

Other routines are set here:

```
else if(type == 190)
{
  link->f = &LinearHillslope_MonthlyEvap;
  link->alg = NULL;
  link->state_check = NULL;
  link->CheckConsistency =
  &CheckConsistency_Nonzero_3States;
}
```

The routines for the algebraic equations and the system state check are set to *NULL*, as they are not used for this model. The routines for the differential equations and state consistency are found in `problems.c`. The routine for the differential equations is `LinearHillslope_MonthlyEvap`:

```
void LinearHillslope_MonthlyEvap
(double t,VEC* y_i,VEC** y_p,
unsigned short int numparents,VEC* global_params,
double* forcing_values,QVSData* qvs,VEC* params,
IVEC* iparams,int state,unsigned int** upstream,
unsigned int* numupstream,VEC* ans)
{
  unsigned short int i;

  double lambda_1 = global_params.ve[1];

  double A_h = params.ve[2];
  double k2 = params.ve[3];
  double k3 = params.ve[4];
  double invtau = params.ve[5];
  double c_1 = params.ve[6];
  double c_2 = params.ve[7];

  double q = y_i.ve[0];      //[m^3/s]
  double s_p = y_i.ve[1];    //[m]
  double s_s = y_i.ve[2];    //[m]

  double q_pc = k2 * s_p;
  double q_sc = k3 * s_s;

  //Evaporation
  double C_p,C_s,C_T,Corr_evap;
  double e_pot = forcing_values[1] * (1e-3/(30.0*24.0*60.0)); //[mm/month] -> [m/min]

  if(e_pot > 0.0)
  {
    C_p = s_p / e_pot;
    C_s = s_s / e_pot;
    C_T = C_p + C_s;
  }
  else
  {
```

(continues on next page)

(continued from previous page)

```
    C_p = 0.0;
    C_s = 0.0;
    C_T = 0.0;
  }

  Corr_evap = (C_T > 1.0) ? 1.0/C_T : 1.0;

  double e_p = Corr_evap * C_p * e_pot;
  double e_s = Corr_evap * C_s * e_pot;

  //Discharge
  ans.ve[0] = -q + (q_pc + q_sc) * A_h/60.0;
  for(i=0;i<numparents;i++)
  ans.ve[0] += y_p[i]->ve[0];
  ans.ve[0] = invtau * pow(q,lambda_1) * ans.ve[0];

  //Hillslope
  ans.ve[1] = forcing_values[0]*c_1 - q_pc - e_p;
  ans.ve[2] = forcing_values[0]*c_2 - q_sc - e_a;
}
```

The names of parameters and states match with those defined in the mathematics above. The current states and hillslope parameters are unpacked from the state vector *y_i* and the vector *params*, respectively. The current precipitation value is available in *forcing_values[0]* and the current potential evaporation is available in *forcing_values[1]*. The fluxes q_{pc} and q_{sc} are calculated and used as *q_pc* and *q_sc*, respectively. The evaluation of the right side of the differential equations is stored in the equation-value vector *ans*. The channel discharges for the parent links are found in the array of state vectors *y_p[i]->ve[0]*, with *i* ranging over the number of parents.

The state consistency routine for the constant runoff model is called `CheckConsistency_Nonzero_3States`. It is defined as:

```
void CheckConsistency_Nonzero_3States(VEC* y,
VEC* params,VEC* global_params)
{
  if(y.ve[0] < 1e-14)     y.ve[0] = 1e-14;
  if(y.ve[1] < 0.0)   y.ve[1] = 0.0;
  if(y.ve[2] < 0.0)   y.ve[2] = 0.0;
}
```

The hillslope states s_p and s_s should not take negative values, as each is a linear reservoir. Similarly, the channel discharge q decays to 0 exponentially as the fluxes from the hillslope and upstream links goes to 0. However, because of the dependence upon q^{λ_1} in the equation for $\frac{dq}{dt}$, q must be kept away from 0. We therefore force it to never become smaller than $10^{-14}\ m^3/s$. It is worth noting that this restriction on q can only work if the absolute error tolerance for q is greater than $10^{-14}\ m^3/s$.

Each of these functions must also be declared in `problems.h`:

```
void LinearHillslope_MonthlyEvap(double t,VEC* y_i,  VEC** y_p,unsigned short int␣
↪numparents,  VEC* global_params,double* forcing_values,  QVSData* qvs,VEC* params,
↪IVEC* iparams,  int state,unsigned int** upstream,  unsigned int* numupstream,VEC*␣
↪ans);
void CheckConsistency_Nonzero_3States(VEC* y,  VEC* params,VEC* global_params);
```

The routine `ReadInitData` only needs to return a value of 0 for model `190`. All states are initialized from through a global file, as no algebraic equations exist for this model, and *no_ini_start* is set to *dim*. No state discontinuities are used for this model, so a value of 0 is returned.

7.1.2 Top Layer Hydrological Model

This model describes a hydrological model with nonlinear reservoirs used to describe the hillslope surrounding the channel. It features a layer of topsoil to create a runoff coefficient that varies in time. This model is implemented as model 254. The setup of the top layer model is similar to that of the constant runoff model presented in Section *Constant Runoff Hydrological Model*. However, the top layer model does make use of additional features.

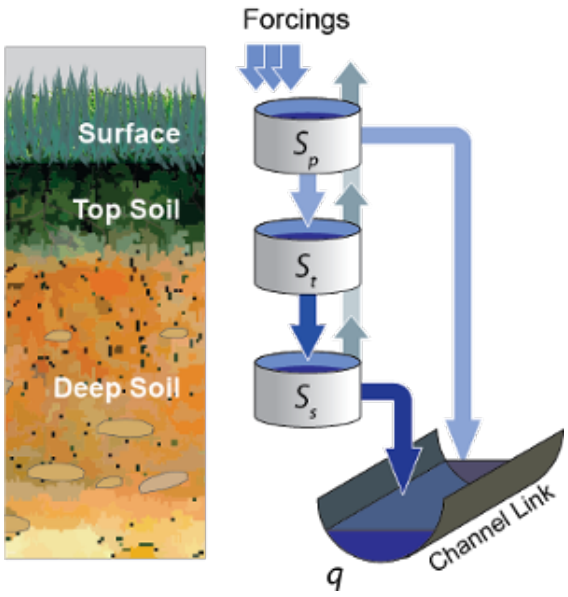


Fig. 1: The top layer hillslope model

Seven states are modeled at every link:

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$s_p(t)$	Water ponded on hillslope surface $[m]$
$s_t(t)$	Effective water depth in the top soil layer $[m]$
$s_s(t)$	Effective water depth in hillslope subsurface $[m]$
$s_{precip}(t)$	Total fallen precipitation from time 0 to t $[m]$
$V_r(t)$	Total flux of water from runoff from time 0 to t $[m^3/s]$
$q_b(t)$	Channel discharge from baseflow $[m^3/s]$

where each state is a function of time (t), measured in $mins$.

These states are given as the solution to the differential equations

$$\begin{aligned}
\frac{dq}{dt} &= \frac{1}{\tau}\left(\frac{q}{q_r}\right)^{\lambda_1}(-q + c_2 \cdot (q_{pc} + q_{sc}) + q_{in}(t)) \\
\frac{ds_p}{dt} &= c_1 p(t) - q_{pc} - q_{pt} - e_p \\
\frac{ds_t}{dt} &= q_{pt} - q_{ts} - e_t \\
\frac{ds_s}{dt} &= q_{ts} - q_{sc} - e_s \\
\frac{ds_{precip}}{dt} &= c_1 p(t) \\
\frac{dV_r}{dt} &= q_{pc} \\
\frac{dq_b}{dt} &= \frac{v_B}{L}(A_h q_{sc} - 60 \cdot q_b + q_{b,in}(t)).
\end{aligned}$$

Here, precipitation and potential evaporation are given as the time series $p(t)$ and $e_{pot}(t)$, measured in mm/hr and $mm/month$, respectively. The function $q_{in}(t)$ is again the total discharge entering the channel from the channels of parent links, measured in m^3/s. The function $q_{b,in}(t)$ is the total of the parents' baseflow, measured in $[m^3/s]$. Fluxes move water around the different layers of the hillslope, and other fluxes move water from the hillslope to the channel. These are defined by

$$\begin{aligned}
q_{pc} &= k_2 s_p \quad [m/min] \\
q_{pt} &= k_t s_p \quad [m/min] \\
q_{ts} &= k_i s_t \quad [m/min] \\
q_{sc} &= k_3 s_s \quad [m/min] \\
k_t &= k_2\left(A + B \cdot \left(1 - \frac{s_t}{S_L}\right)^{\alpha}\right) \quad [1/min].
\end{aligned}$$

Fluxes representing evaporation are given by

$$\begin{aligned}
e_p &= \frac{\frac{s_p}{s_r} \cdot u \cdot e_{pot}(t)}{Corr} \quad [m/min] \\
e_t &= \frac{\frac{s_t}{S_L} \cdot u \cdot e_{pot}(t)}{Corr} \quad [m/min] \\
e_s &= \frac{\frac{s_s}{h_b - S_L} \cdot u \cdot e_{pot}(t)}{Corr} \quad [m/min] \\
Corr &= \frac{s_p}{s_r} + \frac{s_t}{S_L} + \frac{s_s}{h_b - S_L}.
\end{aligned}$$

When potential evaporation is 0 or no water is present in the hillslope, the fluxes e_p, e_t, and e_s are taken to be $0\ m/min$.

Some values in the equations above are given by

$$
\begin{aligned}
u &= 10^{-3}/(30 \cdot 24 \cdot 60) \\
\frac{1}{\tau} &= \frac{60 \cdot v_r \cdot (A_{up}/A_r)^{\lambda_2}}{(1-\lambda_1) \cdot L \cdot 10^{-3}} \quad [1/min] \\
k_2 &= v_h \cdot L/A_h \cdot 60 \cdot 10^{-3} \quad [1/min] \\
k_i &= k_2 \beta \quad [1/min] \\
c_1 &= 0.001/60 \\
c_2 &= A_h/60 \\
q_r &= 1 \quad [m^3/s] \\
A_r &= 1 \quad [km^2] \\
s_r &= 1 \quad [m].
\end{aligned}
$$

Several parameters are required for the model. These are constant in time and represent:

Parameters	Description
A_{up}	Total area draining into this link $[km^2]$
L	Channel length of this link $[km]$
A_h	Area of the hillslope of this link $[km^2]$

Finally, some parameters above are constant in time and take the same value at every link. These are:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
v_h	Velocity of water on the hillslope $[m/s]$
k_3	Infiltration from subsurface to channel $[1/min]$
β	Percentage of infiltration from top soil to subsurface []
h_b	Total hillslope depth $[m]$
S_L	Total topsoil depth $[m]$
A	Surface to topsoil infiltration, additive factor []
B	Surface to topsoil infiltration, multiplicative factor []
α	Surface to topsoil infiltration, exponent factor []
v_B	Channel baseflow velocity $[m/s]$

Much of the required setup for this model is similar to that of the constant runoff coefficient model in Section *Constant Runoff Hydrological Model*. Only the significant changes will be mentioned here.

Several significant differences occur in the routine for `SetParamSizes`:

```
globals->dim = 7;
globals->no_ini_start = 4;
num_global_params = 12;
globals->params_size = 8;
globals->num_dense = 2;
globals->num_forcings = 3;
```

This model has a total of 7 states. However, initial values for only the first 4 must be provided. The others will be set by the routine `ReadInitData`. Therefore *no_ini_start* is taken to be 4. The ordering of the state vectors is given by

States:	q	s_p	s_t	s_s	q_{precip}	V_r	q_b
Index:	0	1	2	3	4	5	6

which means initial conditions for the states q, s_p, s_t, and s_s must be provided. For this model, we allow the possibility of a reservoir forcing the channel discharge q at a particular hillslope. So *num_forcings* is set to 3 (i.e. precipitation, potential evaporation, and reservoir forcing). Each link will require 2 states from upstream links: q and q_b. Accordingly, *num_dense* is set to 2, and *dense_indices* is set to

```
globals->dense_indices[0] = 0;   //Discharge
globals->dense_indices[1] = 6;   //Subsurface
```

In the routine `InitRoutines`, a special case is considered for links with a reservoir forcing. With no reservoir, the Runge-Kutta solver is unchanged from the constant runoff model. The other routines are set by

```
if(link->res)
{
  link->f = &TopLayerHillslope_Reservoirs;
  link->RKSolver = &ForcedSolutionSolver;
}
else
  link->f = &TopLayerHillslope_extras;
link->alg = NULL;
link->state_check = NULL;
link->CheckConsistency =
&CheckConsistency_Nonzero_AllStates_q;
```

If a reservoir is present, then instead of setting f to a routine for evaluating differential equations, it is set to a routine for describing how the forcing is applied:

```
void TopLayerHillslope_Reservoirs(double t,VEC* y_i,
VEC** y_p,unsigned short int numparents,
VEC* global_params,double* forcing_values,
QVSData* qvs,VEC* params,IVEC* iparams,int state,
unsigned int** upstream,unsigned int* numupstream,
VEC* ans)
{
  ans.ve[0] = forcing_values[2];
  ans.ve[1] = 0.0;
  ans.ve[2] = 0.0;
  ans.ve[3] = 0.0;
  ans.ve[4] = 0.0;
  ans.ve[5] = 0.0;
  ans.ve[6] = 0.0;
}
```

All states are taken to be 0, except the channel discharge. This state is set to the current forcing value from the reservoir forcing.

As mentioned earlier, the initial conditions for the last 3 states of the state vector are determined in the routine `ReadInitData`:

```
y_0.ve[4] = 0.0;
y_0.ve[5] = 0.0;
y_0.ve[6] = 0.0;
```

Clearly, these three states are all initialized to 0.

