General Outline of Workflow for Modelling Exercise for Scientific Computing Practical

Following the production of datasets for the year 2014 and 2015 for the HUC catchment 13010001 (Rio Grande headwaters in Colorado, USA), a modelling exercise was carried out to produce a simple snow melt model, using the 2015 dataset. The objective of the modelling exercise was to built a hydrological model (using one of the datasets), and validate its performance (using the other year).

The exercise was executed in sequence as follows:-

1. Building the model using features of the dataset (temperature, stream discharge & mean snow cover), in mathematical notation (using the notes provided in Chapter_8_Practical_Part_2 by Professor Lewis).
2. Translating the mathematical model into python code (wrapped into a function), using the example in the notes provided in Chapter_8_Practical_Part_2 by Professor Lewis.
3. Model calibration using the 2014 dataset:-

```
 a. Identify the values for the following model parameters:-
     i. max temperature
     ii. base level flow

 b. Select initial values for the following model parameters:-
     i. threshold temperature when melting takes place
     ii. network response function decay factor

 The value of max temperature will influence the value of scaling term k use
d in the model.

 c. Optimise the parameters max temperature & m, reporting their final value
s and the associated uncertianty.

 d. Reporting the goodness of fit between 2015 river discharge data and the
 hydrological model prediction
```

4. Model validation using the 2015 dataset - quantifying the goodness of fit between the 2015 river discharge data and the hydrological model prediction

Import of Modules for Practical

The code below imports the modules required to produce the images and run the functions written & used in this practical

In [49]:
```
# Import of required modules for practical
import numpy as np
import matplotlib.pylab as plt
%matplotlib inline
import scipy
import scipy.ndimage.filters
import pandas as pd
from scipy.optimize import minimize
```

Part 1: Introduction

1.1: Site Introduction

The site of interest where the hydrological model is built is the Del Norte monitoring station, which is located in Del Norte, Colorado. Del Norte is located at 37°40′44″N 106°21′11″W, where the Rio Grande river leaves the San Juan mountains and enters the San Luis Valley. The climate in the region is temperate, with temperature typically ranging from -14.6 to 25.9 degrees celcius. Precipitation in the region is relatively low throughout the year, ranging from 8.4 to 37mm. The hydrology of the reigion is mainly driven by snow-fall, with a mean annual snowfall of 101cm.

1.2: Reasoning and Application of Modelling Exercise

Sites such as Del Norte are easy to model, hydrological speaking, as the input of the system is mainly dominated by snow-melt. These system are however, very sensitive to changes in temperature. Such changes can cause changes in the timing and amount of snow-melt entering the river system, which in turn can affect aspects of the river such as:-

1. The bank stability and occurance of flood events
2. The dynamic of the ecological system in the river (affecting spawn timing and habitat suitability)
3. Changes in flow pattern and possible river pathways

These effects can take place both at the immediate site of interest (Del Norte monitoring station) or further downstream in the Rio Grande (reflecting the effects of upstream system to downstream systems).

The purpose of constructing the hydrological model was to predict the river discharge at the Del Norte monitoring station, given information on the daily mean temperature and mean snow cover at the HUC catchment 13010001. With a successful model, possible explorable scenarios include:-

1. Climate change scenarios, where seasonal temperature ranges changes.

Given that the catchment's hydrology is a snow-melt driven system, changes in temperature would have a great influence on the pattern and perhaps the amount of stream discharge observed at the site. Observing how flow discharge changes under future climate scenarios would allow for the development of mitigation strategies to cope with these changes. Possible future challenges could include:- a. Water security issues b. Bank stability issues c. Control of stream flow discharge

1. Exploration of past climates, where temperature and mean snow cover data is available but flow discharge is not present

Having built a model driven by snow-melt, which in turn, is driven by temperature, allows for the exploration of historic stream flow where temperature and mean snow cover are present, but river discharge observations are not.

Part 2: Method and Associated Code

2.1: Deciding on the Selection of Year for Model Calibration and Validation

To recap, both datasets produced in part 1 of this practical contained the following information:-

1. mean snow cover (0.0 to 1.0) for the HUC catchment 13010001 for each day of the year
2. temperature (in degrees Celcius) at the Del Norte monitoring station for each day of the year
3. river discharge at the Del Norte monitoring station for each day of the year

As seen in figure 1, the stream flow discharge trend appears similar for both years. Small differences can be noted which include:-

1. Differences in late spring/early summer stream flow surges (multiple peaks for the 2014 dataset, and a single peak for the 2015 dataset)
2. Difference in autumn stream flow surge (a more noticable peak for the 2014 dataset, compared to the much smaller peak for the 2015 dataset)

These differences are likely a result of the differences in daily temperature variations between the years. Given the clearer autumn stream flow surge observed in 2014, this dataset will be to construct and calibrate the hydrological model. The 2015 dataset will be used to validate the hydrological model.

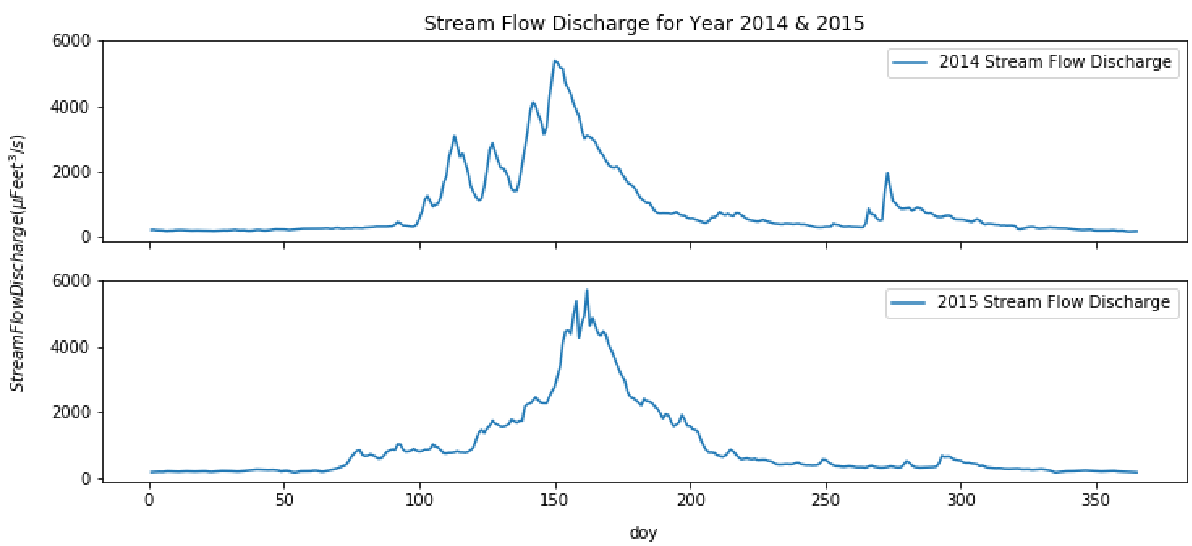

Figure 1: Stream Discharge Trend for Del Norte monitoring station for 2014 & 2015

2.2: The Hydrological Model

The purpose of constructing the hydrological model was to predict the river discharge at the Del Norte monitoring station, given information on the daily mean temperature and mean snow cover at the HUC catchment 13010001. This would allow for exploration of scenarios outside of the range of seasonal temperature change, which would affect the amount of snow cover and the flow discharge accordingly. Given the fact that catchment site was a snow dominated hydological system, a snow melt model was built as the hydrological model.

2.2.1: Building the Model

The following details on the model have been adapted from Chapter_8_Practical_Part_2, written by Professor Lewis.

Defining the Amount of Water in the Snow Pack (SWE)

Given that the site is a snow-melt dominated catchment, the stream flow measured at the Del Norte monitoring station stems from water released from snow pack. The units of this stream flow is described as volume per unit time.

The amount of water held in the snow pack can termed as the Snow Water Equivalent (SWE), which is a measure of water volume. The value of SWE can be estimated using:-

1. The snow area

The snow area is the catchment area (A) multiplied by the mean snow cover (p), which can be written as:

$$SA = Ap <--equation-1$$

1. The snow equivalent depth (d)

The dataset prepared in part 1 of the practical doesn't provide any information on the snow equivalent depth. To circumvent this issue, we can assume that d varies in the same way as snow cover, and can be described as follows:-

$$d = \frac{k}{A}p <--equation-2$$

where k is some constant, A is the catchment area and p is the mean snow cover.

With SA and d defined, the SWE can be described as:-

$$SWE = A.p.\frac{k}{A}.p <--equation-3$$

which can be simplified as:-

$$SWE = kp^2 <--equation-4$$

Describing the Amount of Water Released during Snow Pack Melt

The code below loads the dataset (stored in a npz file) and produces a plot of the temperature, snow cover & stream flow trend for the year 2014.

