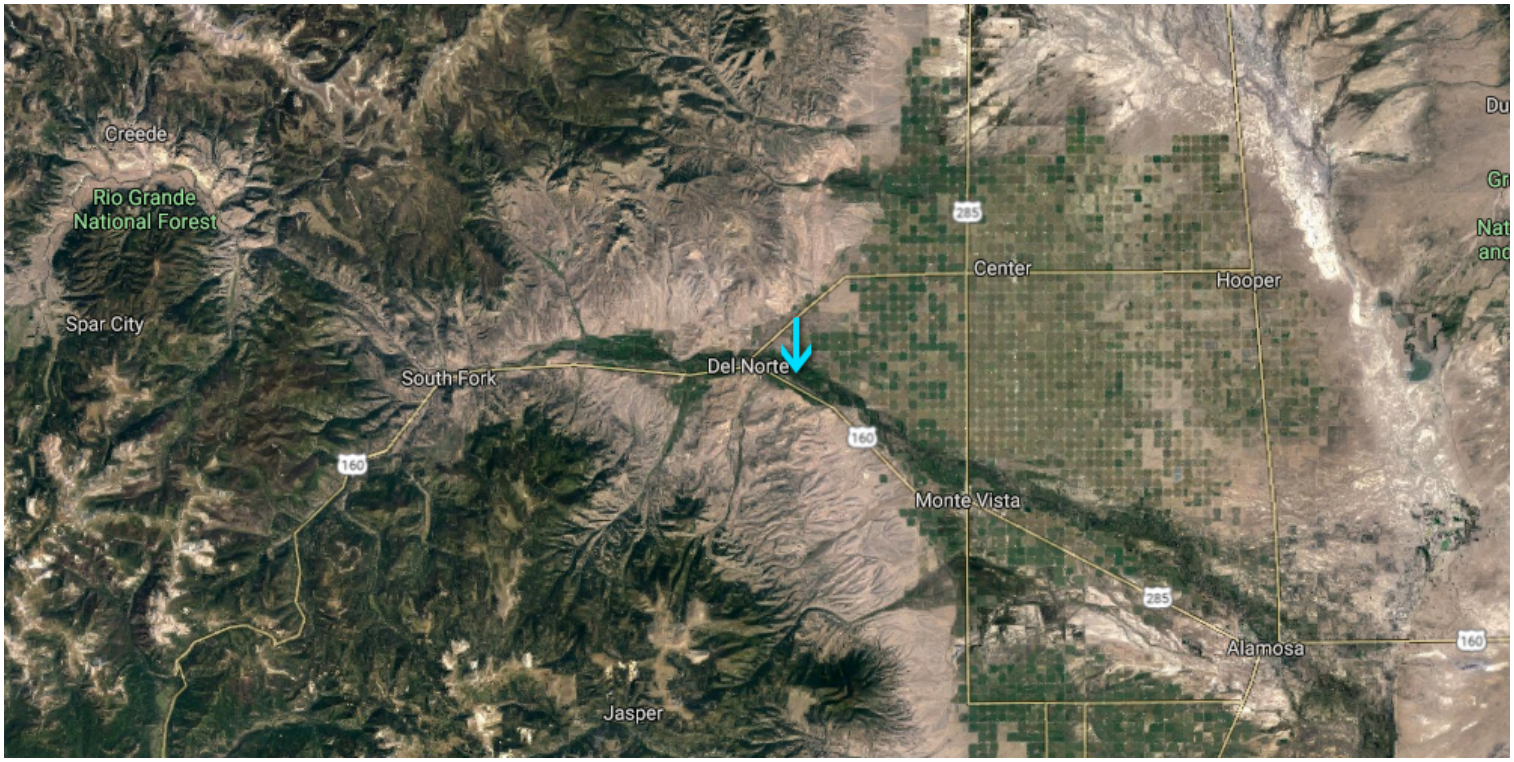
Scientific Computing: Coursework Part 2

1. Introduction:

This report aims to create and analyse a model of snowmelt/discharge using meteorological and land-cover data which has previously been collected and processed. Remaining fairly simple the constituent variables include discharge, temperature, and snow cover for the given site. Data has been collected in situ at the Del Norte 2E site and using satellite measurements from MODIS' Terra and Aqua modules.