7.1.3 Linear Reservoir Hydrological Model

This model describes a hydrological model with linear reservoirs used to describe the hillslope surrounding the channel. This model includes the ability to replace channel routing with a model for a dam. This model is implemented as model 21.

Four states are modeled at every link:

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$S(t)$	Channel storage $[m^3]$
$s_t(t)$	Effective water depth in the top soil layer $[m]$
$s_g(t)$	Volume of water in the hillslope subsurface $[m^3]$

where each state is a function of time (t), measured in $mins$.

These states are given as the solution to the differential-algebraic equations

$$q = \begin{cases} \frac{1}{60\cdot\tau}(S/S_r)^{1/(1-\lambda_1)} & \text{if no dam present} \\ c_1 r^2 \left(\arccos(f) - f\sqrt{1-f^2} - \pi\right)\sqrt{2gh} & \text{if } h < d \\ c_1 O_a \sqrt{2gh} & \text{if } h < H_{spill} \\ c_1 O_a \sqrt{2gh} + c_2 L_{spill}\left(\frac{h-H_{spill}}{H_r}\right)^{3/2} & \text{if } h < H_{max} \\ c_1 O_a \sqrt{2gh} + c_2 L_{spill}\left(\frac{h-H_{spill}}{H_r}\right)^{3/2} & \\ \qquad + \frac{1}{60\cdot\tau}\left(\frac{S-S_{max}}{S_r}\right)^{1/(1-\lambda_1)} & \text{if } h > H_{max} \end{cases}$$

$$\frac{dS}{dt} = k_2 S_s + k_3 S_g - 60\cdot q + 60\cdot q_{in}$$

$$\frac{dS_s}{dt} = uRCp(t)A_h - k_2 S_s$$

$$\frac{dS_g}{dt} = u(1-RC)p(t)A_h - k_3 S_g.$$

Some values in the equations above are given by

$$\begin{aligned} u &= 10^{-3}/60 \\ g &= 9.81 \quad [m/s^2] \\ \frac{1}{\tau} &= \frac{60\cdot v_r \cdot (A/A_r)^{\lambda_2}}{(1-\lambda_1)\cdot L\cdot 10^{-3}} \quad [1/min] \\ k_2 &= v_h \cdot L/A_h \cdot 60\cdot 10^{-3} \quad [1/min] \\ k_3 &= v_g \cdot L/A_h \cdot 60\cdot 10^{-3} \quad [1/min] \\ O_a &= \frac{\pi}{4}d^2 \quad [m^2] \\ r &= d/2 \quad [m] \\ f &= (h-r)/r \quad [] \\ h &= H_{max}(S/S_{max})^{\alpha} \quad [m] \\ H_r &= 1 \quad [m] \\ S_r &= 1 \quad [m^3]. \end{aligned}$$

Several parameters are required for the model. These are constant in time and represent:

Parameters	Description
A	Total area draining into this link $[km^2]$
L	Channel length of this link $[km]$
A_h	Area of the hillslope of this link $[km^2]$

Some parameters above are constant in time and take the same value at every link. These are:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
RC	Runoff coefficient []
S_0	Initial effective depth of water on the surface and subsurface $[m]$
v_h	Velocity of water on the hillslope $[m/s]$
v_g	Velocity of water in the hillslope subsurface $[m/s]$

Additional parameters are required at links with a dam model:

Parameters	Description
H_{spill}	Height of the spillway $[m]$
H_{max}	Height of the dam $[m]$
S_{max}	Maximum volume of water the dam can hold $[m^3]$
α	Exponent for bankfull
d	Diameter of dam orifice $[m]$
c_1	Coefficient for discharge from dam
c_2	Coefficient for discharge from dam
L_{spill}	Length of the spillway $[m]$.

Every link has 7 local parameters. If a dam is present, 8 additional parameters are required. In the routine `SetParamSizes`, these values are used:

```
globals->params_size = 7;
globals->dam_params_size = 15;
```

Discontinuities in the states of the system occur because of the presence of dams. In `InitRoutines`, the appropriate Runge-Kutta solvers are set:

```
if(type == 21 && dam == 1)
  link->RKSolver = &ExplicitRKIndex1SolverDam;
else if(type == 21 && dam == 0)
  link->RKSolver = &ExplicitRKIndex1Solver;
```

Further routines are set:

```
if(dam)
  link->f = &dam_rain_hillslope;
else
  link->f = &nodam_rain_hillslope;
link->alg = &dam_q;
link->state_check = &dam_check;
link->CheckConsistency =
&CheckConsistency_Nonzero_4States;
```

Two different routines are used for the differential equations, depending upon whether a dam is present at the link. Although one routine could be used, considering separately the links with a dam and those without is more efficient. The possible discontinuity states in which a dam could be are indexed by:

Value	Meaning
0	No dam present
1	Water height in the dam is between the orifice diameter and the spillway
2	Water height in the dam is between the spillway and the height of the dam
3	Water height in the dam is above the height of the dam
4	Water height in the dam is below the orifice diameter

These indices are tracked by the *state_check* routine:

```
int dam_check(VEC* y,VEC* global_params,VEC* params, QVSData* qvs,unsigned int dam)
{
  if(dam == 0)    return 0;

  double H_spill = params.ve[7];
  double H_max = params.ve[8];
  double S_max = params.ve[9];
  double alpha = params.ve[10];
  double diam = params.ve[11];
  double S = y.ve[1];
  double h = H_max * pow(S/S_max,alpha);

  if(h < diam)        return 4;
  if(h <= H_spill)    return 1;
  if(h <= H_max)      return 2;
  return 3;
}
```

This model also uses an algebraic equation for channel discharge. The routine for this equation is:

```
void dam_q(VEC* y,VEC* global_params,VEC* params,  QVSData* qvs,int state,VEC* ans)
{
  double lambda_1 = global_params.ve[1];
  double invtau = params.ve[5];
  double S = (y.ve[1] < 0.0) ? 0.0 : y.ve[1];

  if(state == 0)
    ans.ve[0] = invtau/60.0*pow(S,1.0/(1.0-lambda_1));
  else
  {
    double orifice_area = params.ve[6];
    double H_spill = params.ve[7];
    double H_max = params.ve[8];
    double S_max = params.ve[9];
    double alpha = params.ve[10];
    double diam = params.ve[11];
    double c_1 = params.ve[12];
    double c_2 = params.ve[13];
    double L_spill = params.ve[14];
    double g = 9.81;

    double h = H_max * pow(S/S_max,alpha);
    double diff =
```

(continues on next page)

(continued from previous page)

```
    (h - H_spill >= 0) ? h - H_spill : 0.0;

    if(state == 1)
    ans.ve[0] =
    c_1*orifice_area*pow(2*g*h,.5);
    else if(state == 2)
    ans.ve[0] =
    c_1*orifice_area*pow(2*g*h,.5)
    + c_2*L_spill*pow(diff,1.5);
    else if(state == 3)
    ans.ve[0] =
    c_1*orifice_area*pow(2*g*h,.5)
    + c_2*L_spill*pow(diff,1.5)
    + invtau/60.0
    *pow(S-S_max,1.0/(1.0-lambda_1));
    else //state == 4
    {
      double r = diam/2.0;
      double frac =
      (h < 2*r) ? (h-r)/r : 1.0;
      double A =
      -r*r*(acos(frac)
      - pow(1.0-frac*frac,.5)*frac
      - 3.141592653589);
      ans.ve[0] = c_1*A*pow(2*g*h,.5);
    }
  }
}
```

Three initial states must be determined in the routine `ReadInitData`. The initial condition for the algebraic state q should be determined with a call to the algebraic equation routine. In addition, the two hillslope states must be set, and the initial state of the dam returned.

```
double RC = global_params.ve[3];
double S_0 = global_params.ve[4];
double A_h = params.ve[2];
y_0.ve[2] = RC * S_0 * A_h;
y_0.ve[3] = (1.0 - RC) * S_0 * A_h;

state = dam_check(y_0,global_params,params,qvs,dam);
dam_q(y_0,global_params,params,qvs,state,y_0);
return state;
```

7.2 Additional models

In this section it is presented descriptions of some models that are less popular, more specific or that are still under testing & revision phase.

7.2.1 IFC linear model with constant runoff extended

The model `191` can be seen as an extension of model `190` but with three aditional states ($s_{precip}(t)$, $V_r(t)$, $q_b(t)$):

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$s_p(t)$	Water ponded on hillslope surface $[m]$
$s_s(t)$	Effective water depth in hillslope subsurface $[m]$
$s_{precip}(t)$	Total fallen precipitation from time 0 to t $[m]$
$V_r(t)$	Total volume of water from runoff from time 0 to t $[m^3]$
$q_b(t)$	Channel discharge from baseflow $[m^3/s]$

The states $q(t)$, $s_p(t)$ and $s_s(t)$ are obtained as in `191`. The new states are governed by:

$$\frac{ds_{precip}}{dt} = p(t) \cdot c_1$$
$$\frac{dV_r}{dt} = q_{pc}$$
$$\frac{dq_b}{dt} = ((q_{sc} \cdot A_h) - (q_b \cdot 60.0)) \cdot (\frac{v_B}{L}).$$

in which $p(t)$, c_1, q_{pc}, q_{sc}, A_h and L are defined in the description of model `190`, and v_B is an additional global parameter, so that se set of global parameters for this model is given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
RC	Runoff coefficient []
v_h	Velocity of water on the hillslope $[m/s]$
v_g	Velocity of water in the subsurface $[m/s]$
v_B	Channel baseflow velocity $[m/s]$

One addition of this model is the support to artificaily controlled reservoirs, so that the set of forcings is given by:

Forcing	Description
$p(t)$	Precipitation $[mm/hour]$
$e_{pot}(t)$	Potential evapotranspiration $[mm/hour]$
$Res(t)$	Artificial reservoirs $[m^3/s]$

7.2.2 IFC linear model with variable runoff

The model `192` is almost identical to model `191`, with both presenting the same states ($q(t)$, $s_p(t)$, $s_s(t)$, $s_{precip}(t)$, $V_r(t)$, $q_b(t)$), same set of local parameters (A, L, A_h) and forcings ($p(t)$, $e_{pot}(t)$, $Res(t)$). The difference is that, instead of having the infiltration governed by RC as:

$$c_2 = (1 - RC) \cdot (0.001/60)$$
$$\frac{ds_s}{dt} = p(t) \cdot c_2 - q_{sc} - e_s.$$

the model has it depending on a new global parameter $k_{Ifactor}$ that replaces RC by:

$$k_I = k_{Ifactor} \cdot v_h \cdot L/A_h \cdot 60$$
$$\frac{ds_s}{dt} = k_I \cdot s_p - q_{sc} - e_s$$

This way the set global parameters is given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
$k_{Ifactor}$	Multiplicative factor for infiltration process []
v_h	Velocity of water on the hillslope $[m/s]$
v_g	Velocity of water in the subsurface $[m/s]$
v_B	Channel baseflow velocity $[m/s]$

7.2.3 IFC linear model, offline precipitation

The model `195` is very similar to models `191` and `192`, with the same local parameters (A, L, A_h). The same set of states is similar, except by the removal of $V_r(t)$ (being: $q(t)$, $s_p(t)$, $s_s(t)$, $s_{precip}(t)$, $q_b(t)$).

The difference is that the precipitation forcing $p(t)$ is replaced by two other forcings: surface runoff ($r(t)$) and $i(t)$. This way, the partition of the rainfall into surface runoff and infiltration is not executed within Asynch, but it is expected to be performed as a pre-processing step. Thus, the forcings can be summarized as:

Forcing	Description
$r(t)$	Surface Runoff $[mm/hr]$
$i(t)$	Infiltration $[mm/hr]$
$e_{pot}(t)$	Potential evapotranspiration $[mm/month]$

and the affected differential equations are given by:

$$\frac{ds_p}{dt} = r(t) \cdot (\frac{0.001}{60.0}) - q_{pc}$$
$$\frac{ds_s}{dt} = i(t) \cdot (\frac{0.001}{60.0}) - q_{sc} - e_s.$$

As no parameter is necessary to manipulate the separation of precipitation into surface runoff and infiltration, k_{RC} and $k_{Ifactor}$ are absent, so that the global parameters are given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
v_h	Velocity of water on the hillslope $[m/s]$
v_g	Velocity of water in the subsurface $[m/s]$
v_B	Channel baseflow velocity $[m/s]$

7.2.4 IFC linear model, offline precipitation extended

The model `196` can be seen as an extension to model `195`. All local parameters and global parameters are the same. A new state (s_{runoff}) is added, so that the set of states is given by:

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$s_p(t)$	Water ponded on hillslope surface $[m]$
$s_s(t)$	Effective water depth in hillslope subsurface $[m]$
$s_{precip}(t)$	Total fallen precipitation from time 0 to t $[m]$
$s_{runoff}(t)$	Total column of water from runoff from time 0 to t $[m]$
$q_b(t)$	Channel discharge from baseflow $[m^3/s]$

With the differential equation of the new state being given by:

$$\frac{ds_{runoff}}{dt} = r(t) \cdot (\frac{0.001}{60.0})$$

It also includes support for Reservoirs forcing, so that the set of forcings is given by:

Forcing	Description
$r(t)$	Surface Runoff $[mm/hr]$
$i(t)$	Infiltration $[mm/hr]$
$e_{pot}(t)$	Potential evapotranspiration $[mm/month]$
$Res(t)$	Artificial reservoirs $[m^3/s]$

7.2.5 IFC toplayer model with interflow

The model `256` can be seen as an extension to model `254`. One more state (s_{evap}) is present, so that the set of states is given by:

State	Description
$q(t)$	Channel discharge $[m^3/s]$
$s_p(t)$	Water ponded on hillslope surface $[m]$
$s_t(t)$	Effective water depth in the top soil layer $[m]$
$s_s(t)$	Effective water depth in hillslope subsurface $[m]$
$s_{precip}(t)$	Total fallen precipitation from time 0 to t $[m]$
$s_{evap}(t)$	Total evaporation estimated from time 0 to t $[m]$
$V_r(t)$	Total volume of water from runoff from time 0 to t $[m^3]$
$q_b(t)$	Channel discharge from baseflow $[m^3/s]$

The differential equation that were modified of added (when compared to model `254`) are given by:

$$\frac{dq}{dt} = \frac{1}{\tau}\left(\frac{q}{q_r}\right)^{\lambda_1}(-q + c_2 \cdot (q_{pc} + q_{tc} + q_{sc}) + q_{in}(t))$$

$$\frac{ds_{evap}}{dt} = q_{pt} - q_{tc} - q_{ts} - e_t$$

$$\frac{ds_{evap}}{dt} = e_{pot}(t) * \frac{0.001}{60.0}$$

As it can be observed, a new flow is added (q_{tc}). This flow received the name of *interflow* and is given by:

$$q_{tc} = k_{tc} * s_t$$

in which $k_{tc}$is an additional global parameter. Thus, the set of global parameters is given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
v_h	Velocity of water on the hillslope $[m/s]$
k_3	Infiltration from subsurface to channel $[1/min]$
β	Percentage of infiltration from top soil to subsurface []
h_b	Total hillslope depth $[m]$
S_L	Total topsoil depth $[m]$
A	Surface to topsoil infiltration, additive factor []
B	Surface to topsoil infiltration, multiplicative factor []
α	Surface to topsoil infiltration, exponent factor []
v_B	Channel baseflow velocity $[m/s]$
k_{tc}	Interflow coefficient $[1/min]$

The local parameters of each hillslope and the forcings are the same as in `254`.