In [50]:
```
# load the data
file = np.load('dataset_scientific_computing_practical_part.npz')

# extract the data for 2014 and 2015 respectively
data_2014 = file['2014'].tolist() # use of .tolist() to extract the data insid
e
data_2015 = file['2015'].tolist() # converting from a numpy array to a list, s
o can access keys and data inside

# extract the temperature data
temp_2014 = data_2014['temperature']
temp_2015 = data_2015['temperature']

# extract the river discharge data
discharge_2014 = data_2014['river_discharge']
discharge_2015 = data_2015['river_discharge']

# extract the mean snow cover data
snow_2014 = data_2014['snow_cover']
snow_2015 = data_2015['snow_cover']

# extract the doy data
doy_2014 = data_2014['doy']
doy_2015 = data_2015['doy']
```

In [51]:
```
# the code below was modified from the code written by Professor Lewis in Chap
ter_8_Practical_Part_2
# to produce an image of all 3 data for 1 of the year datasets
# additional comments and modification were made as found appropriate

# producing image plot of dataset for year 2014 only
# setting up the plot
plt.figure(figsize=(12,5))
plt.xlim(doy_2014[0]-1, doy_2014[-1]) # correction of start date as should sta
rt from doy 0, not doy 1
plt.xlabel('day of year')

# plotting the data
# the names of the plotted variables and their labels were modified as appropr
iate
plt.plot(doy_2014, temp_2014, 'r', label='Temperature ($^\circ$C)') # plotting
 temperature
plt.plot(doy_2014, snow_2014*100.0, 'b', label='Snow Cover (%)')
plt.plot(doy_2014, 100.0 - snow_2014*100.0, 'c', label='Snow Free Cover (%)')
plt.plot(doy_2014, discharge_2014/100.0, 'g', label='Stream Discharge ( \mu Fe
et^3/s)')
plt.legend(loc='best')
plt.title('Temperature, Snow Cover, & Discharge Trend for 2014 Dataset')
plt.text(50, -40, 'Figure 2: Temperature, Snow Cover & Stream Discharge Trend
 in 2014 for Del Norte monitoring station'\
        , fontsize=12, weight='bold')
```

Out[51]:
```
Text(50,-40,'Figure 2: Temperature, Snow Cover & Stream Discharge Trend in 20
14 for Del Norte monitoring station')
```

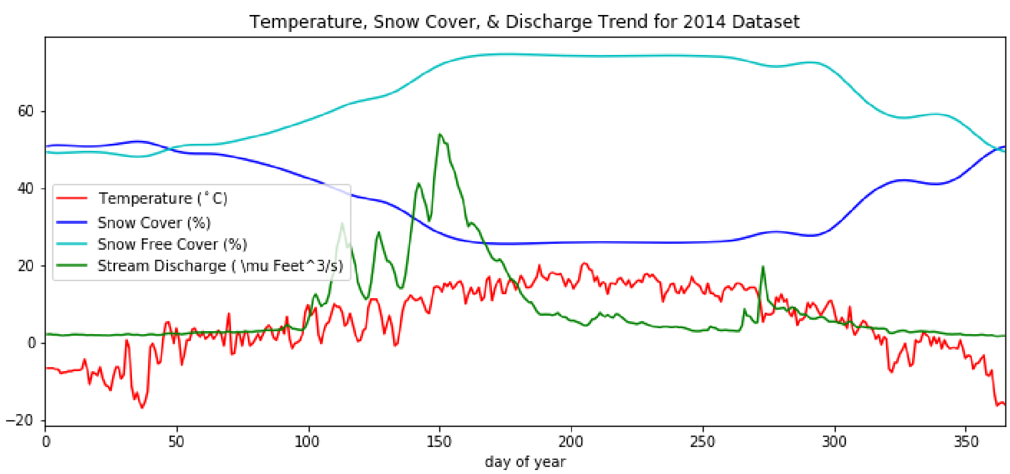

Figure 2: Temperature, Snow Cover & Stream Discharge Trend in 2014 for Del Norte monitoring station

As seen in figure 2, the timing of the stream discharge at Rio Grande roughly corresponds to:-

1. The changes in snow cover (p >0)
2. Temperature increases above a certain threshold (T > T_{thesh})

To begin with, T_{thresh} will be set to 0 degrees celcius, but will be later optimized to reduce the residual sum of squares for the model.

The amount of water entering the Del Norte system, given that it is a snow-melt dominated system, is a proportion (k_p) of SWE on days when melting occurs (when T > T_{thesh}). This can be described as follows:-

$$SWE_{melt}(t) = k_p SWE(t) <--equation-5$$

A possible solution to solve for k_p is to make it proportionate to the excess temperature:-

$$k_p(T) = \frac{T - T_{thresh}}{T_{max}} <--equation-6$$

where T_{max} is the maximum temperature recorded at the site. k_p is constrained between values of 0 to 1, where all negative values are set to 0, where no melting, hence no SWE is released. If negative k_p values are not constrained in this manner, this implies that water is added to SWE (hence depositing more snow) from the Rio Grande, which is not possible. As k_p increases towards 1, more of the SWE is made available as melt water. The maximum k_p of 1 implies that all SWE is now available as melt water for the system.

Accounting for Base Flow in the System

Referring back to figure 2, a base level flow can be observed operating the system. This can be accounted for in the hydrological model as F_{base}. To calculate the amount of water contributed by the melting of the snow pack ($SWE_{melt}(t)$) at any given time, the following equation can be used:-

$$F_{non-base}(t) = F(t) - F_{base} <--equation-7$$

Here, $F_{non-base}(t)$ is used instead of $SWE_{melt}(t)$ to clarify that stream flow is being modelled by the hydrological model. $F_{non-base}(t)$ and $SWE_{melt}(t)$ are conceptually identical, just expressed in in different terms (flows vs. snow water).

From the 2014 dataset, F_{base} can be estimated from the mean flow value of January.

Amount of Water Entering the System

Equation 7 can be rearranged to describe the amount of water entering the Rio Grande:-

$$F_{model}(t) = F_{base} + F_{non-base} <--equation-8$$

Remembering that $F_{non-base}(t)$ and $SWE_{melt}(t)$ are equivalent to one another, equation-8 can be re-written to express the maximum total amount of water in the system at any one time, using equation 5, 6 and 4 respectively.

Substituting $F_{non-base}(t)$ with $SWE_{melt}(t)$, and using equation 5 to substitute for $F_{non-base}$:-

$$F_{model}(t) = F_{base} + k_p(t).\,SWE(t) <--equation-9$$

Applying equation 6 to substitute for $k_p(t)$:-

$$F_{model}(t) = F_{base} + \frac{T - T_{thresh}}{T_{max}}(t).\, SWE(t) <-- equation - 10$$

Applying equation 4 to substitute for SWE:-

$$F_{model}(t) = F_{base} + \frac{T - T_{thresh}}{T_{max}}(t).\, kp^2(t) <-- equation - 11$$

Equation 11 can be rewritten as:-

$$F_{model}(t) = F_{base} + k.\, MAX\left(0, \frac{T_{max} - T_{thresh}}{T_{max}}.\, p(t)^2\right) <-- equation - 11$$

Note that $MAX\left(0, \frac{T_{max}-T_{thresh}}{T_{max}}.\, p(t)^2\right)$ simply refers to the maximum amount of water available to be released from the snow pack at a specific instance in time, which depends on the temperature at the site. As before, if the T_{thresh} is not met, no melting will occur (hence the 0 term in equation 11). $\frac{T_{max}-T_{thresh}}{T_{max}}.\, p(t)^2$ refers to the maximum amount of water that can ever be released into the system (where $\frac{T_{max}-T_{thresh}}{T_{max}}$ equals to 1 if a complete melting of the snow pack occurs).

If it can be assumed that the total amount of flow predicted by the model equals the total amount of measured flow for 2014, the following equation can be written:-

$$\Sigma_t F_{model}(t) = \Sigma_t F(t) <-- equation - 12$$

As such, equation 11 can be rewritten as:-

$$\Sigma_t F(t) = \Sigma_t F_{base} + k.\, \Sigma_t MAX\left(0, \frac{T_{max} - T_{thresh}}{T_{max}}.\, p(t)^2\right) <-- equation - 11$$

From equation 11, the value for constant k can be inferred from the data using the equation below:-

$$k = \frac{\Sigma_t F(t) - F_{base}}{\Sigma_t MAX\left(0, \frac{T_{max}-T_{thresh}}{T_{max}}.\, p(t)^2\right)} <-- equation - 13$$

An initial version of the snow-melt driven hydrological model (based on equation 11) can be coded as in python as a function (as seen below). To build the model, 4 components of the models must be set. These include:-

1. Temperature threshold (T_{thresh})
2. Max temperature (T_{max})
3. Base flow (F_{base})
4. constant k

Of the 4 parameters, only T_{max}, F_{base} and k will serve as constances of the model, and will be calculated using the 2014 dataset. A function was written to calculate the values of these parameters (see below). This function is then called in the function for the hydrological model to construct the model.

Using the output data from hydrological model function, an graphical plot of model prediction vs data observation for the 2014 stream flow was produced to visualize the model performance (and if further improvements are needed). In this plot, a value of 6.0 was set for temperature_thresh as a starting point for the model (later to be changed after applying the appropriate optimization techniques).

In [52]:
```
# To apply the hydrological model (in the snow_melt_model_version_1 function),
 we need to calculate the input values
# for the model parameters. The function below calculates max_temperature, bas
e_flow, k. Note that to calculate the value
# of constant k, temperature_thresh is required. We will make an initial guess
 of temperature_thresh using the value
# used in the notes by Professor Lewis in Chapter8_Practical_Part2

def calculate_model_constant_parameters(data, temperature_thresh):
    '''
    This function calculates the values of the model parameters that serve as
 constances for all years modelled. These
    parameters include max_temperature, base_flow, and k. The value of these p
arameters should be calculated using only
    1 of the years dataset. Once calculated, they should be applied to the mod
el for both the calibration dataset
    and validation dataset.

    Note that to calculate the value for constant k, the input argument temper
ature_thresh must be defined by the user.
    Depending on the value used, this will affect the value of k calculated, w
hich may not be optimized for the model.
    Hence, to ensure an optimized value of k is returned, an optimized value f
or temperature_thresh must be passed.

    max_temperature and base_flow are calculated solely using the data argumen
t passed, and are not dependent on the
    argument temperature_thresh.

    The function uses code that has been adapted from notes in Chapter8_Practi
cal_Part2 written by Professor Lewis, with
    additional notes added to clarify the code.

    Parameters
    ----------
    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

    temperature_thresh: a floating point number
        A floating point number which defines the threshold value for snow pac
k melt to occur. If the daily temperature
        is below this value, no melting occurs, hence no snow-melt input into
 the hydrological system. Exceeding this threshold
        will cause melting to occur.

    Returns
    -------
    The calculated values of the constances of the hydrological model. These i
nclude:-
```

```
    1. max_temperature
    2. base_flow
    3. k
    '''
    # loading the contents of the dictionary data
    temperature = data['temperature']
    stream_flow = data['river_discharge']
    snow = data['snow_cover'] # essentially parameter p in the hydrological mo
del (refer to equation 11)

    # calculating the value of base_flow using the mean stream_flow of January
    base_flow = stream_flow[:31].mean() # adapted from code in Chapter8_Practi
cal_Part2 by Professor Lewis

    # calculate the maximum temperature from the dataset
    max_temperature = temperature.max() # adapted from code in Chapter8_Practi
cal_Part2 by Professor Lewis

    # calculate the value of constant k
    # referring to equation 13, requires the sum of the non-base flow (hence t
otal flow - base flow at each time step) (numerator)
    # and the maximum swe melt to be released at each time step (constrained b
etween 0 and 1) without the constant k (denominator)

    # note that the code below has been adapted and modified from the notes in
 Chapter8_Practical_Part2 by Professor Lewis
    # the variable names have been changed and additional notes have been adde
d

    # lets first calculate the numerator
    non_base_flow = stream_flow - base_flow # adapted from code in Chapter8_Pr
actical_Part2 by Professor Lewis

    # now lets calculate the numerator
    # note that here we're calculating the rate of snow melt at different temp
eratures, and constraining the melting
    # to only take place when a positive difference in temperature is observed
    melting_rate = (temperature - temperature_thresh)/max_temperature # not co
nstrained to only positive values
    contrained_melting_rate = np.max([np.zeros_like(melting_rate), melting_rat
e], axis=0) # to constrains to only positive

            # values (between 0 and 1)

    # note that the values of 2 arrays are compared against each other (a zero
-filled array of the shame shape and data type
    # as swe_melt_without_k) along the axis 0 (the rows), where the largest va
lue for each row in the arrays are retained

    # to complete the calculation of swe_melt_without_k (swe melt at each time
 step) without constant k, p^2 must be multipled
    # here, p is snow
    swe_melt_k = snow*snow*contrained_melting_rate # adapted from code in Chap
ter8_Practical_Part2
                                                                   # by Professor
 Lewis
```

```
    # with the numerator and denominator of equation 13 calculated, the value
 for k can be solved
    k = non_base_flow.sum()/swe_melt_k.sum() # adapted from code in Chapter8_P
ractical_Part2 by Professor Lewis

    # returning values for constant model parameters
    return max_temperature, base_flow, k
```