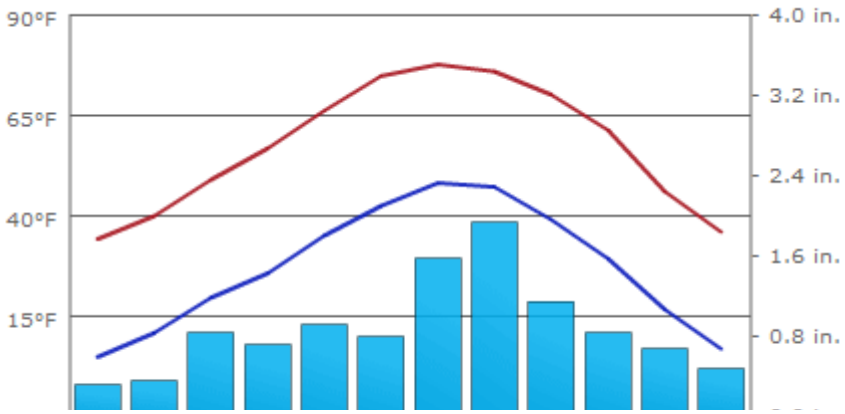
Del Norte monitoring station is a meteorological data collection site situated in Colorado, western-central USA (37.67417, -106.32472). The site sits at 2,390m elevation and is situated eastwards of the Rio Grande national forest boundary inside of which elevation rises to over 3,650m - a major source of snowpack and associated melt water. Interestingly, downstream and to the east of the monitoring station is a large region of center-pivot irrigation agricultural land which relies upon a large store of water (in this case likely from snowmelt) to farm effectively. This can be seen in figure 1 below.

Figure 1:



In many ways the geography of this site is typical to Colorado, bridged by the Rocky Mountains the state is extremely variant in elevation and has the highest peak in the Rockies at 4,401m Mt. Elbert, the lowest point at the border with Kansas follows the course of the Arikaree river at 1,011m. Elevation in the region surrounding the Del Norte site is extremely important in preserving snowfall by maintaining lower daily temperatures with altitude and thereby increasing the water stored in snowpack seasonally. This seasonal store of water is crucial to understanding the dynamics of river discharge at the site as temporal shifts in temperature can bring about huge pulse events which a model can be used to predict both in terms of timing and intesity. The seasonal variability of the site can be seen in Figure 2 below, do note that the low precipitation during months of below-freezing temperature is due to snowfall being catergorised separately but that the water storage in the system is generally increasing during this time.

Figure 2:



2. Method:

- Provide an introduction to the modelling and calibration/validation ('Method')
- Also, start to add some references to literature here.

Before modelling can begin it is important to reiterate the data/variables that are available as well as the goal of the model procedure. The purpose of this report will be to create a model of discharge at the Del Norte site for each day of the consituent year using the previously attained inputs of snow cover and temperature. Although there are only a limited number of variables avaiable a fairly robust model can be built as in mountainous regions seasonal snow accumulation and ablation dominate water-cycle processes (Abudu, 2012) (https://doi.org/10.3882/j.issn.1674-2370.2012.02.001).

The datasets to be used in this report are those provided by the department for the years 2005 and 2006 along with data processed in the previous portion of this coursework for 2016 and 2017. For the latter two years there exists prepared data for daily precipitation totals which may be useful should I decide that the aforementioned variables do not model snowmelt/discharge well enough. It is of course important to factor the introduction of water from precipitation into the runoff/river system as the combination of snowmelt pulse and heavy rain events are important in flood-hazard assessment (Ferguson, 1999) (https://doi.org/10.1177/030913339902300203).

2.1 Building a model:

For the purposes of this report the constituent variables will be represented in the following way:

- p - Snow cover.
- T - Temperature.