7.2.6 IFC toplayer model, offline precipitation

The model `258` can be seen as a the model `254` with the forcings adopted by model `195`. It presents the same states as in model `257` ($q(t)$, $s_p(t)$, $s_t(t)$, $s_s(t)$, $s_{precip}(t)$, $s_{evap}(t)$, $V_r(t)$, $q_b(t)$), but with a slightly difference in the order ($q(t)$, $s_p(t)$, $s_t(t)$, $s_s(t)$, $s_{precip}(t)$, $V_r(t)$, $s_{evap}(t)$, $q_b(t)$).

The differential equations are the same as for model `254`, except by:

$$\frac{ds_p}{dt} = (r(t) \cdot \frac{0.001}{60}) - q_{pc} - e_p$$
$$\frac{ds_t}{dt} = (i(t) \cdot \frac{0.001}{60}) - q_{ts} - e_t$$

And the forcings are the same as for model `196`, it is:

Forcing	Description
$r(t)$	Surface Runoff $[mm/hr]$
$i(t)$	Infiltration $[mm/hr]$
$e_{pot}(t)$	Potential evapotranspiration $[mm/month]$
$Res(t)$	Artificial reservoirs $[m^3/s]$

While the set of local parameters are kept the same (A, L, A_h), the set of global parameters is given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
v_h	Velocity of water on the hillslope $[m/s]$
k_3	Infiltration from subsurface to channel $[1/min]$
β	Percentage of infiltration from top soil to subsurface []
h_b	Total hillslope depth $[m]$
S_L	Total topsoil depth $[m]$
v_B	Channel baseflow velocity $[m/s]$

7.2.7 IFC toplayer with offline precipitation and interflow

The model `259` can be seen as an extension of model `258`, adding the concept interflow adopted by model `256`. The states ($q(t)$, $s_p(t)$, $s_t(t)$, $s_s(t)$, $s_{precip}(t)$, $s_{evap}(t)$, $V_r(t)$, $q_b(t)$), forcings ($r(t)$, $i(t)$, $e_{pot}(t)$, $Res(t)$) and local parameters (A, L, A_h), are the same as in model `258`.

A significative change in the interface is due to the inclusion of the global parameter k_{tc} to govern the interflow, so the the set of global parameters is given by:

Parameters	Description
v_r	Channel reference velocity $[m/s]$
λ_1	Exponent of channel velocity discharge []
λ_2	Exponent of channel velocity area []
v_h	Velocity of water on the hillslope $[m/s]$
k_3	Infiltration from subsurface to channel $[1/min]$
β	Percentage of infiltration from top soil to subsurface []
h_b	Total hillslope depth $[m]$
S_L	Total topsoil depth $[m]$
v_B	Channel baseflow velocity $[m/s]$
k_{tc}	Interflow coefficient $[1/min]$

CHAPTER 8

Data Assimilation

In this section, we discuss the current implementation of the Data Assimilation and the usage of the `assim` command line interface.

8.1 Introduction

The idea is to combine the best different sources of information to estimate the state of a system:

- model equations
- observations, data
- background, a priori information
- statistics

In our specific application, we combine our hydrological model with stream flow observations.

In the following sections, the *Top Layer Hydrological Model* will be used to illustrate the theory of data assimilation.

8.1.1 Best Linear Unbiased Estimator (BLUE)

We aim at producing an estimate x^a of the true state $x^t = (q, S_p, S_t, S_s)$ of the hillslope/river network at initial time, to initialize forecasts.

We are given:

- a background estimate $x^b = (q^b, S_p^b, S_t^b, S_s^b)$ with covariance matrix B given, from a previous forecast,
- partial observations $y^0 = \mathcal{H}(x^t) + \varepsilon^0$, with covariance matrix R given, e.g. water elevation from bridge sensors. Observation operator $\mathcal{H}$ maps the input parameters to the observation variables, in our case it would be the rating curves.

We also assume that:

- $\mathcal{H} = H$ is a linear operator

The best estimate x^a is searched for as a linear combination of the background estimate and the observation y^o:

$$x^a = Lx^b + Ky^o$$

8.1.2 Variational method

We can rewrite the BLUE as an equivalent variational optimization problem (optimal least squares) also :

$$x^a = min(\mathcal{J})$$

where the cost function $\mathcal{J}$ to minimize is:

$$\mathcal{J}(x) = (x - x^b)^T B^{-1}(x - x^b) + (y^o - H(x))^T R^{-1}(y^o - H(x))$$

Let's introduce a model operator:

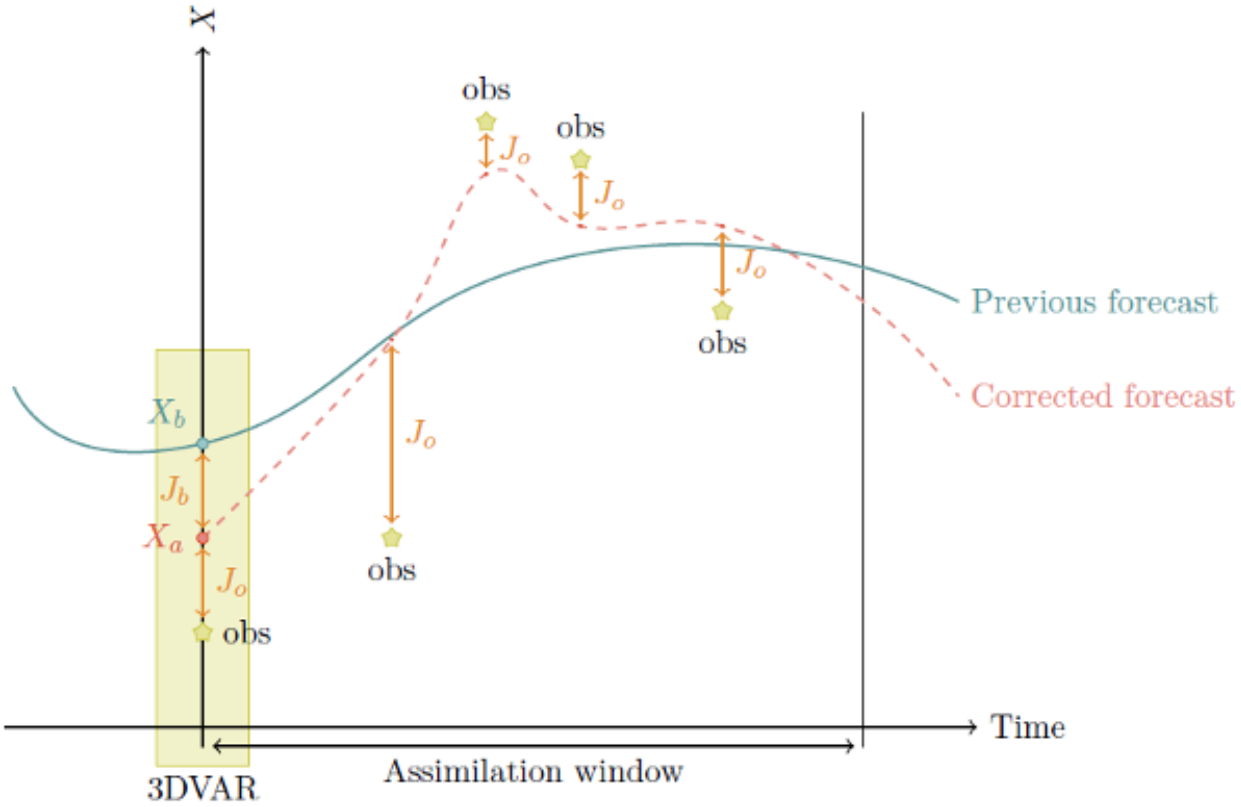
$$x_k^t = \mathcal{M}_{k,k-1}(x_{k-1}^t) = \mathcal{M}_{0\to k}(x_0^t)$$

Since our problem is time-dependant (4D-Var), and the unknown x is the initial state vector:

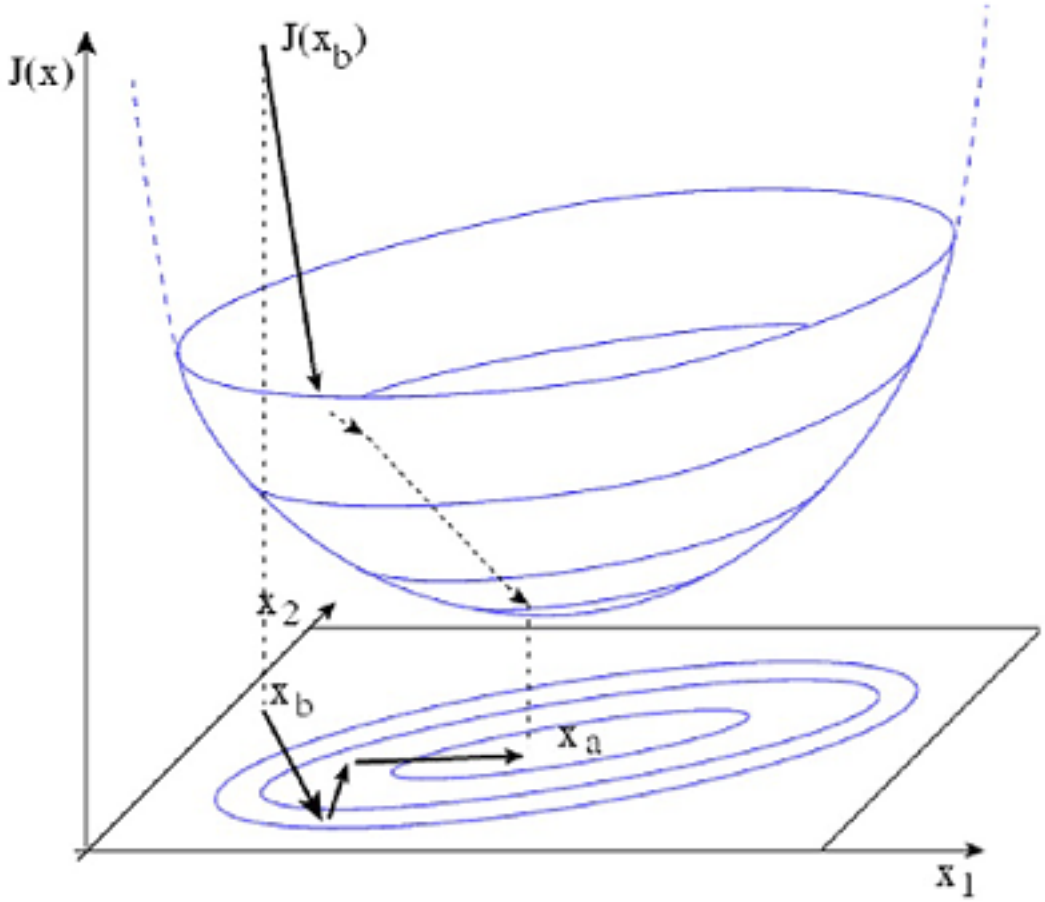
$$\mathcal{J}(x) = \underbrace{(x - x^b)^T B^{-1}(x - x^b)}_{\mathcal{J}_b} + \underbrace{(y^o - H_k(\mathcal{M}_{0\to k}(x)))^T R^{-1}(y^o - H_k(\mathcal{M}_{0\to k}(x)))}_{\mathcal{J}_o}$$

Where $\mathcal{J}_b$ to minimize the distance to the a priori information and $\mathcal{J}_o$ minimize the distance to the observations.