In [53]:
```
# Here we create an initial version of the snow-melt driven hydrological mode
l, wrapped into a function
# this function will serve as the basis for future modifications (if found nec
essary)
def snow_melt_model_version_1(model_parameter, data_modelled, data_calibration
=data_2014):
    '''
    Function to calculate the daily stream flow discharge of the HUC catchment
 13010001.

    This function is version 1 of the snow-melt driven hydrological model. The
 mode has been constructed and calibrated using
    the data from the 2014 dataset. The model has been adapted based off the n
otes and codes provided by Professor Lewis
    in Chapter8_Practical_Part_2, under subsection 8.2.

    Inside this function, the constant parameters required for the hydrologica
l model (max_temperature, base_flow & k)
    are calculated by calling a function written to calculate these parameter
s. These are then fed into the model, and a
    modelled daily stream flow is produced as an output.

    Parameters
    ----------
    model_parameter: a numpy array
        A numpy array which stores the value for the model parameter temperatu
re_thresh. temperature_thresh is a
        floating point number which defines the threshold value for snow pack
 melt to occur. If the daily temperature
        is below this value, no melting occurs, hence no snow-melt input into
 the hydrological system. Exceeding this threshold
        will cause melting to occur.

    data_calibration: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to calibrate the model, to define the value fo
r the constances max_temperature,
        base_flow and k. This argument has been set to data_2014 by default as
 the hydrological model being developed
        should be built and calibrated using the 2014 dataset.

    data_modelled: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
```

```
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the model. Th
e data required includes snow cover
        (accessible using key 'snow_cover') and temperature (accessible using
 key 'temperature'). Both the 2014 and
        2015 dataset used as this argument, depending if the function user wis
hes to calibrate or validate the model.

    Returns
    -------
    Predictions of daily stream flow discharge values for each doy for date_mo
delled year.
    '''
    # grab required parameter constants for constructing the hydrological mode
l
    max_temperature, base_flow, k = calculate_model_constant_parameters(data_c
alibration, model_parameter)

    # load data to be modelled
    snow = data_modelled['snow_cover']
    temperature = data_modelled['temperature']

    # set up the final parameter for the model
    # the code below is similar to that found in the calculate_model_constant_
parameters function above
    # and is based of the notes in Chapter8_Practical_Part2 by Professor Lewis
    # the explanation for the process can be found in the comments for calcula
te_model_constant_parameters function
    # where the code below was replicated from (under different variable name
s) to calculate the rate of melting at different
    # temperatures, with the rate being contrained between 0 and 1
    rate_snow_melt = (temperature - model_parameter)/max_temperature
    constrained_rate_snow_melt = np.max([np.zeros_like(rate_snow_melt), rate_s
now_melt], axis=0)
    swe_melt_k = snow*snow*constrained_rate_snow_melt

    # constructing the model
    model_flow = swe_melt_k*k + base_flow # adapted from code in Chapter8_Prac
tical_Part2 by Professor Lewis

    # returning modelled flow (output of model)
    return model_flow
```

In [54]:
```
# function to produce a visual plot of stream flow data modelled vs observatio
ns of stream flow

def stream_flow_plot(model_parameters, data_modelled, model_version, year, num
ber):
    '''
    Function to plot output from modelled stream flow against observed stream
 flow.

    The function uses sections of code written by Professor Lewis in Chapter8_
Practical_Part2 to produce a plot of
    modelled stream flow vs. observed stream flow. Additional comments have be
en added to make the code clearer, and
    variable names have been changed for code execution purposes.

    The function is able to handle different version of the hydrological mode
l, and plot their outputs. The function user
    is able to specify the version of the hydrological model they wish to use
 through the model_version argument.

    Paramaters
    ---------
    model_parameters: a numpy array
        A numpy array which stores the values for the model parameters tempera
ture_thresh and m. temperature_thresh is a
        floating point number which defines the threshold value for snow pack
 melt to occur. If the daily temperature
        is below this value, no melting occurs, hence no snow-melt input into
 the hydrological system. Exceeding this threshold
        will cause melting to occur.

        m is the decay parameter for the network response function, and is a f
loatoing point number. It controls how quickly
        water from snow pack melts enters and contributes to stream flow. For
 version 1 of the hydrological model, m is not
        used in the model, hence does not need to be set.

    data_modelled: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the hydrologi
cal model (called in the function as
        snow_melt_model_version_? (where ? indicated an integer should be inse
rted to suit the hydrological model version
        the user wishes to use)). The data required includes snow cover (acces
sible using key 'snow_cover') and temperature
        (accessible using key 'temperature'). Both the 2014 and 2015 dataset u
sed as this argument, depending if the function
        user wishes to calibrate or validate the model.
```

```
    model_version: an integer
        Specifies the version of the hydrological model the user wishes to use
 to generate stream flow data.

    year: an integer
        Specifies which dataset is being used to generate the modelled stream
 flow plot. Choose between 2014 and 2015

    number: an integer
        Specifies the figure number for the plot, for labelling the figure.

    Returns
    -------
    A figure consisting of 2 line plots (of different colors) illutrating the
 modelled and observed stream flow
    '''
    # generating the modelled stream flow using the hydrological model
    # checking version of hydrological model want to use
    if model_version == 1:
        model_flow = snow_melt_model_version_1(model_parameters, data_modelled
)
    elif model_version == 2:
        model_flow = snow_melt_model_version_2(model_parameters, data_modelled
)

    # defining x_variable for plot
    doy = data_modelled['doy']

    # set up the plot
    plt.figure(figsize=(12,5))
    plt.xlim(doy[0], doy[-1]) # correction of start date as should start from
 doy 0, not doy 1
    plt.xlabel('day of Year {}'.format(year))
    plt.ylabel('$Stream Flow Discharge ( \mu Feet^3/s)$')

    # plotting the data
    # note the value 6.0 has been passed as the temperature_thresh argument in
 the snow melt model function as a starting point
    # simply to visualize model (the value for temperature_thresh has yet to b
e optimized)
    plt.plot(doy, model_flow/100., 'r', label='modelled stream flow/100')
    plt.plot(doy, data_modelled['river_discharge']/100., 'b', label='observed
 stream flow/100')
    plt.legend(loc='best')
    plt.title('Modelled vs. Observed Stream Flow for Year {}'.format(year))
    plt.text(55, -13, 'Figure {}: Modelled vs. Observed Stream Flow for Year
{}'.format(number, year)\
            , fontsize=12, weight='bold')
```

In [55]:
```
# producing the plot of modelled stream flow vs. observed stream flow for 2014
 using version 1 of the hydrological model
# setting temperature_thresh as 6.0 in the model parameters
model_parameters = np.array([6.0])
stream_flow_plot(model_parameters, data_2014, 1, 2014, 3)
```

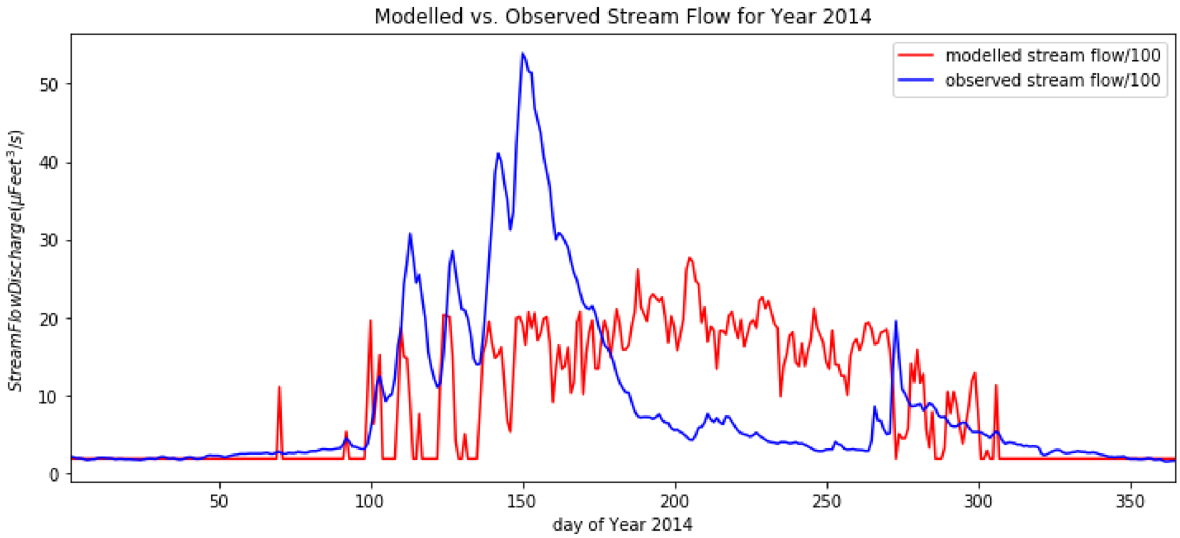

Figure 3: Modelled vs. Observed Stream Flow for Year 2014

As seen from the figure 3, the modelled stream flow does not exactly replicate the observed river flow in 2014. In particular, there are 2 issues with the modelled stream flow:-

1. Rapid oscillations are observed in the modelled stream flow. The observed river flow appears more smooth in comparison.
2. Sudden spikes in stream flow are observed in the modelled results, which stem from input from snow melt. In the observed river flow, this addition in water input takes place gradually, which helps explains the more smoother and more gradual patterns seen. As such, it can be said that there is a delay between snowmelt occuring and flow appearing in the measurements.