The initial model will follow that built in the instructions for this practical, any changes in approach will be discussed at the point of their introduction.

To begin to understand how discharge from a snowmelt dominated environment changes over time first we must understand how much snow (and stored water) exists in the system at any given moment. The volume of water stored as snow throughout the catchment will be known as the Snow Water Equivalent (SWE) and can be estimated from the data by some function of the snow cover (p) by a given depth of snow (d), over the entire area of the catchment (A).

$$SWE = A.\,p.\,d$$

As there is no given information on depth an assumption will have to be made given the available data, therefore we will assume that depth varies similarly to snow cover:

$$d = \frac{k}{A}p$$

Then:

$$SWE = kp^2$$

In order for water contained within snowpack to be released as discharge two conditions must be met:

- Presence of snow cover in the system ($p > 0$)
- Temperature rising above a level to trigger melting ($T > T_{thresh}$)

Where T_{thresh} is the threshold temperature.

Having reviewed the literature in regards to an adequate starting value for the temperature threshold the consensus is high variability in regional, and temporal terms (Hock, 2003) (https://doi.org/10.1016/S0022-1694(03)00257-9), more advanced models tend to use a distributed temperature index throughout the modelled area to better capture precise dynamics (Cazorzi, 1996) (https://www.sciencedirect.com/science/article/pii/0022169495029133). However, for the basis of this model a singular value will be used for modelling the system throughout the year, (Hock, 2003) (https://doi.org/10.1016/S0022-1694(03)00257-9) reviews models based on a variety of mountainous systems and for non-glaciated sites within similar latitudes/altitude to the Del Norte station there is exhibited a range of 2.5 - 5.5 degrees

as temperature threshold for **snow** melt. It is beyond the scope of the available data to tell how much of the snow cover in the catchment can be considered ice which has a decidedly higher threshold value for significant melting. For this reason the T_{thresh} value used will be on the higher end of that taken from literature at 5.5 degrees.

To model the potential SWE entering the system the available water can be set equal to a proportion k_p of the water store (SWE) on days where the temperature threshold is exceeded and melting occurs.

Then:

$$SWE_{melt}(t) = k_p(t)SWE(t)$$

Where SWE_{melt} is the proportion of SWE released per unit time (day).

Setting k_p proportionate to the excess temperature allows a control on the severity/extent of the melt. However this could have limitations as this assumes a somewhat linear relationship.

We can define this as:

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

With negative k_p set to zero, and T_{max} the maximum temperature. So, if $T = T_{thresh} + T_{max}$. $k_p = 1$ the maximum amount of SWE is available as melt water.

There is a base flow throughout the year which may come from sources to the river from outside the catchment, or inside as spring water. This can be estimated by averaging the flow in January where the snowpack should be fairly consistent with very little melting, this can then be applied at a constant rate throughout the year. It is important to note however that in the event of precipitation (given permitting temperature) during the calibration month could askew model predictability.

This can be added to the model as F_{base}.

From this value the remaining discharge (assumed to be via snowmelt) can be calculated as follows:

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

F_{model} entering the system:

$$F_{model}(t) = F_{base} + SWE_{melt}(t)$$

$$F_{model}(t) = F_{base} + kMAX\left(0, \frac{T - T_{thresh}}{T_{max} - T_{thresh}}p(t)^2\right)$$

We can assume that the total volume of flow modelled is equal to that measured (once removing base flow), this acts as a constraint upon the model.

Then:

$$\Sigma_t F_{model}(t) = \Sigma_t F(t)$$

so

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

This allows us to infer the value of k from the data:

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

2.2 Calibrating the model:

In order to calibrate the model to optimal performance first a separate function must be made to quantify the extent of the model residuals, the difference between the values of flow that the model predicts and the actual values of the training set or year. This function will enable searching of parameter space to find the optimal parameter values for the model.