This is illustrated in the following figure:



Now that $\mathcal{J}$ is defined (i.e. once all the ingredients are chosen: control variables, error statistics, norms, observations...), the problem is entirely defined. Hence its solution. To minimize $\mathcal{J}$ we will be using a descent method:



Descent methods to minimize a function require knowledge of (an estimate of) its gradient. Obtaining the gradient through the computation of growth rates is unpractical since it require N + 1 runs, where $N = [x]$. In our state wide application, $N = O(7)$.

There are two typical methods to get an estimate of the gradient with one run of the model:

- Adjoint model
- Forward Sensitivity equations

The later is used in this implementation and sensitivity to background equations are solved along the model equations.

8.1.3 Forward Sensitivity Methods (FSM)

TODO: show relation between $\nabla\mathcal{J}(x)$ and FSM.

8.1.4 Problem simplification

Although the equations are numerous, and the corresponding least squares problem requires solving linear systems of equations, several observations can be made to reduce the overall workload:

- Ungagged sub-basin are removed

- Zone of influence is defined as an area upstream the gage that has an influence on the discharge at gage during the assimilation window. For that we use constant streamflow velocity to assess the maximum distance

8.2 Installation

Data Assimilation is available only if Asynch is built with the `PETSc` library. Refer to the *Installation* for more information. Make sure that `./configure` returns:

```
checking for PETSC... yes
```

8.3 Configuration

`assim` requires an additional configuration `.das` file on the command line, for exemple:

```
assim turkey_river.gbl turkey_river.das
```

8.3.1 Overview

Here is a typical `.das` file taken from the examples folder:

```
%Model variant
254_q

%Observation input (discharge)
%Time resolution of observations
assim51.dbc 15.0

%Step to use (assimilation window)
%12 % use 3 hours
%24 % use 6 hours
48 % use 12 hours
%96 % use 24 hours

%Max least squares iterations
5

# %End of file
--------------------------------
```

8.3.2 Model variant

Format:

```
{model id}
```

This string value specifies which assimilation model is used and which state variable initial conditions are optimized.

Id	Model	State variable
254	Top Layer Model	Every state variable
254_q	Top Layer Model	Discharge
254_qsp	Top Layer Model	Discharge, pond storage
254_qst	Top Layer Model	Discharge, top layer storage

8.3.3 Observation input

Format:

```
{.dbc filename} {time resolution}
```

The observation data are pulled from a PostgreSQL database. The database connection filename can include a path. The file should provide three queries in the following order:

1. A query that returns the link_id where observation (gages) are available with the following schema `CREATE TABLE (link_id integer)`.
2. A query that returns observation for a given time frame (where begin and end time stamp are parameter) with the following schema `CREATE TABLE (link_id integer, datetime as integer, discharge real)`.
3. A query that returns the distance to the border of the domain for the gages with the following schema `CREATE TABLE (link_id integer, distance real)`.

The time resolution is a floating point number with units in minutes.

8.3.4 Assimilation Window

Format:

```
{num observations}
```

The duration of the assimilation window expressed in number of time steps.

8.4 Forecaster

Running a forescaster with data assimilation requires to run a background simulation with `asynch` followed by the analysis with `assim`. And then generate the forecast using the analysed state as initial conditions. Here are the typical steps to

1. First the model needs to be intialized, for instance, with hydrostatic conditions. Given the discharge at the outlet q and the draining area $\mathcal{A}$, compute the equivalent precipitation p_{eq} and using dry uniform intial condition run the model for a long period with the equivalent precipitation. This will fill up the watershed.

$$p_{eq} = \frac{q}{\mathcal{A}}$$

2. Then run a warmup period of 15 days (or wathever the travel time is for your waterhed) with real precipitation data. At this point we should have realistic initial conditions.
3. Finally run the following algorithm:

```
ON discharge_observation
  // Generate the background
  RUN asynch for obs time step

  // Generate the assimilated state
  RUN assim for the assimiliation window

  // Generate forecast
  RUN asynch for the forecast lead time
```

Here is a snippet of an implementation of this algorithm in Javascript:

```
//Get the initial condition file (the timestamp is in the filename)
var file = getLatestFile(/^background_(\d+).(rec|h5)$/);

//Main time loop
while (file.timestamp < endTime) {

  // Assimilation window
  const assim_window = 12 * 60

  // Steps of 6 hours
  const duration = 6 * 60;

  // Generate the config file for DA
  render(templates.assim, 'assim.gbl', {
    duration: assim_window,
    begin: file.timestamp,
    end: file.timestamp + assim_window * 60
  });

  cp.execFileSync('mpiexec', ['-n', '4', 'assim', 'assim.gbl', 'assim.das'],
↪{stdio:[0,1,2]});

  // Generate the config file for the background (regular asynch run)
  render(templates.background, 'background.gbl', {
    duration: duration,
    begin: file.timestamp,
    end: file.timestamp + duration * 60
  });

  cp.execFileSync('mpiexec', ['-n', '1', 'asynch', 'background.gbl'], {stdio:[0,1,2]}
↪);

  file.timestamp += duration * 60;
}
```

The full implementation is available in the `examples/assim` folder.

8.5 Notes

Note: The author of these docs is not the primary author of the code so some things may have been lost in translation.

Data assimilation is implemented only for the *Top Layer Hydrological Model* (`254`). Implementing Data Assimi-

lation requires the user to provide additional model's equations. A more generic method could be used (Jacobian approximation) but would probably be less efficient.

Data assimilation only works with discharge observations (or whatever the first state variable is). This is currently hardcoded but could be extended to support other types of observation such as soil moisture.

Observations should be interpolated to get a better assimilated states (especially for locations that are close to observations). For instance with discharge observations available at a 15 minutes time step, links that are upstream at a distance $d < 15 * v_0$ are not corrected.

The larger the assimilation window, the larger is the domain of influence upstream the gages and the better the corrected state. A short assimilation window would only make correction to the links close to the gage and that could induce some ossilations. In Iowa 12 hours, seems to be the sweet spot between computation time and correction.

The solution of the equations at a link depends on the upstreams links and not only the direct parent links. This difference between the forward model and the assimilation model makes Asynch less suitable for solving the system of equations. To be more specific, the partionning of the domain between processors is more senstive since a bad partionning may results in a lot a transfers between procs. Eventually a solver like CVODES (Sundials), that solves the sensitivity equations, may be more appropriate.

For small watersheds (N <= 15K links, i.e. Turkey River), `assim` works best using serial execution (num procs = 1).

The performances of the assimilation are not very good when the correction of discharge is negative (falling limb).

Discontinuities (i.e. at reservoirs) are not supported.

Strong nonlinearities could be problem. The extension of 4D-Var to non linear problems, called Incremental 4D-Var, may be more appropriate.

8.6 Bibliography

CHAPTER 9

Cookbook

This is not a section about French cuisine, although you will find pretty good recipes here and contribution from our users.

9.1 Preprocessing

9.1.1 Generating a .rvr file from the DB for a specific subassin

The following query retrieve records suitable for generating a .rvr file. For Turkey River at Garber (434514):

```
WITH
outlet_link AS (SELECT * FROM master_update WHERE link_id = 434514),
waterhshed_links(link_id, parent_link_id) AS (SELECT master_update.link_id, master_
↪update.parent_link FROM master_update, outlet_link WHERE master_update.left BETWEEN␣
↪outlet_link.left AND outlet_link.right)
SELECT link_id, coalesce(array_length(parents_link_id, 1), 0), parents_link_id FROM (
  SELECT a.link_id AS link_id, array_remove(array_agg(b.link_id), NULL) AS parents_
↪link_id
  FROM waterhshed_links a LEFT JOIN waterhshed_links b ON (b.parent_link_id = a.link_
↪id)
  GROUP BY a.link_id) a
ORDER BY link_id;
```

9.2 Postprocessing

9.2.1 Reading the HDF5 outputs with Python

You may have to install the `tables` package.

```
pip install tables
```

Reading PyTables is pretty straighforward:

```
import tables

# Open the file in read-only mode
h5file = tables.open_file("outputs.h5", "r")

# Print some info about the file
print(h5file)

for row in h5file.root.outputs:

    # Read current row content
    # The name of the comuns depends on your outputs settings
    link_id = row['LinkID']
    time = row['Time']
    state_0 = row['State0']

    # Do something with the values
    print "link_id:", link_id, "time:", time, "state_0:", state_0

  # Close the file
  h5file.close()
```

Here is an other way using the the `numpy` and `h5py` package:

```
import numpy as np
import h5py

# open file and iterate over each row
with h5py.File(h5_file_path, "r") as hdf_file:
    hdf_file_content = np.array(hdf_file.get("outputs"))
    for i in range(len(hdf_file_content)):
        # read current row content
        cur_row = hdf_file_content[i]
        # get the values of each columns
        cur_linkid = cur_row[0]
        cur_time = cur_row[1]
        cur_state0 = float(cur_row[2])
```

CHAPTER 10

Installation

This section provides a quick guide for compiling and running ASYNCH on a Unix based system, including Iowa's local HPC clusters. Windows users can use the Visual Studio project in the `ide\msvc` folder.

10.1 Requirements

In addition to the source code, several programs and libraries are needed:

- A C compiler
- GNU Make
- An MPI implementation
- HDF5, libpq libraries

A brief description of each is provided below.

10.1.1 C Compiler

ASYNCH is written in ANSI C89 standard. ASYNCH has been successfully compiled and tested with the GNU C compiler gcc (version 4.6 and later), Microsoft Visual Studio 2015 and Intel C++ compiler. Using Intel's compiler is recommanded if available.

10.1.2 GNU Make

GNU Make is a utility for directing the generation of binaries from source code. `make` is available in most Linux distro.

10.1.3 MPI Implementation

The Message Passing Interface (MPI) is a standard for transferring data on parallel computers ASYNCH uses MPI for communication between processes to perform calculations in parallel ASYNCH has been successfully used and tested with `OpenMPI`.

From the OpenMPI webpage :

> The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. OpenMPI is available on Iowa's HPC clusters In theory, ASYNCH should work properly with any other implementation adhering to the MPI standard.

If installing an MPI implementation on a machine for ASYNCH, be sure to install the development packages of the MPI implementation. In Linux repositories, these packages are usually denoted with a `-dev` or similar in the package name. If in doubt, try typing "mpirun" and "mpicc" in a terminal. Both of these should be present to run ASYNCH. If mpirun is not present, you have not installed the MPI binaries (meaning you probably haven't tried installing MPI at all). If mpicc is not present, then you are missing the development package.

10.1.4 HDF5 Library

`HDF5` is a format and library widely used for storing binary scientific data in an efficient and portable way.

From the HDF5 webpage :

> HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for filexible and efficient I/O and for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format.

For ASYNCH build, the 1.8.x branch is used.

10.2 Optional Libraries

10.2.1 PostgreSQL Library

`libpq` is a library for communicating with a PostgreSQL database, created by the makers of PostgreSQL.

From the libpq webpage :

> libpq is the C application programmer's interface to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL back-end server and to receive the results of these queries.

libpq can be ommitted while running `./configure` with `--without-postgresql`. If you want to use libpq on a machine for ASYNCH, be sure to install the PostgreSQL development packages. In Linux repositories, these packages are usually denoted by a "-dev" or similar in the package name.

10.2.2 METIS Library

`METIS` is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices.

From the METIS webpage :

> Experiments on a large number of graphs arising in various domains including finite element methods, linear programming, VLSI, and transportation show that METIS produces partitions that are consistently better than those produced by other widely used algorithms. The partitions produced by METIS are consistently 10% to 50% better than those produced by spectral partitioning algorithms.

METIS can be ommitted while running `./configure` with `--without-metis`.

10.2.3 PETSc Library

`PETSc` is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.

From the PETSc webpage :

> PETSc is intended for use in large-scale application projects, many ongoing computational science projects are built around the PETSc libraries. PETSc is easy to use for beginners. Moreover, its careful design allows advanced users to have detailed control over the solution process.

PETSc can be ommitted while running `./configure` with `--without-petsc`.

10.3 Optional Software

Some additional software is available that may be useful, depending upon what the user wishes to do. These include

- Git
- dos2unix
- NoMachine NX

10.3.1 Git

Git is a distributed revision control system. Although not needed for running ASYNCH, the source code repository does require Git for access. Git is in most Linux repositories. GitHub offers guides about its usage.

10.3.2 dos2unix

This is a useful utility if editing input text files for ASYNCH from a Windows machine. Unix and Windows use a slightly different format for text documents. Although a file may look the same under both a Linux and Windows text editor, subtle diferences can still exist. In general, editing a text file from Linux on a Windows machine will convert the file to the Windows format. To change the format to Unix, use the utility dos2unix. If a file is already under Unix format, this utility will not modify the file Using a text file in Windows format with ASYNCH will result in errors. This process can be slow, depending upon the size of the text files involved As such, ASYNCH does not automatically check the format of input text files. dox2unix can be found in most Linux repositories.

10.3.3 FastX 2

FastX is a program for connecting to the HPC systems with a GUI desktop environment. It is similar to No Machine connections but is newer and a little more robust when using Duo 2 factor authentication. This can be useful for users wishing to access an Iowa HPC resource, though this is not the only way. Information about using and obtaining FastX for Iowa HPC resources can be found at FastX connections.

10.4 Source Code, Compiling, and Running ASYNCH

The ASYNCH source code is available in a repository hosted by GitHub. Downloading on of the release version the code from the repository requires the use of Git See *Git*. The source code can also be downloaded directly from GitHub as a zip file.