Accounting for Delay in Water Entering System from Snow Melt

A solution to accomodate for both issues is to introduce a network response function (nrf) to the model. The nrf can be described using the following equation:-

$$nrf = e^{-mt}$$

where e is an exponential, m is the decay parameter for the network responce function, and t is time step (!check!). The code below (authored by Professor Lewis, as seen in Chapter8_Practical_Part2) wraps the network responce function into a function usable for the hydrological model.

In [56]:
```
# the function below is a modified and annotated version of the network respon
ce function code authored by
# Professor Lewis in Chapter8_Practical_Part2

def network_response_function(m):
    '''
    This function provides a network response function for use in the hydrolog
ical model, to account for the delay in
    water entering the stream following snow pack melt and to smoothern out th
e modelled flow.

    The function is a modified and annotated version of the code written by Pr
ofessor Lewis in Chapter8_Practical_Part2

    Parameter
    ---------
    m: a floating point number
        The decay parameter for the network response function. Controls how qu
ickly water from snow pack melts enters
        and contributes to stream flow.

    Returns
    -------
    A network response function for use in the hydrological model
    '''
    # window size for the network response function
    ndays = 15*int(1/m)
    nrf_x = np.arange(ndays) - ndays/2

    # coding the network response function
    nrf = np.exp(-m*nrf_x)
    nrf[nrf_x<0] = 0 # similar reason to negative temperature difference? (che
ck)

    # normalizing output from network response functioon so that sum is equal
 to 1
    nrf = nrf/nrf.sum()

    return nrf
```

With the code above, we can now apply the network responce function to the hydrological model, to accomodate for the delay and produce a smoother modelled flow. This produces an updated version of the hydrological model, now termed snow_melt_model_version_2. Note the use of the value 0.03 as the m argument in the network_response_function. Similar to the temperature_thresh, this is simply a starting point for the value m, and is no means the final value for the model (have not carried out calibration and validation).

From snow_melt_model_version_2, we can produce a plot of modelled stream flow vs. observed stream flow for the year 2014.

In [57]:
```
# Here, an updated version of the hydrological model is produced, accounting f
or the delay in snow-melt affecting the stream flow

def snow_melt_model_version_2(model_parameters, data_modelled, data_calibratio
n=data_2014):
    '''
    Function to calculate the daily stream flow discharge of the HUC catchment
 13010001.

    This function is version 2 of the snow-melt driven hydrological model. The
 mode has been constructed and calibrated using
    the data from the 2014 dataset. The model has been adapted based off the n
otes and codes provided by Professor Lewis
    in Chapter8_Practical_Part_2, under subsection 8.2.

    Similar to the snow_melt_model_version_1, the constant parameters required
 for the hydrological model
    (max_temperature, base_flow & k)are calculated by calling a function writt
en to calculate these parameters.
    These are then fed into the model, and a modelled daily stream flow is pro
duced as an output.

    The modeleld daily stream flow is then treated by the network response fun
ction to account for the delay in water
    from snow-melt entering the stream. The ndimage.filters function from the
 scipy module was used to apply the network
    response function to the modelled daily stream flow.

    Parameters
    ----------
    model_parameters: a numpy array
        A numpy array which stores the values for the model parameters tempera
ture_thresh and m. temperature_thresh is a
        floating point number which defines the threshold value for snow pack
 melt to occur. If the daily temperature
        is below this value, no melting occurs, hence no snow-melt input into
 the hydrological system. Exceeding this threshold
        will cause melting to occur.

        m is the decay parameter for the network response function, and is a f
loatoing point number. It controls how quickly
        water from snow pack melts enters and contributes to stream flow.

    data_modelled: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the model. Th
e data required includes snow cover
        (accessible using key 'snow_cover') and temperature (accessible using
```

```
 key 'temperature'). Both the 2014 and
        2015 dataset used as this argument, depending if the function user wis
hes to calibrate or validate the model.

    data_calibration: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to calibrate the model, to define the value fo
r the constances max_temperature,
        base_flow and k. This argument has been set to data_2014 by default as
 the hydrological model being developed
        should be built and calibrated using the 2014 dataset.

    Returns
    -------
    Predictions of daily stream flow discharge values for each doy for date_mo
delled year.
    '''
    # get output from version 1 of the hydrological model
    model_flow = snow_melt_model_version_1(model_parameters[0], data_modelled)

    # get output from network responce function
    nrf = network_response_function(model_parameters[1])

    # convolve NRF with modelled flow data
    model_flow_nrf = scipy.ndimage.filters.convolve1d(model_flow, network_resp
onse_function(model_parameters[1]))

    # return output
    return model_flow_nrf
```

In [58]:
```
# producing the plot of modelled stream flow vs. observed stream flow for 2014
 using version 2 of the hydrological model
# setting up the model parameters, with temperature_thresh as 6.0 and m as 0.0
3
model_parameters = np.array([6.0, 0.03])
stream_flow_plot(model_parameters, data_2014, 2, 2014, 4)
```

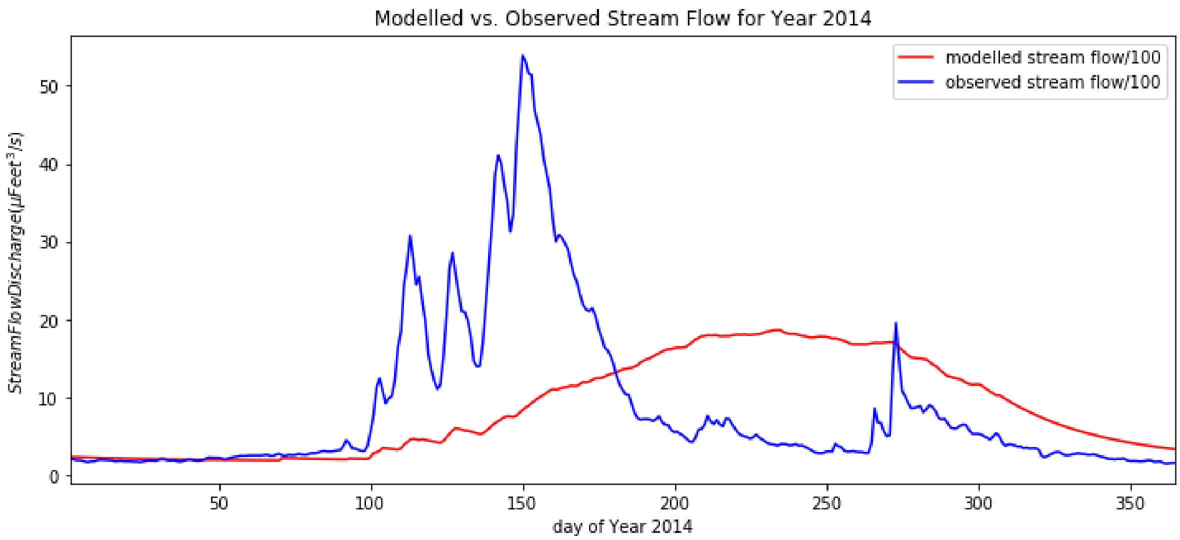

Figure 4: Modelled vs. Observed Stream Flow for Year 2014

The modelled stream flow in figure 4 is much smoother compared to that seen in figure 3. However, lots of the undulations seen in the observed stream flow are not absent, which is not what we want. To improve upon this, we have to optimize the model parameters T_{thresh} and m.

Optimization of Model Parameters

The next step in the modelling process is to optimize the following model parameters:-

1. temperature*thresh ($T*{thresh}$)
2. network response function decay factor (m)

This can be accomplished using a Monte Carlo sampling approach, which in essence, allows for the testing of plausible values for T_{thresh} and m (determined by the starting value assigned to them plus/minus a random value extracted from a random normal distribution) and calculating value of the cost function. The cost function function can be thought as measure of model performance, where a lower value signifies a better model performance (lower difference real observations vs. model output). As such, by selecting for values of T_{thresh} and m that minimize the cost function, the best version of the current model is produced, and the model can be said to be optimized.