For the model ($f(\vec{p}) = y$) where $f(\vec{p})$ is the transformation function that converts the given data in snow cover, temperature, and discharge into modelled flow (y). A likelihood (cost) function ($l(\vec{p})$) can be built taking into account the variance in the observations, (method from chapter 6):

$$l(\vec{p}) = \left[\frac{1}{\sqrt{2\pi\sigma_{obs}^2}}\right]^N \prod_{i=1}^{N} \exp\left[-\frac{(y_n^i - f(\vec{p})^i)^2}{2\sigma_{obs}^2}\right].$$

It is convenient to take a logarithm of $l(\vec{p})$, so that we have the **log-likelihood**:

$$L(\vec{p}) = -\sum^{N} \left[\frac{(y_n^i - f(\vec{p})^i)^2}{\hat{}}\right] + \text{Const.}$$

3. Modelling Snowmelt:

In the previous, accompanying part to this report datasets were downloaded, cleaned, and interpolated where needed for the variables temperature, snow cover, and discharge at the Del Norte 2E site and surrounding catchment.This part of the reporting will aim to create a hydological model of discharge based upon constituent datasets of the aforementioned variables.

3.1 Module Imports:

In [208]:
```
# Importing the necessary python modules which will be used later in the code.

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import scipy
import scipy.ndimage.filters
from scipy.optimize import minimize
import pickle
import pandas as pd
```

3.2 Importing Training Datasets:

In [191]:
```
# Importing the department datasets from instruction code given in the practical 2 se
tup.
# The files will be unpacked and altered to the correct units if necessary.
# I will restack the variables so that the new dictionary headers match those that I
 created in my own work.

pkl_file = open('data/Pdata2005.pkl', 'rb')
# note encoding='latin1' because pickle generated in python2
data = pickle.load(pkl_file, encoding='latin1')
pkl_file.close()


# Setting variable names and converting units.
discharge05 = data['flow'] / 35.315
temp05 = data['temp']
snow05 = data['snowprop']

header = ["flow", "temp", "snow"]

arr = np.vstack((discharge05, temp05, snow05))

data05 = dict(zip(header, arr))


pkl_file = open('data/Pdata2006.pkl', 'rb')
# note encoding='latin1' because pickle generated in python2
data = pickle.load(pkl_file, encoding='latin1')
pkl_file.close()


# Setting variable names and converting units.
discharge06 = data['flow'] / 35.315
temp06 = data['temp']
snow06 = data['snowprop']

arr = np.vstack((discharge06, temp06, snow06))

data06 = dict(zip(header, arr))
```

3.3 Importing Previous Datasets:

In [219]:
```
# Setting variables for the dataset filenames produced in part 1:
# I setup this function before I had figured out exactly how minimisation and modelli
ng would work.
# There are parts which have been made redundant by further code in this section.

filename16 = "Hydrological_Variables_2016.npz"
filename17 = "Hydrological_Variables_2017.npz"

def load_unpack(filename):
    """
    A function to quickly unpack previously saved datasets of hydrological variables:

    Arguments:

    filename (str) - The filename of the file to be loaded and unpacked.

    Returns:

    Discharge (D), Temperature (T), and Snow Cover (S) as arrays.

    """
    # load in file.
    data = np.load(f"{filename}")
    # convert to dictionary.
    data_dict = dict(data)
    # call specific keys in dictionary as defined in previous part.
    # convert data to float rather than object values to allow calculations with np t
o be taken.
    D = np.asarray(data_dict["discharge"], dtype=float)
    T = np.asarray(data_dict["temperature"], dtype=float)
    S = np.asarray(data_dict["snow_cover"], dtype=float)

    # return data.
    return D, T, S

discharge16, temp16, snow16 = load_unpack(filename16)
discharge17, temp17, snow17 = load_unpack(filename17)


## QUICK DATAFIX:

# Temperature datasets were missing values and had NaN placeholders.
# Whilst this wasn't a problem in part 1 this creates unnecessary hassle when using
# this data to create and calibrate a model.

def interpolate_nan(data):
    """
    Function to replace/interpolate over NaN values in data array.
    Arguments:

    data (numpy array) - The array to be cleansed.

    Returns:

    data (numpy array) - The interpolated array.

    Method from: https://stackoverflow.com/questions/9537543/replace-nans-in-numpy-ar
ray-with-closest-non-nan-value

    I would like to add that only the np.flatnonzero() was unknown to me before searc
hing for a solution.

    """
    # Find index of NaN values in array.
    mask = np.isnan(data)
```

```
    # Replace data at index of NaN values with interpolated values between non-zero
 (i.e. non-NaN)
    # indices in mask array using values from the input array (data).
    data[mask] = np.interp(np.flatnonzero(mask), np.flatnonzero(~mask), data[~mask])
    return data

# Applying to temperature data.
temp16 = interpolate_nan(temp16)
temp17 = interpolate_nan(temp17)

header = ["flow", "temp", "snow"]

arr = np.vstack((discharge16, temp16, snow16))
data16 = dict(zip(header, arr))

arr = np.vstack((discharge17, temp17, snow17))
data17 = dict(zip(header, arr))


# Now I have 4 idential dictionary-arrays set up for each data year which will allow
 me to build a model
# which can seamlessly work with all data.
```