If the source code is ever updated, you may want to run `make clean` before recompiling. This removes all binaries and object files of the old version. Once compiled, ASYNCH can be run with the command:

```
mpirun -np <number of processes> <path>/asynch < gbl filename>
```

10.4.1 Updating the package

This operation is only necessary if you cloned the git repository. If you are using a release source tarball, you can skip to the next step.

```
autoreconf --install
make dist
```

10.4.2 Installing the package

These are the generic instruction for an out of source build (prefered method):

```
mkdir build && cd build
../configure CFLAGS="-O3 -DNDEBUG"
make
make check
make install
```

Note: Newer version of gcc requires to add `-Wno-format-security` to `CFLAGS` argument so that the configure script should be invoked with `../configure CFLAGS="-DNDEBUG -Wno-format-security"`.

Other typical `configure` arguments are also available, such as:

- **–prefix**: defines the folder path of the output compiled runnable.
  - Example: `--prefix=/home/user/asynch/dist`
- **CC**: defines the C compiler to be used.
  - Example: `CC=gcc` (GNU compiler) or `CC=icc` (intel compiler)

10.5 Iowa HPC Clusters

Currently, the executable used on Neon and Argon ar maintained by yours truly. Users of Iowa's HPC resources should NOT need to download and compile source code on Neon and Argon. Binaries for ASYNCH are located in `/Dedicated/IFC/.neon/bin` on Neon and in `/Dedicated/IFC/.argon/bin` on Argon. Libraries for linking *libasynch* with your own software are located in the directory `/Dedicated/IFC/.<cluster>/lib`. As of the compilation of derived work, all required software should be available. The build system included with the source code should work without modification on these clusters.

10.5.1 Setting up the environment on ARGON

These clusters do use third party software through modules. The module for OpenMPI and HDF5 must be loaded once per login session to run ASYNCH. Refers to the *Getting started* section for more information. For Argon:

```
# User specific environment and startup programs for Argon

export PATH=$PATH:$HOME/.local/bin:/Dedicated/IFC/.argon/bin

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Dedicated/IFC/.argon/lib

# Load module OpenMPI and HDF5
module load zlib/1.2.11_parallel_studio-2017.1
module load hdf5/1.8.18_parallel_studio-2017.1
module load openmpi/2.0.1_parallel_studio-2017.1
```

These load OpenMPI version 1.8.8 for use with the Intel compiler as well as the HDF5 1.8.18 library. Instead of loading these modules manually, the commands can be added to the end of the file `.bash_profile` in the user's home directory. Note that Neon and Argon each have a seperate $HOME hence `.bash_profile` file. In addition, if using the Python interface functions on Argon, the appropriate Python module must be loaded. This can be done with a call to:

```
module load python27
```

This can also be added to the `.bash_profile` file to automate the loading process.

10.5.2 Installing the package on ARGON

First, `git clone` the repository or `tar xf` a release package.

Then run the classic GNU build tool chain:

```
mkdir build && cd build
../configure --prefix=/Dedicated/IFC/.argon CFLAGS="-O3 -march=core-avx2 -DNDEBUG"␣
↪PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/Dedicated/IFC/.argon/lib/pkgconfig
make
make check
make install
```

10.6 Updating the package

Whenever the `autoconf` or `automake` files are modified, the build system needs to be update:

```
# Using 'make dist' with a 32 UID
export TAR_OPTIONS="--owner=0 --group=0 --numeric-owner"

autoreconf --install
mkdir build && cd build
make dist
```

10.7 Standard Makefile Targets

- `make all` Build programs, libraries, documentation, etc. (Same as `make`.)
- `make install` Install what needs to be installed.
- `make install-strip` Same as `make install`, then strip debugging symbols.
- `make uninstall` The opposite of `make install`.
- `make clean` Erase what has been built (the opposite of `make all`).
- `make distclean` Additionally erase anything `./configure` created.
- `make check` Run the test suite, if any.
- `make installcheck` Check the installed programs or libraries, if supported.
- `make dist` Create PACKAGE-VERSION.tar.gz.

CHAPTER 11

Custom Models

An extendible collection of models is built-in ASYNCH. The evaluation of the differential and algebraic equations occurs in the source code *problems.c*, while the definition of the models (i.e. number of parameters, precalculations, etc.) is set in *definetype.c*. New models can be added here by modifying those two source files, plus adding needed function declarations to *problems.h*.

Every built in model is given a unique id known as the *model type*. This nonnegative integer value is used to identify the model throughout the initialization process. The model type is specified in the global file used to initialize ASYNCH. User defined models are possible, which can be created outside ASYNCH's built-in collection of models.

11.1 Model Definition

The definition of every model is given in *definetype.c*. This module consists of five routines used to initialize each model. A description of the contents of each of these routines is given below.

11.1.1 SetParamSizes

This routine defines several integer values needed to describe a model.

Name	Description
dim	The number of states modeled by the differential and algebraic equations. This parameter is also the number of such equations at a single link.
template_flag	A flag to specify if the model uses an XML parser for evaluating the differential equations. 0 indicates no parser, 1 indicates a parser is used.
assim_flag	A flag to specify if the model uses a data assimilation scheme. 0 indicates no data assimilation, 1 indicates data assimilation is used. This feature will be removed in future versions.
diff_start	The index in the equation-value vectors where the differential equations begin. All equations before this index are assumed to be algebraic. If all equations are differential, then this value should be 0.
no_ini_start	The index in the state vectors corresponding to the first state not requiring initial conditions specified by the initial states specification in a global file. These states are generally initialized by other parameters of the model in the function *InitRoutines*. If all states require initialization through the global file (typical), then *no_ini_start* should be set to dim.
num_global_params	The number of global parameters specified in a global file for this model. If the number provided by the global file is less than expected, an error occurs. If more parameters are given than expected, a warning is given. The extra parameters are accessible. Although providing more parameters than needed is not recommended, it can be useful for testing.
uses_dam	A flag to indicate if the model uses dams. 0 indicates no, 1 indicates yes. If dams are not available for this model, a value of 0 is expected for the *dam flag* in the global file.
params_size	The total number of local parameters available at each link. This includes all parameters read from the link parameters of the global file as well as all precalculations (specified in the *Precalculations*).
iparams_size	The number of integer valued parameters at each link. This may be removed in future versions.
dam_params_size	The number of additional parameters at links with a dam. These parameters are included at the end of the vector of parameters at each link with a dam.
area_idx	The index in the parameter vector of the upstream area parameter. This para eter is used frequently with peakflow data.
areah_idx	The index in the parameter vector of the hillslope area. This parameter is frequently used with peakflow data.
disk_params	The number of local parameters available at each link read from a parameter file or database table. The *params_size* minus the *disk_params* is the number of recalculated parameters plus any dam parameters at the link.
num_dense	The number of states passed down from one link to another. This number cannot be larger than dim. If equal to 0, then the links are totally disconnected.
convertarea_flag	Flag to indicate whether the model converts the parameter with index area_idx from km^2 to m^2. This can be needed for peakflow output data. The flag is set to 1 if the units are converted, 0 if not.
num_forcings	The number of forcings for the model. If a global file specifies less than this number of forcings, an error occurs. If more than this number of forcings is specified, a warning is given.
dense_indices	An array containing the indices in the state vectors that are passed from one link to another. This array must contain *num_dense* indices.

11.1.2 ConvertParams

This routine allows for unit conversions on the local parameters at each link. These conversions occur immediately after the parameters are loaded into memory, and thus will be in place for all calculations. This feature is useful for when a data source provides values with units different than those expected by the model.

11.1.3 InitRoutines

This routine specifies routines associated with the model. In this routine, the following arguments are available.

Name	Description
link	The current link where the routines are to be set.
type	The model index.
exp_imp	A flag to determine if an implicit or explicit RK method is to be used. 0 if the method is explicit, 1 if it is implicit.
dam	A flag for whether a dam is present at this link. 0 if no dam is present, 1 if a dam is present.

The following routines must be set at each link.

Name	Description
RK-Solver	The routine for the numerical integrator.
f	The routine to evaluate the differential equations of the model.
alg	The routine to evaluate the algebraic equations of the model.
Jacobian	The routine to evaluate the Jacobian of the system of differential equations. This must be set if an implicit RK method is used.
state_check	The routine to check in what discontinuity state the system is. The number of the discontinuity state is determined by the model.
Check-Consistency	This routine alters the state vectors to be consistent with constraints of the system. **Notice: these constraints MUST exist in the exact solution of the equations for the link (for example, nonnegative solutions to a linear system).**

11.1.4 Precalculations

This routine allows computations that are static in time and independent of state to be performed. The results are stored with the link parameters. This feature can be used to prevent redundant computations. The following information is accessible in this routine:

Name	Description
link_i	The current link where precalculations are performed.
global_params	The parameters which are constant in space and time.
params	The parameters for this link. Results from this routine will be stored in this vector. Other parameters from a database or parameter file (.prm) are accessible here.
iparams	The integer parameters for this link.
disk_params	The first entry of params that should be set for this location.
params_size	The first entry for dam parameters. These are only accessible if the dam flag is set.
dam	The flag to indicate if a dam is present at the current link. If dam is 1, then a dam model is present here. If dam is 0, then a dam model is not present.
type	The index of the model.

Before exiting, all entries in params from index *disk_params* up to (but not including) *params_size* should be set.

11.1.5 ReadInitData

This routine sets any initial conditions which are *not* determined through the *Initial Conditions* section of the global file (.gbl) (Section *Initial States*). Generally, this is to set the initial conditions for unknowns in models determined by algebraic equations, or those ODEs which have hardcoded initial conditions. The *ReadInitData* routine sets the initial conditions link by link. The following information is available in this routine:

Name	Description
global_params	The parameters which are constant in space and time.
params	The parameters for this link.
iparams	Integral parameters for this link.
qvs	Discharge vs storage table. This information is available only if a dam is present at this link.
dam	The flag to indicate if a dam is present at the current link. If *dam* is 1, then a dam model is present here. If *dam* is 0, then a dam model is not present.
y_0	The vector of initial values. All indices between *diff_start* (inclusive) and *no_ini_start* (exclusive) are set. These values were determined from the initial conditions specified in the global file. Both *diff_start* and *no_ini_start* are defined by the routine *SetParamSizes*.
type	The index of the model.

The return value *ReadInitData* is the discontinuity state of the system, based upon the initial value vector *y_0*.

11.2 Model Equations Definition

The equations for the model are defined in the file *problems.c*. Each set of built-in equations requires a routine to be defined here. Further, the differential and algebraic equations for a model must be defined in separate routines (although the routine for the differential equations may call the function for the algebraic equations). As is typical in C, any routines created in *problems.c* should be declared in *problems.h*. The routines defined here should be attached to each model in the *InitRoutines* method in *definetype.c*.

11.2.1 Differential Equations

Every model must have a set of differential equations. The equations defined in this routine are for a single link only. Mathematically, the form of these equations should appear as

$$\begin{aligned}\frac{dy_s}{dt} &= f_s(...)\\ \frac{dy_{s+1}}{dt} &= f_{s+1}(...)\\ &\vdots\\ \frac{dy_{dim}}{dt} &= f_{dim}(...)\end{aligned}$$

where s is *diff_start*. Note that the index of the first state determined by a differential equation is *diff_start* (or s here). Thus, these states should appear after any states determined through algebraic equations in state and equation-value vectors. When the differential equation routine is called, the rate of change of each of the state variables y_i is the expected output. Thus, this routine should evaluate all of the functions on the right of the equations. Examples of differential equations used for ASYNCH can be found in Section *Built-in Models*.

Warning: doxygentypedef: Cannot find typedef "DifferentialFunc" in doxygen xml output for project "api" from directory: .doxygen/api/

It is worth noting that only states from the upstream links are available in this routine. Dependence upon further upstream links breaks the underlying tree structure.

11.2.2 Algebraic Equations

Some models may have a set of algebraic equations. The equations defined in this routine are for a single link only. Mathematically, the form of these equations should appear as

$$\begin{aligned} y_0 &= g_0(...) \\ y_1 &= g_1(...) \\ &\vdots \\ y_{s-1} &= g_{s-1}(...) \end{aligned}$$

where s is *diff_start*. Note that the index of the first state determined by an algebraic equation is 0. Thus, these states should appear before any states determined through differential equations in state and equation-value vectors. When this routine is called, the expected output is the evaluation of the right side function. Support for algebraic equations is limited to explicit equations of the state variables. This means none of the states $y_0, \ldots, y_{s-1}$ are available for use in this routine. Only the states defined through differential equations are available ($y_s, \ldots, y_{dim}$). Examples of models with algebraic equations can be found in Section *Built-in Models*.

Warning: doxygentypedef: Cannot find typedef "AlgebraicFunc" in doxygen xml output for project "api" from directory: .doxygen/api/

It is worth noting that only states from this link are available in this routine.

11.2.3 State Check

Some models may include discontinuities in the states of the system. This routine determines in which discontinuity state the system currently is. The return value is the integer representing the current discontinuity state.

Warning: doxygentypedef: Cannot find typedef "CheckStateFunc" in doxygen xml output for project "api" from directory: .doxygen/api/

11.2.4 System Consistency

For many models, the equations describing the differential and algebraic system states come with built-in constraints. Common examples include non-negative values or maximum state values. These constraints may not necessarily be satisfied due to numerical errors. A routine for system consistency is called by the integrator to guarantee these constraints are satisfied.

Warning: doxygentypedef: Cannot find typedef "CheckConsistencyFunc" in doxygen xml output for project "api" from directory: .doxygen/api/

Note: The solutions to the algebraic and differential equations MUST support these constraints. For instance, an equation with an exponential decaying solution has a minimum value for the solution. However, such an equation has no limit on the maximum value of its solution. Thus, a consistency routine can be created to impose the minimum value, but not a maximum value.

The values of states derived through algebraic equations are not available in the consistency routine. This is done for efficiency, as the algebraic states may not be needed to check consistency.