The number of iterations done in a Monte Carlo approach is up to the user's discretion. The more iterations carried out, the more confidence can be had in the output value (but comes at the cost of computational time, hence a balance is needed). To implement the Monte Carlo approach, the Metropolis-Hastings algorithm can be applied. The notes below which details the workings of the algorithm have been adapted from Chapter7_FittingPhenologyModels, written by Professor Lewis/Dr. Jose Gonzales.

```
The Metropolis-Hastings is a sequential method that proposed & accepts sample value
s of a model's parameters based on the likelihood value. If the cost function impro
ves (hence drops), the sample values gets accepted. If the sample value doesn't imp
rove, a uniform random number between 0 and 1 is drawn. If the ratio of the propose
d to likelihood value is greater than the random number, the sample value gets acce
pted.

This allows for solutions that do not improve the cost function to still be used to
 explore other possible values for the model parameters. In essence, the plausible
 values for the model parameters do not get trapped in a local minima, this allowin
g for the exploration of the entire problem space.
```

Perhaps a good place to start with the Monte Carlo approach is to get 'good' rough guesses for the model parameters T_{thresh} and m. This will help give a more focussed scope of values for the Monte Carlo approach to explore, as well as serve as a good indicator for the size of the increment to be applied to each model parameter (when exploring the problem space). To do this, a brute force method can be applied that explores a plausible range of values for each parameter, as defined by the user. Here, we identify the best solution by looking at the goodness of fit in terms of the sum of squared residuals (squared difference between the model output and observations). The code below illustrates this:-

In [59]:
```
# lets create a function that will explore 400 plausible value pairs for tempe
rature_thresh and m
# note that the number of pairings can be changes by changing the default valu
e of no_iterations_temp and no_iterations_m
# we will stick to 400 values for the time being

def brute_force(starting_value_temp, ending_value_temp, \
                starting_value_m, ending_value_m, \
                no_iterations_temp=20, no_iterations_m=20, data=data_2014):
    '''
    Function that implements a brute force approach to sampling plausible valu
es for the model parameters temperature_thresh
    and m, returning the value pairings and their associated sum of squared re
siduals. This function only explored a small
    fraction of plausible values for the model parameters, and as such should
 only be used as a rough guideline for estimating
    the value of the model parameters.

    The function is also very sensitive to the value ranges given for the mode
l parameters, and as such the function user should
    give some thought into plausible values to use.

    Given the courseness of the solutions given as the output, these values as
 arguments for the starting values for
    temperature_thresh and m to be explored more finely in the Metropolis-Hast
ing approach.

    Note that this function is based on the code prepared by Professor Lewis/D
r. Jose Gonzales in Chapter5_Linear_models,
    with additional comments and variable names added.

    Parameters
    ---------
    starting_value_temp: a floating point number
        Starting value for temperature_thresh that the user wishes to explore.

    ending_value_temp: a floating point number
        Ending value for temperature_thresh that the user wishes to explore.

    starting_value_m: a floating point number
        Starting value for m that user wishes to explore.

    ending_value_m: a floating point number
        Ending value for m that user wishes to explore.

    no_iterations_temp: an integer
        Number of values wish to explore for temperature_thresh

    no_iterations_m: an integer
        Number of values wish to explore for m

    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
```

```
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the model.

    Returns
    -------
    3 variables are returned:-
        1. minimum_pairing = a list
            A list which stores the best solutions for temperature_thresh (ind
ex 0) and m (index 1), and the sum of residual
            square (index 2)

        2. solution_plot = a contour plot
            A contour plot illustrating the location of the initial value used
 for temperature_thresh (6.0) and m (0.03)
            in the sections preceeding the parameter optimization approach, an
d the location of the optimized value for
            temperature_thresh and m following the brute force approach

        3. results_df = a pandas dataframe
            A pandas dataframe illustrating the improvement in model performan
ce based on sum of residuals square. The dataframe
            only stores information on the best guess for model parameters (fo
llowing the brute force method) and the intial rough
            guess for model parameters. Column 0 stores information on values
 for temperature_thresh. Column 1 stores information
            on values for m. Column 2 stores information on the sum of residua
ls squares for the model based on the values of
            temperature_thresh and m in each row.
    '''
    # Part 1: Find Optimal Solution using Brute Force Method
    # load in the data stored in the dictionary
    doy = data['doy']
    flow = data['river_discharge']
    snow = data['snow_cover']
    temperature = data['temperature']

    # create a 2D numpy array to store the sum of square (sos) for model param
eter pairing
    sos = np.zeros((no_iterations_temp, no_iterations_m)) # adapted from Chapt
er5_Linear_models by Professor Lewis

    # define range of values to explore for temperature_thresh and m
    temp_range = np.linspace(starting_value_temp, ending_value_temp, no_iterat
ions_temp)
    m_range = np.linspace(starting_value_m, ending_value_m, no_iterations_m)

    # exploring plausible value for model parameters as determined by user
    # adapted from Chapter5_Linear_models by Professor Lewis
    # looping over temperature_thresh
    for ii, temperature_thresh in enumerate(temp_range):
        # looping over m
        for jj, m in enumerate(m_range):
            # set up the model parameters
```

```
            model_parameters = np.array([temperature_thresh, m])

            # calculate the sum of the residuals squared  based on the values
 of temperature_thresh and m used
            residual = snow_melt_model_version_2(model_parameters, data) - flo
w
            sq_residual = residual*residual
            sum_of_residual = sq_residual.sum()

            # storing the sum_of_residuals into 2D array created
            sos[ii, jj] = sum_of_residual

    # find optimal temperature_thresh and m value that leaads to sos_min
    # use a mask to find values, where all elements in sos are false except mi
nimum value of sos
    # then use mask to multiply temperature_thresh and m respective, and selec
t unique value greater than 0
    # this method and code was adapted from Chapter5_Linear_models by Professo
r Lewis
    sos_mask = sos == sos.min()
    temperature_solutions = np.unique(temp_range[None, :]*sos_mask)
    temperature_opt_solution = temperature_solutions[temperature_solutions>0]
    m_solutions = np.unique(m_range[None, :]*sos_mask)
    m_opt_solution = m_solutions[m_solutions>0]

    # returning smallest pairing of temperature_thresh and m value that equals
 smallest sum of residual square
    minimum_pairing = [temperature_opt_solution, m_opt_solution, sos.min()]

    # Part 2: Generate a contour plot
    # code was adapted from Chapter5_Linear_models by Professor Lewis
    solution_plot = plt.figure(figsize=(12,3))

    # set up x and y axis
    x_axis = np.linspace(starting_value_temp, ending_value_temp, no_iterations
_temp)
    y_axis = np.linspace(starting_value_m, ending_value_m, no_iterations_m)

    # fill in contour plot
    plt.contour(x_axis, y_axis, sos, np.logspace(8, 10, 20), cmap=plt.cm.magma
_r)
    # np.logspace defines location of contour lines
    # note modification of range of logspace to accomodate for sum of residual
 square

    # plot initial rough guess
    plt.plot(6.0, 0.03, 'o', mfc='None', mec='g', label='Initial Rough Guess')

    # plot optimal guess
    plt.plot(temperature_opt_solution, m_opt_solution, 'o', mfc='None', mec=
'r', label='Brute Force Guess')
    plt.legend(loc='best')

    # additional plot labelling
    plt.xlabel('Possible Temperature Threshold values ($^\circ$C)')
    plt.ylabel('Possible decay factor value ($m$)')
    plt.title("Contour Plot of Brute Force 'Best Guess' vs. Initial Rough Gues
```

```
s for Model Parameters")
    plt.text(2, -0.3, "Figure 5: Contour Plot of Brute Force 'Best Guess' vs.
 Initial Rough Guess for Model Parameters"\
            , fontsize=12, weight='bold')

    # Part 3: Generating a dataframe
    # store information on brute force best guess vs. initial rough guess in a
 pandas dataframe
    # code adapted from DSM from https://stackoverflow.com/questions/16597265/
appending-to-an-empty-data-frame-in-pandas

    # initialize dataframe
    results_df = pd.DataFrame()

    # generate and store temperature values
    temp_data = pd.DataFrame({"temperature_thresh": [6.0, round(temperature_op
t_solution[0],2)]}) # rounding to 2 decimal places
    results_df = results_df.append(temp_data)

    # generate and store m values
    results_df['m'] = pd.Series([0.03, round(m_opt_solution[0],2)])

    # generate and store sum of residual square values
    # need to generate sum of residual square for intial rough guess
    rough_guess = np.array([6.0, 0.03]) # initial model parameter values used
    residual = snow_melt_model_version_2(rough_guess, data_2014) - flow
    sq_residual = residual*residual
    sum_of_residual = sq_residual.sum()

    # storing sum of residual square data into dataframe
    results_df['sum_of_residuals_square'] = pd.Series([sum_of_residual, sos.mi
n()])

    # return output from function
    return minimum_pairing, solution_plot, results_df
```

From the brute_force function, a contour plot can be produced which illustrates the location of the 'best guess' and rough guess solution within a cost function.

In [60]:
```
# implementing function to visualize improvement in model performance and prin
t out improvement in model performance
# here we've increased the number of model evaluations to 10000, essentially i
mplementing a much finer grid
refined_guess, solution_plot, results_df = brute_force(starting_value_temp =
1.0, \
                                                       ending_value_temp = 10.
0, \
                                                       starting_value_m = 0.01
, \
                                                       ending_value_m = 1, \
                                                       no_iterations_temp=100,
 \
                                                       no_iterations_m=100, \
                                                       data=data_2014)
```

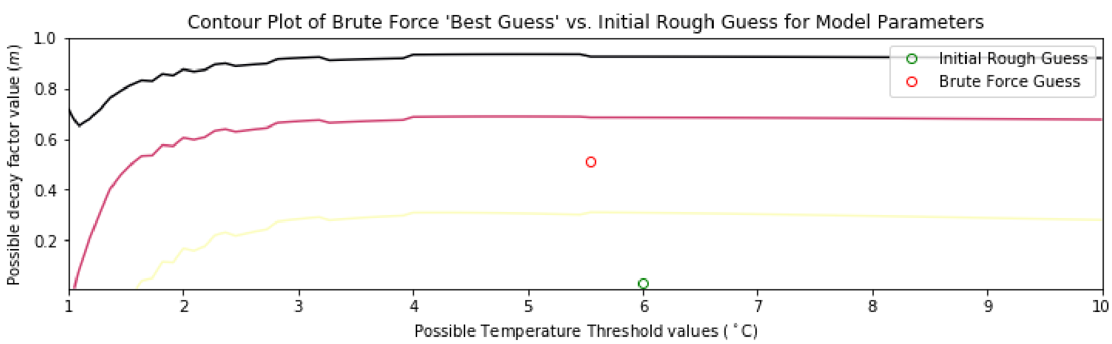

Figure 5: Contour Plot of Brute Force 'Best Guess' vs. Initial Rough Guess for Model Parameters

Due to the sum of residuals square of the model (even with the 'best guess' solution following the brute force method implementation) being quite very (on the order of 10^8), the contour plot doesn't reveal much revelent information to us. Instead, it would be better to visualize the results of the brute force estimates against the initial rough guess values in a table form.