3.4 Plotting the data:

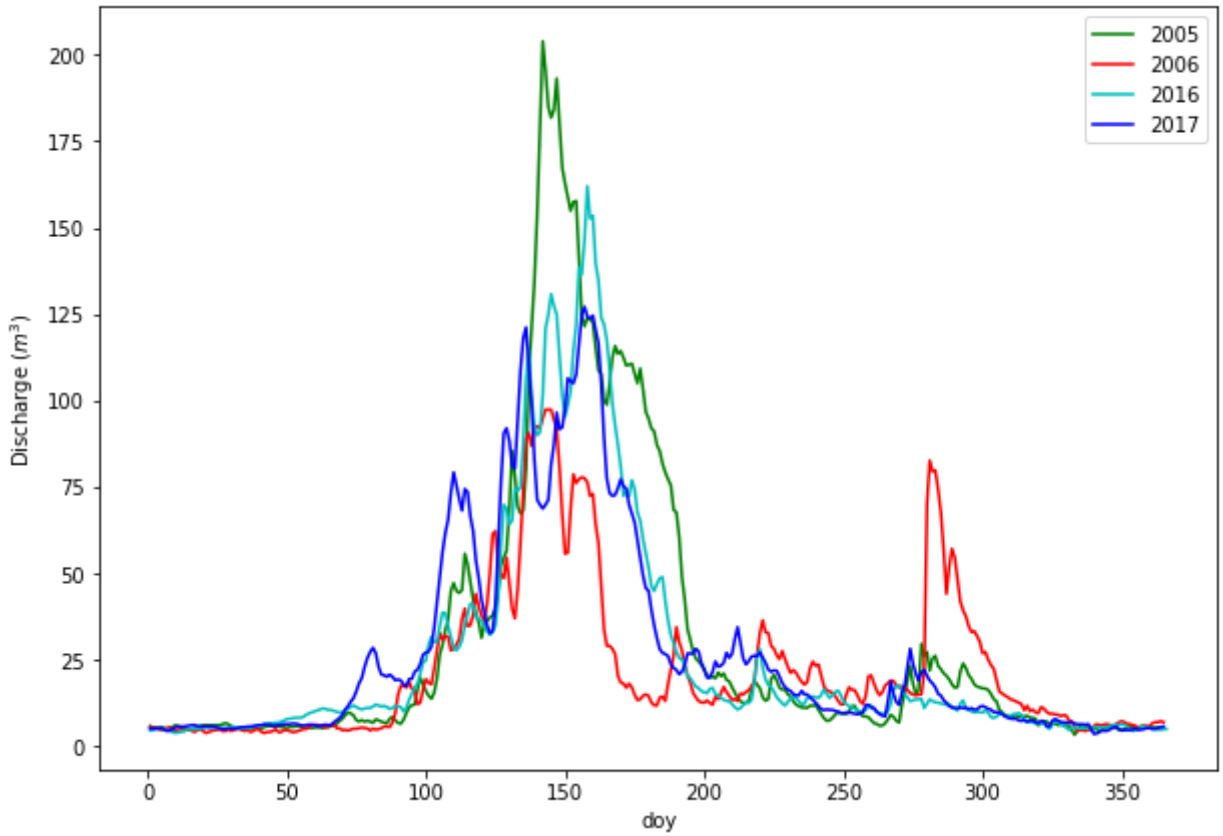
To get an idea of the existing similarity between the data which will be used to create/train the model (year 2005) and the remaining years which will be used for validation, a simple graphical output will suffice.