CHAPTER 12

C API

ASYNCH comes with a collection of routines to access specific details of the underlying solver structure. This allows a user to construct his or her own programs, while making calls to the solver in particular ways. This is useful for modifying how the solvers behave, and for producing more specialized outputs. A user can also specify his or her own models in a separate module. Currently, interface routines exist for the C/C++ and Python programming languages.

12.1 Typical Interface Usage

In this section, we provide an overview of how to use the API routines for running simulations. Regardless of the language, the general order of calling API routines is the same. Further routines may be added to increase flexibility.

A basic program that uses the ASYNCH solvers to perform calculations and takes advantage of all the features of a global file is the following (written in C):

```
//Make sure we finalize MPI
void asynch_onexit(void)
{
    int flag;
    MPI_Finalized(&flag);
    if (!flag)
        MPI_Finalize();
}

int main(int argc,char* argv[])
{
  int res;
  char *global_filename = argv[1];

  //Initialize MPI stuff
  res = MPI_Init(&argc, &argv);
  if (res == MPI_SUCCESS)
      atexit(asynch_onexit);
  else
```

(continues on next page)

(continued from previous page)

```
  {
      print_err("Failed to initialize MPI");
      exit(EXIT_FAILURE);
  }

  //Init asynch object and the river network
  AsynchSolver asynch;
  Asynch_Init(&asynch, MPI_COMM_WORLD);
  Asynch_Parse_GBL(&asynch, global_filename);
  Asynch_Load_Network(&asynch);
  Asynch_Partition_Network(&asynch);
  Asynch_Load_Network_Parameters(&asynch, 0);
  Asynch_Load_Dams(&asynch);
  Asynch_Load_Numerical_Error_Data(&asynch);
  Asynch_Initialize_Model(&asynch);
  Asynch_Load_Initial_Conditions(&asynch);
  Asynch_Load_Forcings(&asynch);
  Asynch_Load_Save_Lists(&asynch);
  Asynch_Finalize_Network(&asynch);
  Asynch_Calculate_Step_Sizes(&asynch);

  //Prepare output files
  Asynch_Prepare_Temp_Files(&asynch);
  Asynch_Write_Current_Step(&asynch);
  Asynch_Prepare_Peakflow_Output(&asynch);
  Asynch_Prepare_Output(&asynch);

  //Perform the calculations
  Asynch_Advance(&asynch, 1);

  //Create output files
  Asynch_Take_System_Snapshot(&asynch, NULL);
  Asynch_Create_Output(&asynch, NULL);
  Asynch_Create_Peakflows_Output(&asynch);

  //Clean up
  Asynch_Delete_Temporary_Files(&asynch);
  Asynch_Free(&asynch);

  return EXIT_SUCCESS;
}
```

Details of each function call can be found in Section [sec: user interface routines].

The program begins by initializing the asynchsolver object. The MPI communicator consisting of all processes (`MPI_COMM_WORLD`) is used for the calculations. Next, the global file specified as a command line argument to the program is parsed. Based upon the information specified by the global file, the different components of the network are constructed.

Next, the outputs for the program are initialized. The files for holding calculation results while the program runs is initialized (i.e. the temporary files). The initial values of the states are written to disk. Any initializations needed for the output peakflow and output time series sources are done.

With a call to `Asynch_Advance`, the calculations are performed. Any time series results are temporarily stored in the temporary files.

Once the calculations are complete, any necessary outputs are created (snapshot, output time series, output peakflow data). This could include writing to files or database tables, depending upon the options selected in the global file.

Lastly, clean up routines are called. The temporary files are deleted. The asynchsolver object is also deleted from memory with a call to . Note that this routine does not need to be called if using the interface functions from a language that supports automatic garbage collection.

Of course, outputting more information might be useful (timing results, command line parameter checking, results printed to screen, etc), and additional features may be needed (outputting data to multiple sources, creating peakflow data over intervals of time, etc). However, this is the basic structure needed to perform simulations. The source files *asynchdist.c* and *asynchdist.py* are essentially duplicates of the above program, but with information printed to screen.

12.2 User Interface Routines

In this section, routines for operating the solver are described. These routines can be used to create an instance of an ASYNCH solver, and manipulate properties such as total simulation time, when data output occurs, etc… Creation of custom outputs is discussed in Section [sec: custom outputs] and creation of custom models is discussed in Section [sec: custom-models].

12.2.1 Solver Constructor / Destructor

AsynchSolver ***Asynch_Init** (MPI_Comm *comm*, bool *verbose*)

This routine initializes an instance of an AsynchSolver.

Multiple solvers could be created if multiple problems are to be solved. An instance of an AsynchSolver contains all the relevant data structures and information to solve the equations for an underlying model. An AsynchSolver object should be destroyed with a call to *Asynch_Free*.

Pre MPI should be properly initialized with MPI_Init before calling Asynch_Init.

Return A pointer to an AsynchSolver object.

See Asynch_Free

Parameters

- `comm`: The MPI communicator to use with this solver object.

void **Asynch_Free** (*AsynchSolver* **asynch*)

This routine deallocates the memory occupied by an AsynchSolver object created with a call to Asynch_Init.

See Asynch_Init

Parameters

- `asynch`: A pointer to a AsynchSolver object to free.

12.2.2 Solver Initialization

void **Asynch_Parse_GBL** (*AsynchSolver* **asynch*, char **filename*)

This routine opens and processes a global file.

It reads all specified database connection files, but does not process any other input file. An error in this routine is considered fatal, and results in a call to the routine *MPI_Abort* on the communicator used to create asynch.

Pre asynch should be initialized with Asynch_Init before calling Asynch_Parse_GBL.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `filename`: Path of a global file.

void **Asynch_Load_Network** (*AsynchSolver *asynch*)

This routine processes topology inputs for the AsynchSolver object as set in the global file read by Asynch_Parse_GBL.

This initializes each Link object and sets their parent and child information. Generally, this is the first initialization routine to call after parsing a global.

Pre Asynch_Parse_GBL should be called before Asynch_Load_Network.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Partition_Network** (*AsynchSolver *asynch*)

This routine assigns the Links of the *asynchsolver* object to the MPI processes.

Pre This routine must be called after *Asynch_Load_Network*.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Network_Parameters** (*AsynchSolver *asynch*)

This routine processes parameter inputs for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Setting *load_all* to true causes every MPI process to store the parameters at every Link.

Pre This routine can be called before *Asynch_Partition_Network* only if *load_all* is set to true.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Dams** (*AsynchSolver *asynch*)

This routine processes the is_dam inputs for the AsynchSolver object as set in the global file read by.

Pre This routine should be called after *Asynch_Partition_Network* and *Asynch_Load_Network_Parameters* have been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Numerical_Error_Data** (*AsynchSolver *asynch*)

This routine processes numerical solver inputs for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Pre This routine should be called after *Asynch_Partition_Network* has been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Initialize_Model** (*AsynchSolver *asynch*)
This routine sets the model specific routines for each link for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Pre This routine should be called after *Asynch_Partition_Network* and *Asynch_Load_Network_Parameters* have been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Initial_Conditions** (*AsynchSolver *asynch*)
This routine processes the initial condition inputs for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Pre This routine should be called after *Asynch_Partition_Network* and *Asynch_Initialize_Model* have been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Forcings** (*AsynchSolver *asynch*)
This routine processes the forcing inputs for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Pre This routine should be called after *Asynch_Partition_Parameters* has been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Load_Save_Lists** (*AsynchSolver *asynch*)
This routine processes save list inputs for the AsynchSolver object as set in the global file read by *Asynch_Parse_GBL*.

Pre This routine should be called after *Asynch_Partition_Network* has been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Finalize_Network** (*AsynchSolver *asynch*)
This routine checks that all inputs are loaded for the AsynchSolver object.

Some small final initializations are also performed.

Pre This routine should be called as the last initialization routine.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

12.2.3 Integration

void **Asynch_Calculate_Step_Sizes** (*AsynchSolver *asynch*)
This routine processes calculates appropriate step sizes for the integrators at each link in the AsynchSolver object.

Pre This routine must be called before a call to *Asynch_Advance*, and after all initializations are performed.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **Asynch_Advance** (*AsynchSolver* *asynch, int *print_level*)

This routine advances the numerical solver up to the time set in *maxtime*.

See Section [sec: model type and maxtime]. Calculations to solve the model differential and algebraic equations are performed, using forcing data as needed.

Pre This routine should be called after *Asynch_Partition_Network* has been called.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `print_level`: If 0, no time series information is produced. Otherwise, time series information is produced.

12.2.4 Timeseries Output

void **Asynch_Prepare_Output** (*AsynchSolver* *asynch)

This routine prepares the output sources for time series data.

Preparation includes creating files or database tables.

Pre This routine must be called before any time series data can be produced.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

int **Asynch_Create_Output** (*AsynchSolver* *asynch, char *file_suffix*)

This routine takes all data written to temporary files and moves them to a final output destination for time series data.

If the output format is data file or CSV file, then the string *additional_out* is appended to the filename. If *additional_out* is *NULL*, then no appending to the filename occurs.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `file_suffix`: String appended to the end of any output filename. \ return Returns 0 if output is written, -1 means there is no data to output.

void **Asynch_Prepare_Peakflow_Output** (*AsynchSolver* *asynch)

This routine prepares the output sources for the peakflow data.

Preparation includes creating files or database tables.

Pre This routine must be called before any peakflow data can be produced.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

int **Asynch_Create_Peakflows_Output** (*AsynchSolver* *asynch)

This routine takes all calculated peakflow data and writes them to a final output destination.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use. \ return Returns 0 if output is written, -1 means there is no data to output.

int **Asynch_Write_Current_Step** (*AsynchSolver* *asynch)

This routine writes the current state of each link to temporary files for every link where a time series output has been specified.

Normally, calling *Asynch_Advance* with the *print_flag* set is enough to write output time series. However, advancing an AsynchSolver object without the *print_flag* set or calls that modify how data is outputted to the temporary files will cause data to not be written for some times. This routine can be called to commit missing data.

Return Returns 0 if the step is written, 1 if no temporary file is available.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

12.2.5 Temporary files

void **Asynch_Prepare_Temp_Files** (*AsynchSolver* *asynch)

This routine prepares the temporary files for time series data.

Preparation includes creating files or database tables.

Pre This routine must be called before any time series data can be calculated. A call to *Asynch_Advance* with the *print_flag* set before preparing temporary files will create an error.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

int **Asynch_Delete_Temporary_Files** (*AsynchSolver* *asynch)

This routine deletes the tempfiles for *asynch*.

This is useful for cleaning up temporary files at the end of a simulation, or for deleting the files if they must be reconstructed.

Return Returns 0 if the tempfiles were deleted, 1 if no tempfiles exist, and 2 if an error occurred.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

int **Asynch_Set_Temp_Files** (*AsynchSolver* *asynch, double *set_time*, void **set_value*, unsigned int *output_idx*)

This routine moves the temporary file pointer to a previous value.

All future values are deleted. The point where the pointer is moved is determined by *set_value*, which is in the output series determined by *output_idx*. The local time where the next data will be written for each link is *set_time*.

Pre A warning occurs if *set_value* is not found for a link, and that link's tempfile pointer is not changed.

Return Returns 0 if the temporary files are set, 1 if a warning occurred, 2 if there was an error setting the files.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `set_time`: The local time corresponding the step where the temp files will be set.
- `set_value`: The value to which the tempfiles will be set.
- `output_idx`: The index of the series output of which *set_value* corresponds.

12.2.6 Snapshots

int **`Asynch_Take_System_Snapshot`** (*AsynchSolver* **asynch*, char **prefix*)

This routine processes calculates appropriate step sizes for the integrators at each link in the AsynchSolver object.

This routine creates snapshot output data to either a recovery file or a database table.

The current value of every state at every link is outputted. If a recovery file was the format specified in the global file, then the string *name* is appended to the end of the recovery filename. This appending does not occur if *name* is *NULL* or if a database table is the selected format.

Pre This routine must be called before a call to *Asynch_Advance*, and after all initializations are performed.

Return An error code. Returns 0 if a snapshot was made, 1 if an error was encountered, -1 if no snapshot is made.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `prefix`: String to prepend to the snapshot output filename.

Return Returns 0 if a snapshot was made, 1 if an error was encountered, -1 if no snapshot is made.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `prefix`: String to prepend to the snapshot output filename.

int **`Asynch_Get_Snapshot_Output_Name`** (*AsynchSolver* **asynch*, char **filename*)

This routine gets the filename of the output snapshot file.

If a database connection is used, then the contents of *snapshotname* is not modified. The Python interface routine returns the string with the filename instead of taking it as an argument.

Return 0 if the filename was set successfully. 1 otherwise.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `filename`: Name of the snapshot output filename returned.

int **`Asynch_Set_Snapshot_Output_Name`** (*AsynchSolver* **asynch*, char **filename*)

This routine sets the filename of the output snapshot file.

If a database connection is used, then no changes are made.

Return 0 if the filename was set successfully. 1 otherwise.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

- `filename`: Name to set the snapshot output filename.

12.2.7 Getters and Setters

unsigned short **`Asynch_Get_Model_Type`** (*AsynchSolver* **asynch*)
This routine sets the model type.

Return The model type

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **`Asynch_Set_Model_Type`** (*AsynchSolver* **asynch*, unsigned short *type*)
This routine returns the model type.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `type`: The model type.

double **`Asynch_Get_Total_Simulation_Duration`** (*AsynchSolver* **asynch*)
This routine returns the value of *duration*, as defined in Section[sec:simulation period].

Return The current duration of the simulation time of the AsynchSolver object (unit is model dependant, but usually minutes is used).

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

void **`Asynch_Set_Total_Simulation_Duration`** (*AsynchSolver* **asynch*, double *duration*)
This routine returns the value of *duration*, as defined in Section[sec:model type and duration].