In [61]:
```
# visualize improvement in model performance based on new 'best guess' of mode
l parameters in table form
# lets also give the table a title
# code adapted from Addison in https://stackoverflow.com/questions/8924173/how
-do-i-print-bold-text-in-python
print( '\033[1m' + 'Table 1: Results of Brute Force vs. Rough Guess Estimate o
f Model Parameters') # '\033[1m' bold the text
results_df
```

Table 1: Results of Brute Force vs. Rough Guess Estimate of Model Parameters

Out[61]:

	temperature_thresh	m	sum_of_residuals_square
0	6.00	0.03	5.076129e+08
1	5.55	0.51	3.098328e+08

As shown in table 1, the brute force method returns parameter estimates of T_{thresh} and m which give the model a lower sum of residual square. From the brute force estimates, we can generate a plot of modelled vs. observed flow for the 2014 dataset to get a visual representation of how the model performs.

In [62]:
```
# generate modeled vs. observed flow for the 2014 dataset using the brute forc
e parameter estimates
# extract brute force model parameter estimates
brute_guess = np.array([refined_guess[0][0], refined_guess[1][0]])
stream_flow_plot(brute_guess, data_2014, 2, 2014, 5)
```

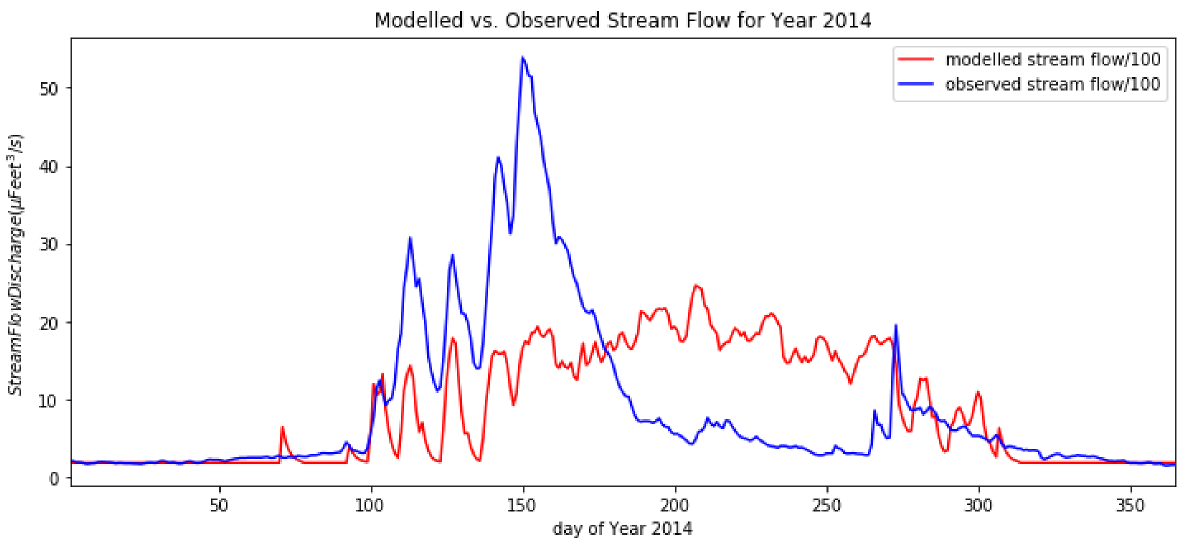

Figure 5: Modelled vs. Observed Stream Flow for Year 2014

As observed in figure 5, the modelled flow seems to more closely replicate the observed stream flow, with more undulations in stream flow observed (hence not as smooth) compared to the modelled stream flow plot in figure 4. These undulations are not as extreme as compared to that seen in figure 3, and is a sign that we're making progress.

Before moving on to the Metropolis-Hasting algorithm to fully optimize the model parameters, we could try conducting a synthetic experiment on the model dataset to see if the values generated from the brute force approach would be the best values moving forward in the Monte Carlo sampling approach.

Here in the synthetic experiment, we select only use the brute force values for the model parameters, and compare the modelled flow results with only 10% of the modelled flow (for the 2014 dataset) with noise added to this 10% of modelled output. The reasoning for this is to observe how the current model (and it's parameters) handles noise, and helps to determine if the current parameter values are suitable for moving forward to model validation work. To introduce noise to 10% of the modelled output, a random number generator (which draws numbers from a Gaussian distribution) is used. These numbers are then scalled by 0.6 (which signifies the modelled output having a standard deviation of 0.6), before being added to the 10% of modelled output.

From the synthetic experiment, we can generate a plot to see how the current model (and its parameter values) handles noisy data.

In [63]:
```
# generating modeled flow using the brute force parameter estimates
modelled_flow = snow_melt_model_version_2(brute_guess, data_2014)

# generating data for x axis for plotting
doy = np.arange(1,366) # same as data_2014['doy'] but without use of range fun
ction to generate data

# code below was adapted and modified from Chapter_6_NonLinear_Model_Fitting b
y Professor Lewis/Dr. Jose Gonzalez
# additional comments have been added and variable names had been changes to s
uit this practical
# adding noise to the modelled flow result
flow_with_noise = modelled_flow + np.random.rand(len(doy))*0.6 # need to gener
ate 365 random data with noise
                                                          # the scale by 0.6
 (standard error for noise)

# selecting 10% of model output randomly
selection = np.random.rand(len(doy)) # generate random values between 0 and 1
                                     # here produce as many random numbers as
 there are datapoint for each component
                                     # of the 2014 dataset

random_selection = np.where(selection > 0.9, True, False) # generate a boolean
 array, where only 10% of observation made available
                                                          # rest of observatio
ns made unavailable (switched off)
select_doy = doy[random_selection] # selecting subset of doy (x-axis plotting)
select_flow = flow_with_noise[random_selection] # selecting subset of modelled
 flow (y-axis plotting)

# plotting outcome from synthetic experiment
fig = plt.figure(figsize=(12,3))
plt.plot(doy, modelled_flow, '-', label='Modelled Flow') # plot modelled flow
plt.plot(select_doy, select_flow, 'o', label='Simulated Modelled Flow with Noi
se') # plot modelled flow with noise
plt.legend(loc='best') # set legend location
plt.xlabel('doy') # set x-axis label
plt.ylabel('$Stream Flow Discharge ( \mu Feet^3/s)$') # set y-axis label
plt.title('Modelled Flow & Simulated Modelled Flow with Noise Plot for 2014 Da
taset') # set plot title
plt.text(35, -800, 'Figure {}: Modelled Flow & Simulated Modelled Flow with No
ise for Year {}'.format(6, 2014)\
            , fontsize=12, weight='bold') # set figure title
```

```
Out[63]: Text(35,-800,'Figure 6: Modelled Flow & Simulated Modelled Flow with Noise fo
         r Year 2014')
```

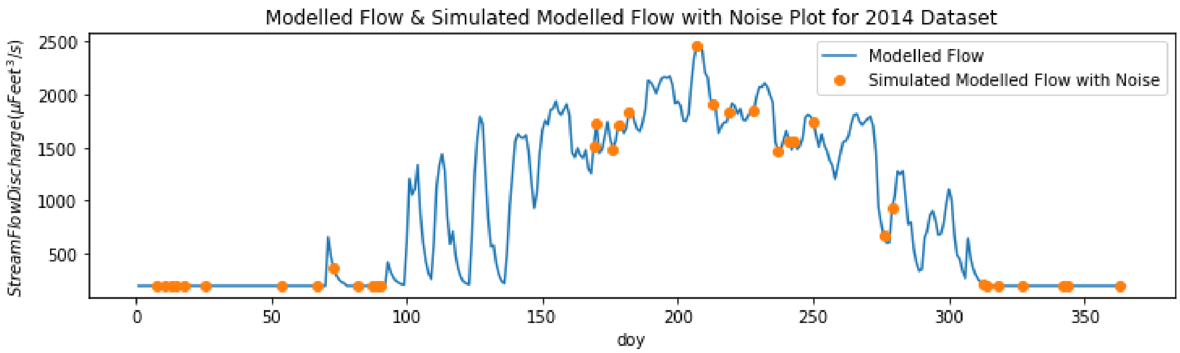

Figure 6: Modelled Flow & Simulated Modelled Flow with Noise for Year 2014

Figure 6 seems to indicate that the current model model & its parameters are able to handle data with noise quite well. We can now be quite confident in using the brute force parameter estimates as a starting point for the Metropolis-Hastings algorithm. Before we define the algorithm in a function, we must first write up a cost function. A cost function calculates the difference between the estimated (from model) and true value (observation) of data, and is often used for parameter estimation.