In [220]:

```
## DISCHARGE ##

# Creating a x-axis for leap and normal years.
x_axis = np.arange(1,366)
x_axisl = np.arange(1,367)

# Creating the figure and axis labels.
plt.figure(figsize=(10,7))
plt.xlabel('doy')
plt.ylabel('Discharge ($m^3$)')

# plot data
plt.plot(x_axis, data05["flow"],'g',label='2005')
plt.plot(x_axis, data06["flow"],'r',label='2006')
plt.plot(x_axisl, data16["flow"],'c',label='2016')
plt.plot(x_axis, data17["flow"],'b',label='2017')

plt.legend()

# Creating placeholder lists for summary statistics.

total = []
mean = []
mini = []
argmin = []
maxi = []
argmax = []
stdev = []

datasets = [data05, data06, data16, data17]

for data in datasets:
    total.append(data["flow"].sum())
    mean.append(data["flow"].mean())
    mini.append(data["flow"].min())
    argmin.append(data["flow"].argmin())
    maxi.append(data["flow"].max())
    argmax.append(data["flow"].argmax())
    stdev.append(data["flow"].std())

years = ["2005", "2006", "2016", "2017"]


# Creating a table using pandas to display summary statistics

df = pd.DataFrame({ "Year": years,
                    "Total": total,
                     "Mean": mean,
                    "StDev": stdev,
                    "Minimum": mini,
                    "Minimum Time (doy)": argmin,
                    "Maximum": maxi,
                    "Maximum Time (doy)": argmax
                    })

df
```

Out[220]:

	Year	Total	Mean	StDev	Minimum	Minimum Time (doy)	Maximum	Maximum Time (doy)
0	2005	11331.643777	31.045599	43.169020	3.454623	332	203.879371	141
1	2006	8139.968852	22.301285	22.329820	3.964321	20	97.409033	142
2	2016	9529.792506	26.037684	33.683888	3.964352	9	161.972096	157
3	2017	9855.209171	27.000573	30.187022	3.681184	339	127.142432	156