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `duration`: The value to set the maximum simulation time.

unsigned short **`Asynch_Get_Num_Links`** (*AsynchSolver* **asynch*)
This routine returns the total number of links in the network of *asynch*.

Return Number of links in the network

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

Link ***`Asynch_Get_Links`** (*AsynchSolver* **asynch*)

unsigned int **`Asynch_Get_Num_Links_Proc`** (*AsynchSolver* **asynch*)
This routine returns the total number of links assigned to the current MPI process in the network of *asynch*.

This number represents the total number of links whose differential and algebraic equations are solved by the current process.

Return Number of links assigned to the current MPI process

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

Link ***Asynch_Get_Links_Proc**(*AsynchSolver* **asynch*)

12.2.8 Database

void **Asynch_Set_Database_Connection**(*AsynchSolver* **asynch*, **const** char **conn_string*, unsigned int *conn_idx*)

This routine sets a new database for an input or output.

If information for a database has already been set, it is released, and the new connection information is set. Database information includes hostname, username, password, etc. This is the same information that is available in the header of database connection files (see Section [sec: database connection files]). Several database connections exist for every AsynchSolver object. *conn_idx* can take the values:

- ASYNCH_DB_LOC_TOPO
- ASYNCH_DB_LOC_PARAMS
- ASYNCH_DB_LOC_INIT
- ASYNCH_DB_LOC_RSV
- ASYNCH_DB_LOC_HYDROSAVE
- ASYNCH_DB_LOC_PEAKSAVE
- ASYNCH_DB_LOC_HYDRO_OUTPUT
- ASYNCH_DB_LOC_PEAK_OUTPUT
- ASYNCH_DB_LOC_SNAPSHOT_OUTPUT
- ASYNCH_DB_LOC_FORCING_START

The last value for *conn_idx* is the database connection for the first forcing specified in the global file. Database connections for other forcings can be access sequentially. For example, to access the forcing with index 2 (the third forcing in a global file), set *conn_idx* as `ASYNCH_DB_LOC_FORCING_START + 2`

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `conn_string`: A connection string
- `conn_idx`: Connection index

12.2.9 Forcing

int **Asynch_Activate_Forcing**(*AsynchSolver* **asynch*, unsigned int *idx*)

This routine activates a forcing for use.This means when a call to the routine *Asynch_Advance* is made, the forcing with index *idx* will be applied for calculations.By default, all forcings set in a global file are initially active.This routine has the opposite effect of *Asynch_Deactivate_Forcing*.

Return Returns 0 if the forcing was activated, 1 if an error occurred

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.

- `idx`: The index of the forcing to activate.

int **Asynch_Deactivate_Forcing**(*AsynchSolver* *asynch, unsigned int *idx*)

This routine deactivates a forcing for use.

This means when a call to the routine *Asynch_Advance* is made, all values for the forcing with index *idx* will be taken as 0. By default, all forcings set in a global file are initially active. This routine has the opposite effect of *Asynch_Activate_Forcing*.

Return Returns 0 if the forcing was activated, 1 if an error occurred

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `idx`: The index of the forcing to activate.

unsigned int **Asynch_Get_First_Forcing_Timestamp**(*AsynchSolver* *asynch, unsigned int *forcing_idx*)

This routine returns the first timestamp for a forcing.

Pre This can only be used if the forcing with index *forcing_idx* is using a format of binary files, gz binary files, or database table. See Section [sec: forcing inputs] for adescription of these formats.

Return The timestamp of the last timestamp for the forcing with index *forcing_idx*.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `forcing_idx`: An index of a forcing.

void **Asynch_Set_First_Forcing_Timestamp**(*AsynchSolver* *asynch, unsigned int *unix_time*, unsigned int *forcing_idx*)

This routine sets the first timestamp for a forcing.

Pre This can only be used if the forcing with index *forcing_idx* is using a format of binary files, gz binary files, or database table. See Section [sec: forcing inputs] for adescription of these formats.

Return The timestamp of the last timestamp for the forcing with index *forcing_idx*.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `unix_time`: The value to set the first forcing timestamp.
- `forcing_idx`: An index of a forcing.

unsigned int **Asynch_Get_Last_Forcing_Timestamp**(*AsynchSolver* *asynch, unsigned int *forcing_idx*)

This routine returns the last timestamp for a forcing.

Pre This can only be used if the forcing with index *forcing_idx* is using a format of binary files, gz binary files, or database table. See Section [sec: forcing inputs] for adescription of these formats.

Return The timestamp of the last timestamp for the forcing with index *forcing_idx*.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `forcing_idx`: An index of a forcing.

void **Asynch_Set_Last_Forcing_Timestamp**(*AsynchSolver* **asynch*, unsigned int *unix_time*, unsigned int *forcing_idx*)

This routine sets the last timestamp for a forcing.

Pre This can only be used if the forcing with index *forcing_idx* is using a format of binary files, gz binary files, or database table. See Section [sec: forcing inputs] for adescription of these formats.

Return The timestamp of the last timestamp for the forcing with index *forcing_idx*.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `unix_time`: The value to set the last forcing timestamp.
- `forcing_idx`: An index of a forcing.

void **Asynch_Set_Forcing_DB_Starttime**(*AsynchSolver* **asynch*, unsigned int *unix_time*, unsigned int *forcing_idx*)

This routine sets the start time used for a forcing.

This value is used for converting between timestamps in a database table and the local time of the solvers. This can only be used if the forcing with index *forcing_idx* is using a format of database table. See Section [sec: forcing inputs] for a description of this format.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `unix_time`: The value to set the start timestamp.
- `forcing_idx`: An index of a forcing.

12.2.10 State Variables Getters and Setters

void **Asynch_Set_Init_File**(*AsynchSolver* **asynch*, char **filename*)

This routine sets a file for reading initial value data.

The *init_flag* is set based upon the extension of *filename*. The initial data is **NOT** read while executing this routine. A call to *Asynch_Load_System* is needed to set the filename. This routine cannot be used to set the format to a database connection.

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `filename`: Filename to use for initial value data.

void **Asynch_Set_System_State**(*AsynchSolver* **asynch*, double *unix_time*, double **states*)

This routine sets the current time of each link to *t_0* and the current state to the vector in the array *values*.

The vectors are assumed to be in the same order as the links provided by the topology source specified in the global file (see Section [sec: topology]).

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `unix_time`: The timestamp to set at each link.
- `states`: The vector of the current state of each link.

12.2.11 State Variables Getters and Setters

12.2.12 Custom Outputs

Time series outputs can be customized so as to produce results specific to a particular model. For instance, fluxes that are used internally to the model, or values that are not needed at all for computing state solutions, may be outputted.

To create a custom output, specify the name of your new output in the global file (see Section [sec: time series output]). Next, in your program after reading the global file but before performing simulations, call the routine *Asynch_Set_Output* to set your output. Then perform any calculations or further modifications as usual.

A custom routine must be provided by the user for determining how the output is calculated. The specifications for this function are provided below.

Similarly, custom peakflow outputs can be created. These are set with a call to the routine *Asynch_Set_Peakflow_Output*.

The routines for setting custom outputs are described below.

int **Asynch_Set_Output_Int** (*AsynchSolver* **asynch*, char **name*, OutputIntCallback **callback*, unsigned int **used_states*, unsigned int *num_states*)

This routine sets a custom output time series.

A function is required that will be called every time output data is to be written for a link. The function set in this routine should have the specification Asynch_Set_Output_Int must also be given an array *used_states*. This array contains the indices of the states in the state vectors which are needed by the user specified routine *callback*. All states listed in this array are guaranteed to be available for the routine *callback*.

See Asynch_Set_Output_Double

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `name`: Name of the custom time series.
- `callback`: Routine to call when writing data for the custom output.
- `used_states`: Array of the all indices in the state vector used by *callback*.
- `num_states`: Numver of indiciced in used_states.

int **Asynch_Set_Output_Double** (*AsynchSolver* **asynch*, char **name*, OutputDoubleCallback **callback*, unsigned int **used_states*, unsigned int *num_states*)

This routine sets a custom output time series.

A function is required that will be called every time output data is to be written for a link. The function set in this routine should have the specification Asynch_Set_Output_Double must also be given an array *used_states*. This array contains the indices of the states in the state vectors which are needed by the user specified routine *callback*. All states listed in this array are guaranteed to be available for the routine *callback*.

See Asynch_Set_Output_Int

Parameters

- `asynch`: A pointer to a AsynchSolver object to use.
- `name`: Name of the custom time series.
- `callback`: Routine to call when writing data for the custom output.
- `used_states`: Array of the all indices in the state vector used by *callback*.

- `num_states`: Numver of indiciced in used_states.

CHAPTER 13

Python API

Most of the previous sections about the user interface routines applies to the Python binding. Naturally, some specifics concerning Python come into play.

The module `asynch_interface` for Python can be found in the directory *asynch_py* in the ASYNCH root directory. The module can be loaded a in Python script with a command such as:

```
from asynch_py.asynch_interface import asynchsolver
```

The routines in the Python interface can be found in Section [sec: user interface routines]. These routines are predominantly the same as those from the C interface, with the exception of the initialization and finalization routines (Sections [sec: solver initialization] and [sec: asynch$_f$ree], respectively).

To demonstrate the usage of these routines in Python, consider the following code, which is analogous to the program given in Section [sec: interface usage]:

```
#Include the ASYNCH Python interface
from asynch_py.asynch_interface import asynchsolver
import sys

#Prepare system
asynch = asynchsolver()
asynch.Parse_GBL(sys.argv[1])
asynch.Load_System()

#Prepare outputs
asynch.Prepare_Temp_Files()
asynch.Write_Current_Step()
asynch.Prepare_Peakflow_Output()
asynch.Prepare_Output()

#Advance solver
asynch.Advance(1)

#Take a snapshot
```

(continues on next page)

(continued from previous page)

```
asynch.Take_System_Snapshot(None)

#Create output files
asynch.Create_Output(None)
asynch.Create_Peakflows_Output()

#Cleanup
asynch.Delete_Temporary_Files()
```

The contents of the source code *asynchdist.py* is essentially the program above. The module `sys` is loaded to read the filename of the global file from the command line. An instance of the `asynchsolver` class is created and stored as `asynch`. From this object, all the routines from the interface are called. As is traditional in Python, clean up of the `asynchsolver` instance is performed automatically.

Creating custom models and outputs with the Python interface is similar to that of the C interface. See Sections [sec: custom outputs] and [sec: custom models]. Any routines passed into the Python interface written in Python must be decorated as an appropriate function type. The available data types are

Type	Description
ASYNCH_F_DATATYPE	ODE
ASYNCH_RKSOLVER_DATATYPE	Runge-Kutta Solver
ASYNCH_CONSISTENCY_DATATYPE	System Consistency
ASYNCH_ALG_DATATYPE	Algebraic Equations
ASYNCH_STATECHECK_DATATYPE	State Check
ASYNCH_OUTPUT_INT_DATATYPE	Output Time Series with Integer Values
ASYNCH_OUTPUT_DOUBLE_DATATYPE	Output Time Series with Double Precision Values
ASYNCH_PEAKOUTPUT_DATATYPE	Output Peakflow Routine
ASYNCH_SETPARAMSIZES_DATATYPE	SetParmSizes Routine for Custom Models
ASYNCH_CONVERT_DATATYPE	ConvertParams Routine for Custom Models
ASYNCH_ROUTINES_DATATYPE	InitRoutines Routine for Custom Models
ASYNCH_PRECALCULATIONS_DATATYPE	Precalculations Routine for Custom Models
ASYNCH_INITIALIZEEQS_DATATYPE	ReadInitData Routine for Custom Models

Decorating a routine in Python can be done with the `@` symbol. For instance, the following is the definition of a function for converting units of state parameters:

```
@ASYNCH_CONVERT_DATATYPE
def ConvertParams_MyModel(params,model_type,lib_ptr):
  params.contents.ve[1] *= 1000
  params.contents.ve[2] *= 1e6
```

A function decorated with an ASYNCH function type cannot be called as a regular Python function. It should only be called from a routine written in C.

The routines for creating a custom model have one additional argument available. This is a library object to the ASYNCH C routines. Such routines may be used by and given to Python interface functions. For instance, consider this example routine used for the `ReadInitData` routine:

```
@ASYNCH_ROUTINES_DATATYPE
def InitRoutines_MyModel(link_p, model_type, exp_imp,dam,lib_p):
  lib = cast(lib_p,py_object).value
  link = link_p.contents

  if link.res:
```

(continues on next page)

(continued from previous page)

```
    link.f = LinearHillslope_Reservoirs_MyModel
    link.RKSolver = cast(lib.ForcedSolutionSolver, ASYNCH_RKSOLVER_DATATYPE)
  else:
    link.f = LinearHillslope_MyModel
    link.RKSolver = cast(lib.ExplicitRKSolver, ASYNCH_RKSOLVER_DATATYPE)
    link.alg = cast(None,ASYNCH_ALG_DATATYPE)
    link.state_check = cast(None, ASYNCH_STATECHECK_DATATYPE)
    link.CheckConsistency = cast(lib.CheckConsistency_Nonzero_AllStates_q, ASYNCH_
↪CONSISTENCY_DATATYPE)
```

In this sample, the Runge-Kutta methods are set to functions defined in the ASYNCH library. Also note that if a routine does not need to be set (here, the algebraic and state check routines), then the routine is set to `None` casted as the appropriate function data type. The `SetParamSizes` routine for creating custom models requires creating an array. This can be done with a call to the function `Allocate_CUINT_Array` from the ASYNCH library. It requires only one argument, the size of the array, and its return value can be directly set to the member `dense_indices`.