In [64]:
```
# cost function for the Metropolis Hastings algorithm (where all modelled outp
uts used)
# function based on code written in the synthetic experiment subsection of
# Chapter_6_NonLinear_Model_Fitting by Professor Lewis/Dr. Jose Gonzales

def cost_function(model_parameters, select_flow, \
                  data, sigma_flow, \
                  func=snow_melt_model_version_2):
    '''
    Function to calculate the cost function of the hydrological model. This fu
nction is used to compare the model performance
    of the hydrological model, based on the parameter values inputted for temp
erature_thresh and m. These 2 parameters are
    stored as a list (in model_parameters) to be passed as argument 1 in this
 function.

    This function is based upon the notes and code written in Chapter_6_NonLin
ear_model_Fitting &
    Chapter7_FittingPhenologyModels by Professor Lewis and Dr. Jose Gonzales.

    Parameters
    ----------
    model_parameters: a numpy array
        A numpy array which stores the values for the model parameters tempera
ture_thresh and m. temperature_thresh is a
        floating point number which defines the threshold value for snow pack
 melt to occur. If the daily temperature
        is below this value, no melting occurs, hence no snow-melt input into
 the hydrological system. Exceeding this threshold
        will cause melting to occur.

        m is the decay parameter for the network response function, and is a f
loatoing point number. It controls how quickly
        water from snow pack melts enters and contributes to stream flow.

    select_flow: a numpy array
        A numpy array containing the observed flow for which the hydrological
 model is trying to replicate

    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the hydrologi
cal model.

    sigma_flow: a floating point number
        A floating point number that represents the standard deviations for th
e flow data. For version 1 of the cost function,
        the value corresponds to the standard deviation of the modelled flow d
```

```
ata with noise added to the dataset. For version 2
         of the cost function, the value corresponds to the standard deviation
 of the flow data observation for the year the
         hydrological model is trying to model.

    Returns
    -------
    A floating point number which represents the calculated cost function for
 the hydrological model, based on the values of
    the model parameters passed
    '''
    # generate model flow
    modelled_flow = func(model_parameters, data)
    # remember to pass temperature_thresh as
    # index 0 of the list and m as index 1 of the list

    # calculate the cost function
    cost = -0.5*(modelled_flow - select_flow)**2/sigma_flow**2

    # return model cost function
    return -cost.sum()
```

With the cost function defined, we can begin writing up the Metropolis-Hastings algorithm. Once written, we can then apply it onto the brute force model parameters. The algorithm will return a numpy array of values for T_{thresh} and m explored, with the size of the array defined by the number of iterations of the algorithm and the model parameters wish to explore (2), hence n x 2 size. This will allow for a visualization of the most frequently returned value for T_{thresh} and m, can be seen a peak (which corresponds to the lowest returned cost function value) in a histogram plot. Figure 7 illustrates this below.

In [65]:
```
# function to run Metropolis-Hastings algorithm
# the function is based on the metropolis_hastings function written by Profess
or Lewis/Dr. Jose Gonzales
# in Chapter7_FittingPhenologyModels, under the heading Uncertainty, subheadin
g The Metropolis

def metropolis_hastings(intial_model_parameters, data, cost_function, n_iterat
ions=50000):
    '''
    Function to run a Metropolis-Hastings algorithm on the parameters used in
 the snow-melt hydrological model.
    The function relies on the generation of the cost function value (minimum)
 to evaluate the best values for the
    snow-melt hydrological model. The number of iterations that will be run by
 this function is specified by the
    n_iterations argument.

    This function is based on the metropolis_hastings function written by Prof
essor Lewis/Dr. Jose Gonzales
    in Chapter7_FittingPhenologyModels, under the heading Uncertainty, subhead
ing The Metropolis. Modifications made to
    the original function include:-
    1. Changing some of the function arguments names and number of arguments f
or the function
    2. Addition of comments to the code

    Parameters
    ----------
    initial_model_parameters: a numpy array
        A numpy array containing the parameters needed to run the hydrological
 snow-melt model. These include temperature_thresh
        and m. Note that temperature_thresh should be in index 0, and m should
 be in index 1 of the numpy array.

        These parameters values were generated from the brute force method, an
d evaluated using the minimize numerical
        optimization function. These values serve as the starting point for th
e exploration of plausible values of
        temperature_thresh and m.

    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the hydrologi
cal model and to generate the observed
        stream flow data to evaluate the model performance (through the cost f
unction).

    cost_function: a function
```

```
        A python function which specifies how to run a cost function (pre-writ
ten before this function). The cost function
        evaluates the model performance by returning a score. Lower score valu
e indicate an improvement in model performance.

    n_iterations: an integer
        An integer that specifies the number of iterations of the Metropolis A
lgorithm to run for the plausible model parameter
        space exploration. A default value of 50000 has been set by.

    Returns
    -------
    2 variables are returned from this function:-
        1. sampled_values = a numpy array
            A numpy array which specifies the parameter space explored by the
 Metropolis-Hastings algorithm
        2. sample_plot = a plot image
            A plot image that visualization the parameter space explored by th
e Metropolis-Hastings algorithm. This allows
            for the visualization of the 'best' parameter values to use for th
e hydrological model, specified by the location
            of the peaks of the histograms.
    '''
    # generates the model flow data
    model_flow = snow_melt_model_version_2(intial_model_parameters, data)

    # generate the flow observation data for dataset
    flow_obs = data['river_discharge']

    # define standard deviation for flow_obs
    sigma_flow = np.std(flow_obs)

    # specify number of parameters to evaluate in the Metropolis-Hastings algo
rithm
    n_params = len(intial_model_parameters)

    # set aside storage space for possible values for each parameter to take b
ased on number of iterations of the algorithm
    samples_values = np.zeros((n_iterations, n_params))

    # begin running the Metropolis-Hastings Algorithm
    # specify initial_model_parameters as the current parameter vector
    param_curr = intial_model_parameters*1

    # calculate the initial cost function
    cost_curr = cost_function(param_curr, flow_obs, data, sigma_flow)

    # Iterating over the specified number of iterations
    for i in range(n_iterations):
        # randomly sample the sample space for the model parameters
        param_proposed = param_curr  + np.random.normal(size=n_params)*np.arra
y([0.1, 0.01])
        # note the use of np.random.normal() to generate 2 random values (betw
een 0 and 1) from a random normal distribution
        # the arguments passed in np.array() specify the step size to sample i
n the model parameter space
        # here we're sampling about +- 0.1 (max) away from temperature_thresh
```

```
 and +-0.01 (max) away from m

        # Calculating the cost function for the proposed parameter
        proposed_cost = cost_function(param_proposed, flow_obs, data, sigma_fl
ow)

        # Evaluate the cost function (The Metropolis acceptance)
        # if the proposed cost function value is greater than the initial one,
 will accept value and store it in sample_values
        # if the value is less than the initial one, will sometimes accept (ba
sed on random chance) to allow exploration of new
        # sample space
        if np.random.rand() < np.exp(proposed_cost - cost_curr ):
            # here we accept the proposed parameter values and use it as a new
 starting point to explore the sample space
            param_curr = param_proposed

            # update the cost function value accordingly
            cost_curr  = proposed_cost

        # even if the parameter value is rejected(due to no improvement in the
 cost function or not randomly selected if there
        # was no improvement), still store the value in the sample_values (to
 signify the sample space explored) but
        # the value for the cost function is not updated
        samples_values[i, :] = param_proposed

    # produce a histogram plot to visualize the explored sample space
    fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12,3)) # set up subplot
    axs = axs.flatten() # flatten axis to allow for subplots
    for i in range(2): # want to produce 2 histogram to correspond to 2 model
 parameters
        axs[i].hist(samples[:,i], bins=50, color='0.8') # plotting histogram
        if i == 0:
            axs[i].set_xlabel(f'$temperature_thresh$')
        else:
            axs[i].set_xlabel(f'$m$')

    return samples_values, fig
```

Unfortunately, the Metropolis-Hasting function was not succesfully implemented due to an error associated with unable to pass the filter weight for the snow_melt_model_version2*. Due to lack to time, the author is unable to further optimize the parameter values for $T*{thresh}$and$m$. The brute force method has however yielded results for suitable model parameters, that can be applied to evaluate the model performance for the year 2014 (calibration year) and 2015 (validation year).

Section 3 will now address the model's accurary by plotting the model output against the calibration and validation year. The model parameters used to generate these outputs will include both the initial guess and the brute force guess, and will be displayed in a table. Finally, the model's accuracy for predicting the calibration and validation year will be assessed by reporting the sum of redisual square value for each year

Part 3: Model Results

Using the the initial guess and brute force estimate for the values of T_{thresh} & m, the following plots were produced for the calibration year (2014 dataset).

In [91]:
```
# function for producing plots of modelled stream flow (using different parame
ter estimates) and observed stream flow
def stream_flow_plots(data, year, fig_no):
    '''
    Function for producing plots of modelled stream flow, for all versions of
 the model, with the observed stream flow.

    Parameters
    -----------
    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the hydrologi
cal model.

    year: an integer
        An integer that represents the year of the dataset used. Choice betwee
n 2014 and 2015.

    fig_no: an integer
        An integer that is used to set up the figure number of the plot.

    Returns
    -------
    A plot of the modelled stream flow against the observed stream flow
    '''
    # generate model parameters for different versions of the model
    model_1_param = np.array([6.0])
    model_2_param_initial = np.array([6.0, 0.03])
    model_2_param_brute = brute_guess

    # generate model flow data
    model_1_flow = snow_melt_model_version_1(model_1_param, data)
    model_2_flow_initial = snow_melt_model_version_2(initial_guess, data)
    model_2_flow_brute = snow_melt_model_version_2(brute_guess, data)

    # grab observed stream flow from data
    obs_flow = data['river_discharge']

    # grab doy from data
    doy = data['doy']

    # generate plot
    # plot set up
    plt.figure(figsize=(12,5))
    plt.xlim(doy[0], doy[-1]) # correction of start date as should start from
 doy 0, not doy 1
    plt.xlabel('day of Year {}'.format(year))
    plt.ylabel('$Stream Flow Discharge ( \mu Feet^3/s)$')
```

```
    # plotting the data
    plt.plot(doy, model_1_flow/100., 'y', label='1st hydrological model estima
ted flow/100')
    plt.plot(doy, model_2_flow_initial/100., 'r', label='2nd hydrological mode
l intial estimated flow/100')
    plt.plot(doy, model_2_flow_brute/100., 'g', label='2nd hydrological model
 brute estimated flow flow/100')
    plt.plot(doy, obs_flow/100., 'b', label='observed stream flow/100')
    plt.legend(loc='best')
    plt.title('Modelled vs. Observed Stream Flow for Year {}'.format(year))
    plt.text(55, -13, 'Figure {}: Modelled vs. Observed Stream Flow for Year
{}'.format(fig_no, year)\
             , fontsize=12, weight='bold')
```

In [92]:
```
# generate the plot for year 2014
stream_flow_plots(data_2014, 2014, 7)
```