CHAPTER 14

Internals

14.1 Data Structures

This section gives an overview of the data structure used in Asynch and mostly intended for the project maintainers.

`struct AsynchSolver`
This is the main structure that holds the state of the server and associated data structures for a simulation.

Public Members

MPI_Comm **`comm`**
COMM on which the solver works.

int **`np`**
Number of procs in the comm.

int **`my_rank`**
This processes rank in the comm (varies by proc)

bool **`verbose`**
Set to true is asynch.

AsynchModel ***`model`**
The object model.

ErrorData **`errors_tol`**
Object for global error data.

GlobalVars ***`globals`**
Global information.

Link ***`sys`**
Array of links in the network [dim].

unsigned int **`N`**
Number of links in sys.

unsigned int **num_methods**
Number of methods in rk_methods.

RKMethod ***rk_methods**
List of RK methods available.

TransData ***my_data**
Data for communication between procs.

short int ***getting**
List of data links to get information about.

int ***assignments**
Link with sys location i is assigned to proc assignments[i].

Link ****my_sys**
Array of pointer to links in the network assigned to this proc [dim].

unsigned int **my_N**
Number of links in sys assigned to this proc.

unsigned int **save_size**
Number of links in save_list.

unsigned int ***save_list**
List of link ids to print data.

unsigned int **my_save_size**
Number of links assigned to this proc in save_list.

Link ****my_save_link_list**
List of Links to print data.

unsigned int **peaksave_size**
Number of links in peaksave_list.

unsigned int ***peaksave_list**
List of link ids to print peakflow data.

unsigned int **my_peaksave_size**
Number of links assigned to this proc in peaksave_list.

Link ****my_peaksave_link_list**
List of Links to print peakflow data.

Lookup ***id_to_loc**
Lookup table to convert from ids to sys locations.

Workspace **workspace**
Temporary workspace.

char **rkdfilename**[**ASYNCH_MAX_PATH_LENGTH**]
Filename for .rkd file.

FILE ***outputfile**
File handle for outputing temporary data.

FILE ***peakfile**
File handle for the peakflow data.

ConnData **db_connections**[**ASYNCH_MAX_DB_CONNECTIONS**]
Database connection information.

Forcing **forcings**[**ASYNCH_MAX_DB_CONNECTIONS** - **ASYNCH_DB_LOC_FORCING_START**]
Forcing information.

struct GlobalVars
Structure to contain all data that is global to the river system.

Public Members

unsigned short **model_uid**
Index for the model used.

double **maxtime**
Integrate up to this time (duration) [minutes].

double **t_0**
Initial time to start integration.

double **t**
Current time of integration.

time_t **begin_time**
Unix begin time.

time_t **end_time**
Unix end time.

unsigned short **method**
RK method to use (if it is the same for all links)

unsigned int **max_localorder**
Max local order of implemented numerical methods.

unsigned short **max_rk_stages**
The largest number of internal stages of any RK method used !!!! Is this needed? !!!!

unsigned short **max_parents**
The largest number of parents any link has.

int **iter_limit**
If a link has >= iter_limit of steps stored, no new computations occur.

int **max_transfer_steps**
Maximum number of steps to communicate at once between processes.

unsigned int **discont_size**
Size of discont, discont_send, discont_order_send at each link.

unsigned short int **uses_dam**
1 if this type can use dams, 0 else

double ***global_params**
List of global parameters.

unsigned int **num_global_params**
Number of global parameters.

unsigned int **num_params**
Number of params at each link without a dam /unsigned int iparams_size; //!< Number of iparams at each link.

unsigned int **dam_params_size**
Number of params at each link with a dam.

unsigned int **num_disk_params**
Number of parameters to read from disk.

unsigned int **area_idx**
Index of upstream area (A_i) in params.

unsigned int **areah_idx**
Index of hillslope area (A_h) in params.

unsigned short int **init_flag**
0 if reading .ini file, 1 if reading .uini file, 2 if reading .rec file

unsigned short int **rvr_flag**
0 if reading .rvr file, 1 if using database

unsigned short int **prm_flag**
0 if reading .prm file, 1 if using database

double **print_time**
Each link will write state every print_time minutes. -1 uses a formula.

unsigned short int **print_par_flag**
0 to use specified name for output files, 1 to add parameters

unsigned short int **dam_flag**
0 if not using .dam file, 1 if using

unsigned short int **hydrosave_flag**
0 if not saving hydrographs, 1 if saving

unsigned short int **peaksave_flag**
0 if not saving peak flows, 1 if saving

char ***peakfilename**
Filename for .pea file.

unsigned int **max_dim**
Maximum num of degree of freedom in the system (assim uses variable dimensions)

unsigned int **outletlink**
For database: holds the link id of the outlet. Use 0 if reading entire database.

unsigned int **string_size**
Size of filename buffers.

unsigned int **query_size**
Size of database query buffers.

unsigned short int **convertarea_flag**
1 if hillslope and upstream areas are converted from km^2 to m^2, 0 if not

double **discont_tol**
The error tolerance to use for locating discontinuities.

unsigned int **min_error_tolerances**
The minimum number of error tolerances needed at every link. Used for uniform error tolerances.

double **dump_time**
Each link states will dump every dump_time minutes.

unsigned int **init_timestamp**
The timestamp of the initial state (only used to get initial condition from the DB)

unsigned int **num_states_for_printing**
Number of states used for printing.

unsigned int ***print_indices**
List of indices in solution vectors where data is written to output [num_states_for_printing].

unsigned int **num_outputs**
Number of outputs.

struct ConnData
Structure to hold information about an PostgreSQL database.

Public Members

PGconn ***conn**
Connection to a database.

char **connectinfo**[**ASYNCH_MAX_CONNSTRING_LENGTH**]
Connection information for a database.

char **query**[**ASYNCH_MAX_QUERY_LENGTH**]
Buffer space for making queries.

unsigned int **time_offset**
Added to the integration time to get the actual unix time.

struct ErrorData
Holds the error estimation information for a link.

See Hairer, E. and Norsett, S.P. and Wanner, G., Solving Ordinary Differential Equations I, Nonstiff Problems.

Public Members

double **facmax**
Parameter for error estimation.

double **facmin**
Parameter for error estimation.

double **fac**
Parameter for error estimation.

double ***abstol**
Absolute tolerance [num_dof].

double ***reltol**
Relative tolerance [num_dof].

double ***abstol_dense**
Absolute tolerance for dense output [num_dof].

double ***reltol_dense**
Relative tolerance for dense output [num_dof].

struct Forcing
This structure holds all the data for a forcing in the river system.

Public Members

unsigned int **raindb_start_time**
: This is the unix time corresponding to the local intergrator time 0.

unsigned int **good_timestamp**
: A timestamp in the db where a forcing actually exists.

unsigned int **next_timestamp**
: Holds the next timestep to use for pulling data.

unsigned int **lastused_first_file**
: The value of first_file when the GetPasses routine was last called. 0 if never set.

unsigned int **lastused_last_file**
: The value of last_file when the GetPasses routine was last called. 0 if never set.

unsigned int **number_timesteps**
: The number of times which feature a forcing at some link.

CHAPTER 15

Contributing to Asynch

You are here to help on Asynch? Awesome, feel welcome and read the following sections in order to know what and how to work on something. If you get stuck at any point you can create a ticket on GitHub.

15.1 Contributing to development

If you want to deep dive and help out with development on Read the Docs, then first get the project installed locally according to the *Installation* instruction. After that is done we suggest you have a look at tickets in our issue tracker that are labelled Good First Bug. These are meant to be a great way to get a smooth start and won't put you in front of the most complex parts of the system.

If you are up to more challenging tasks with a bigger scope, then there are a set of tickets with a Enhancement tag. These tickets have a general overview and description of the work required to finish. If you want to start somewhere, this would be a good place to start. That said, these aren't necessarily the easiest tickets. They are simply things that are explained. If you still didn't find something to work on, search for the Sprintable label. Those tickets are meant to be standalone and can be worked on ad-hoc.

When contributing code, then please follow the standard Contribution Guidelines set forth at contribution-guide.org.

15.2 Keeping the documentation updated

Whenever an update in the code adds, changes or removes elements that affect the user experience (i.e.: changes in input forcing formats, existing hlm models, global file format, etc), it is expected from the developer to perform the respective updates in the *Read the Docs* documentation.

Here are some tips and explanations regarding the documentation process.

15.2.1 Documentation structure

The documentation publicly available at `http://asynch.readthedocs.io` is hosted by Read the Docs in a project that is maintained by the same maintainers of Asynch.

This documentation is written in `reStructuredText` formatting language (*.rst files) and uses Sphinx with Read the Docs libraries for compilation purposes. All the files are relevant for the documentation are located in the `doc/` folder.

15.2.2 Working locally

A desktop environment can be set up to compile the documentation locally. This approach is good for debugging the result before publishing it publicly.

For setting up the environment, you will have to install Read the Docs on your machine in a separated virtual Python environment.

Ensure **git 1.5** or higher, **Python 3.6** or higher, and both Python **virtualenv** and **virtualenvwrapper** are installed. If using Windows, ensure Python **virtualenvwrapper-win** and cmder are also installed (use **cmder** to perform the following command operations).

Create a virtual environment for compiling Read the Docs documentations:

```
mkvirtualenv readthedocs
```

If your command session does not activate the newly created virtual environment automatically, activate it:

```
workon readthedocs
```

Navigate to this virtualenv directory:

```
cd [USER_HOME]\Env\readthedocs\
```

Checkout Read the Docs into a new folder and enter there:

```
mkdir checkouts
cd checkouts
git clone https://github.com/rtfd/readthedocs.org.git
cd readthedocs.org\
```

Install all requirements for Read The Docs:

```
pip install -r requirements.txt
```

or:

```
python -m pip install -r requirements.txt
```

Navigate to the `docs` directory of the local clone of Asynch repository:

```
cd [ASYNCH]\docs\
```

Perform the changes you want in the `.srt` files within this folder. After that, compile using the command:

```
make html
```

Note: 1-) When compiling, ensure you are still working on the readthedocs virtual Python environment;

2-) When compiling, some Python packages may be required. Be prepared to perform pip installs;

3-) The command 'make' also works on Windows when runned within cmder.

Access the results opening the file `[ASYNCH]/docs/.build/html/index.html` with a web browser.

This mini tutorial was adapted from here.

15.2.3 Publishing

Everytime a git `push` or `pull request` is performed into the `master` or `develop` branches in the Git Hub repository, or on a branch that creates a new `tag`, the Read the Docs server reads, compiles and publishes the documentation online.

This connection between Git Hub and Read the Docs is established through the so called *webhooks*. The official Asynch Git Hub account has a webhook that triggers the compiling steps from the Read the Docs server.

15.3 Managing releases

Once you are happy with your changes in the `develop` branch and ran a couple of test simulations, here is the procedure to release a new version `x.y.z` (e.g. `1.5.0`):

15.3.1 Branch

Create a branch for the release following the `release-x.y.z` naming scheme and semantic versionning rules :

```
git branch release-x.y.z
```

15.3.2 Edit

Edit the release notes (`doc/release_notes.rst`).

Edit `configure.ac` to bump the version number:

```
AC_INIT([asynch], [x.y.z], [samuel-debionne@uiowa.edu])
```

Commit your changes.

```
git add configure.ac doc/release_notes.rst
git commit -m "Bump version number to x.y.z"
git push
```

15.3.3 Generate the tarball

In a new empty folder, run the following commands to clone the repository, generate the configure script and the tarball.

```
git clone https://github.com/Iowa-Flood-Center/asynch.git
git checkout release-x.y.z
autoreconf -i
mkdir build && cd build
export TAR_OPTIONS="--owner=0 --group=0 --numeric-owner"
../configure
make dist
```

That should generate a `release-x.y.z.tar.gz` that needs to be tested.

15.3.4 Test the tarball

In a new empty folder, follow the instructions in *Installing the package*:

```
tar xf release-x.y.z.tar.gz
cd release-x.y.z
mkdir build && cd build
../configure CFLAGS="-O2 -DNDEBUG"
make
make check
make install
```

Adjust the release branch if there is any problem with the build (e.g. missing header file).

15.3.5 Release on Github

Merge the release branch `release-x.y.z` to `master`. The easiest way is to submit a new Pull Request. The *base* branch should be `Iowa-Flood-Center/asynch` / `master` and the *compare* branch `Iowa-Flood-Center/asynch` / `release-x.y.z`.

Review your Pull Request, or better let someone else do the review. If everything looks good, and you have Travis CI's blessing, do a *"Merge and Squash"*.

You can safely delete the release branch at this point.

Click on *"Draft a new release"* in Releases:

Field	Value
Tag version	vx.y.z (v1.5.0)
Release title	Pick a city in Iowa
Description	A short version of the release notes

Attach the tarball that was generated in the previous step. This is usefull because the tarball does not require the target computer to have autotools installed.

Ready? *"Publish Release"*! Every followers of the repo get notified of the new version. Good job!

Bibliography

[da] Maelle Nodet, “Introduction to Data Assimilation”, Université Grenoble Alpes, Mars 2012

[fs] 19. Lakshmivarahan and J. M. Lewis, “Forward Sensitivity Approach to Dynamic Data Assimilation”, *Advances in Meteorology*, vol. 2010, Article ID 375615, 12 pages, 2010. doi:10.1155/2010/375615

[grad] Sengupta, B., K.J. Friston, and W.D. Penny. “Efficient Gradient Computation for Dynamical Models.” *Neuroimage* 98.100 (2014): 521–527. PMC. Web. 10 July 2017.

Index

A

C

E

F

G