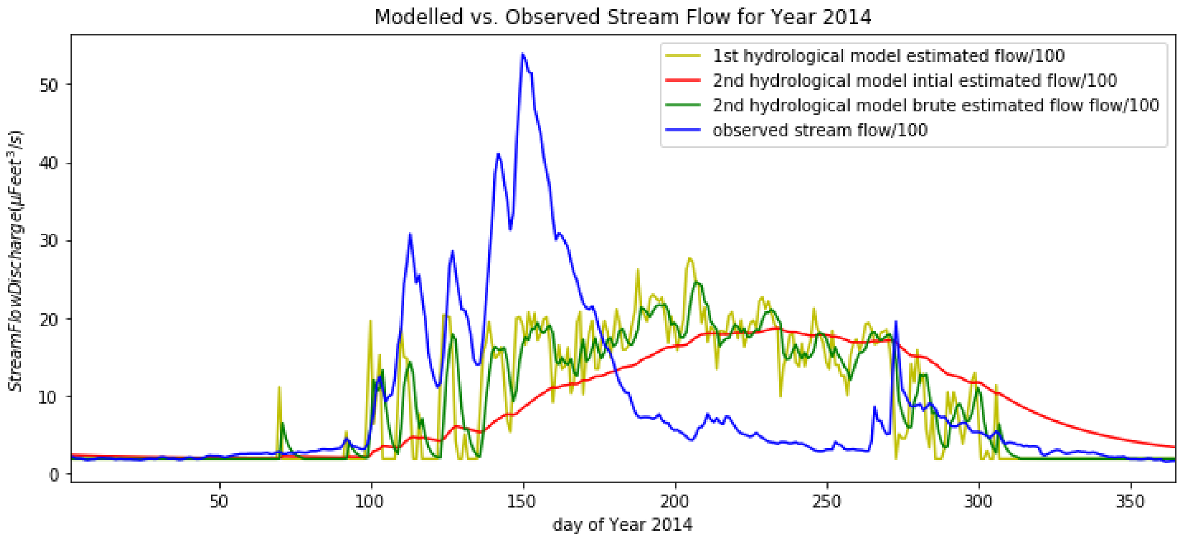

Figure 7: Modelled vs. Observed Stream Flow for Year 2014

As seen in Figure 7, the 2nd version of the hydrological model which uses the brute force estimates for the parameters T_{thresh} and m produces a modelled flow in between the 1st hydrological model and the 2nd hydrological model using the initial parameter estimates. The addition of the network response function does indeed help to delay the effect of the snow-melt affecting the flow discharge.

That being said, the 2nd version of the model, even with the brute force estimated values, doesn't seem to exactly track the observed flow discharge for the year 2014. This lack in model accuracy is captured in table 2 below, in the form of reported sum of residual square.

In [93]:
```
# function for creating dataframe for reporting model performance

def model_performance_table(data):
    '''
    Function for a data frame for reporting model performance

    Parameters
    -----------
    data: a dictionary
        A dictionary containing doy (a list of integers), temperature (a panda
s dataframe), stream flow discharge
        (a numpy array) and mean snow cover (a numpy array) data. The data was
 prepared in part 1 of the practical.
        Either one of the dataset (2014 or 2015) can be passed as data for thi
s function. To access the contents of
        the dictionary, the following keys can be used:- 'doy', 'temperature',
 'river_discharge', and 'snow_cover'.

        This dictionary is used to feed the data required to run the hydrologi
cal model.

    Returns
    -------
    '''
    # generate model parameters for different versions of the model
    model_1_param = np.array([6.0])
    model_2_param_initial = np.array([6.0, 0.03])
    model_2_param_brute = brute_guess

    # generate model flow data
    model_1_flow = snow_melt_model_version_1(model_1_param, data)
    model_2_flow_initial = snow_melt_model_version_2(initial_guess, data)
    model_2_flow_brute = snow_melt_model_version_2(brute_guess, data)

    # generate observed flow
    flow_obs = data['river_discharge']

    # generate sum of residual square for each model
    # model_1
    residual_model_1 = model_1_flow - flow_obs
    sum_residual_model_1 = residual_model_1*residual_model_1
    sum_residual_model_1 = sum_residual_model_1.sum()

    # model_2_intial
    residual_model_2_initial = model_2_flow_initial - flow_obs
    sum_residual_model_2_intial = residual_model_2_initial*residual_model_2_in
itial
    sum_residual_model_2_intial = sum_residual_model_2_intial.sum()

    # model_2_brute
    residual_model_2_brute = model_2_flow_brute - flow_obs
    sum_residual_model_2_brute = residual_model_2_brute*residual_model_2_brute
    sum_residual_model_2_brute = sum_residual_model_2_brute.sum()

    # save information into a dataframe
    # initialize dataframe
```

```
    results_df = pd.DataFrame()

    # store temperature values
    temp_data = pd.DataFrame({"temperature_thresh": [6.0, 6.0, round(brute_gue
ss[0],2)]})
    results_df = results_df.append(temp_data)

    # store m values
    results_df['m'] = pd.Series(['NaN', 0.03, round(brute_guess[0],2)])

    # store sum of residual square values
    results_df['sum_of_residuals_square'] = pd.Series([sum_residual_model_1,\
                                                        sum_residual_model_2_in
tial,\
                                                        sum_residual_model_2_br
ute])

    # return dataframe
    return results_df
```

In [94]:

```
print( '\033[1m' + 'Table 2: Model Performance for Year 2014')
model_performance_table(data_2014)
```

Table 2: Model Performance for Year 2014

Out[94]:

	temperature_thresh	m	sum_of_residuals_square
0	6.00	NaN	4.109202e+08
1	6.00	0.03	5.076129e+08
2	5.55	5.55	3.740301e+08

The intial introduction of the network response function seems to reduce the model performance. However, upon using the brute force method, the model performance has improved.

Having produced the plot and table representing model accuracy for the calibration year (2014), lets do the same for the validation year.

In [95]:
```
# generate the plot for year 2015
stream_flow_plots(data_2015, 2015, 8)
```

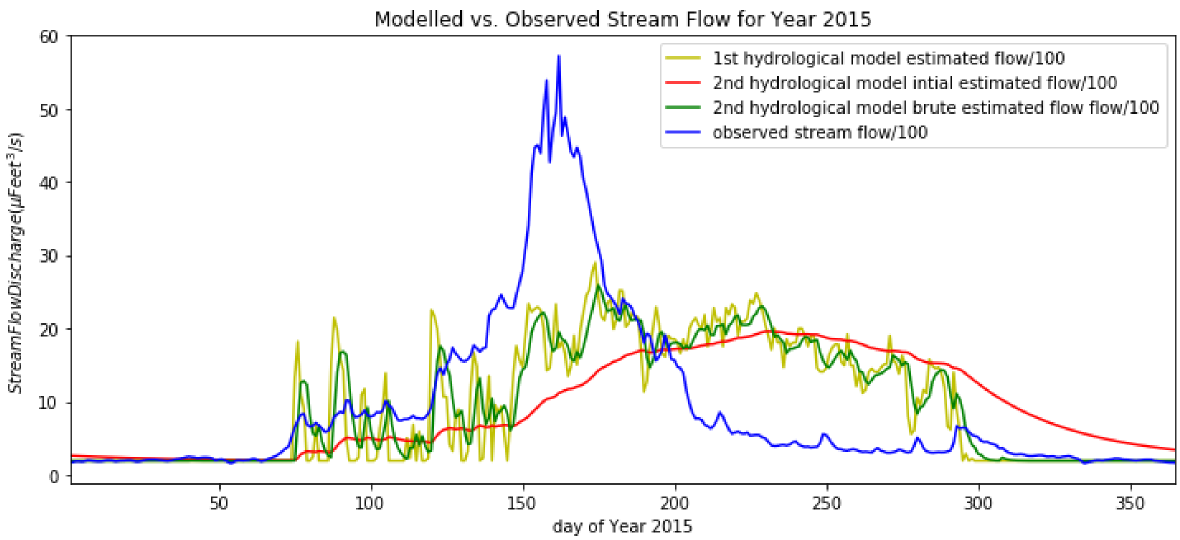

Figure 8: Modelled vs. Observed Stream Flow for Year 2015

Similar to the calibration year, the hydrological model using the brute force estimates seems to be the best best current model for modelling the stream flow.

In [96]:
```
print( '\033[1m' + 'Table 3: Model Performance for Year 2015')
model_performance_table(data_2015)
```

Table 3: Model Performance for Year 2015

Out[96]:

	temperature_thresh	m	sum_of_residuals_square
0	6.00	NaN	3.587850e+08
1	6.00	0.03	5.089776e+08
2	5.55	5.55	3.375964e+08

The model performance for the validation year also seems to indicate an intial worsening in model performance when switching versions (from 1 to 2, with the inclusion of the network response function). However, when run with the brute force estimated values, the model performance improves, as seen in the drop in the reported sum of residual square (from 3.587850e+08 to 3.375964e+08)

Section 4: Discussion

The main comment with regards to the hydrological model produced is it's lack in ability to closely track the observed stream flow for both the calibration year (2014) and validation year(2015). This could be due to the author's inability to further optimize the model parameter values T_{thresh} and m, due to the failure to correctly execute the Metropolis-Hasting function.

The quality of the snow-melt data, which serves as the main driver of the model, could also be a reason for the inability of the model to closely model the observed stream flow. The snow-melt data, in its preprocessed from, had lots of gaps in the dataset due to lack of data (cloud cover reasons, poor data quality), which had to be filled in through interpolation. This was despite the author's best efforts to fill in as much of the dataset without using interpolation techniques (by using data from both the Terra and Aqua satellites). When applying the interpolation, if the gap in data was too large, a default value of 0.5 was filled into to the represents the missing data.

This in turn, is likely to severely affect the hydrological model produced above to model the stream flow discharge accurately.

Section 5: Conclusion

The practical has produced 2 versions of the snow-melt hydrological model to replicate the stream discharge observed at Del Norte. Due to the inability to fully optimize the model parameters T_{thresh} and m, as well as the poor quality of the snow melt dataset used, the model is unable to closely model the observed stream discharge. It would be interesting to observe the model's performance when using a higher quality snow melt dataset.