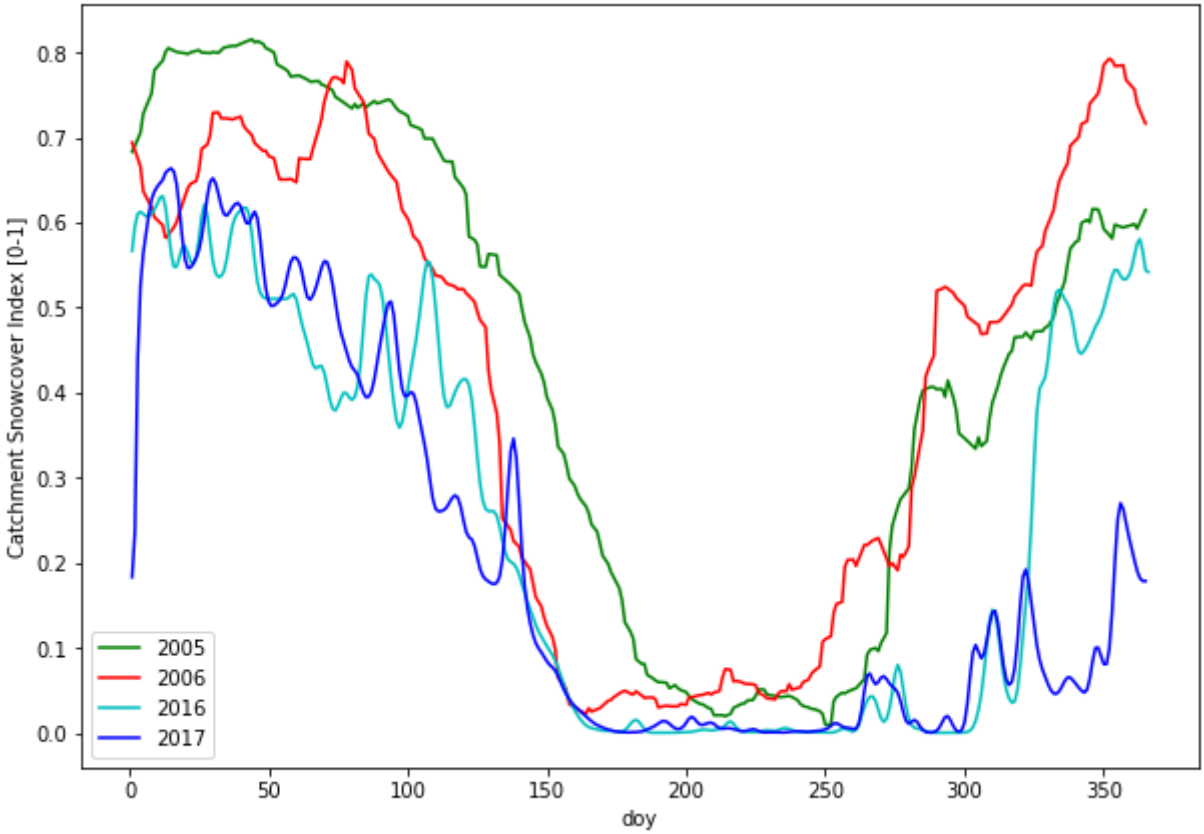
Although crowded it is clear that the 4 available data years exhibit some common trends, a real peak in the yearly watershed is felt around the end of May / start of June with some propensity for early or late season flux events. November through to February see exceptionally low output throughout the data as snowpack remains frozen and begins to replenish. It is also clear that 2006 is somewhat of an outlier with significantly less discharge during the normal summer peak and a heavy pulse event in the autumn, it is likely that this abnormality will make fitting a model using this year as a baseline unlikely to translate well to the other given years.

In [222]:

```
## SNOW COVER ##

# Creating the figure and axis labels.
plt.figure(figsize=(10,7))
plt.xlabel('doy')
plt.ylabel('Catchment Snowcover Index [0-1]')

# plot data
plt.plot(x_axis, data05["snow"],'g',label='2005')
plt.plot(x_axis, data06["snow"],'r',label='2006')
plt.plot(x_axisl, data16["snow"],'c',label='2016')
plt.plot(x_axis, data17["snow"],'b',label='2017')

plt.legend()

# Creating placeholder lists for summary statistics.

mean = []
mini = []
argmin = []
maxi = []
argmax = []
stdev = []

for data in datasets:
    mean.append(data["snow"].mean())
    mini.append(data["snow"].min())
    argmin.append(data["snow"].argmin())
    maxi.append(data["snow"].max())
    argmax.append(data["snow"].argmax())
    stdev.append(data["snow"].std())

# Creating a table using pandas to display summary statistics

df = pd.DataFrame({ "Year": years,
                    "Mean": mean,
                    "StDev": stdev,
                    "Minimum": mini,
                    "Minimum Time (doy)": argmin,
                    "Maximum": maxi,
                    "Maximum Time (doy)": argmax
                    })

df
```

Out[222]:

	Year	Mean	StDev	Minimum	Minimum Time (doy)	Maximum	Maximum Time (doy)
0	2005	0.441105	0.282818	9.346889e-03	249	0.815732	43
1	2006	0.408119	0.274626	2.241416e-02	163	0.792999	351
2	2016	0.250777	0.238547	2.305410e-07	189	0.631250	11
3	2017	0.205268	0.224249	2.846601e-04	234	0.663925	14



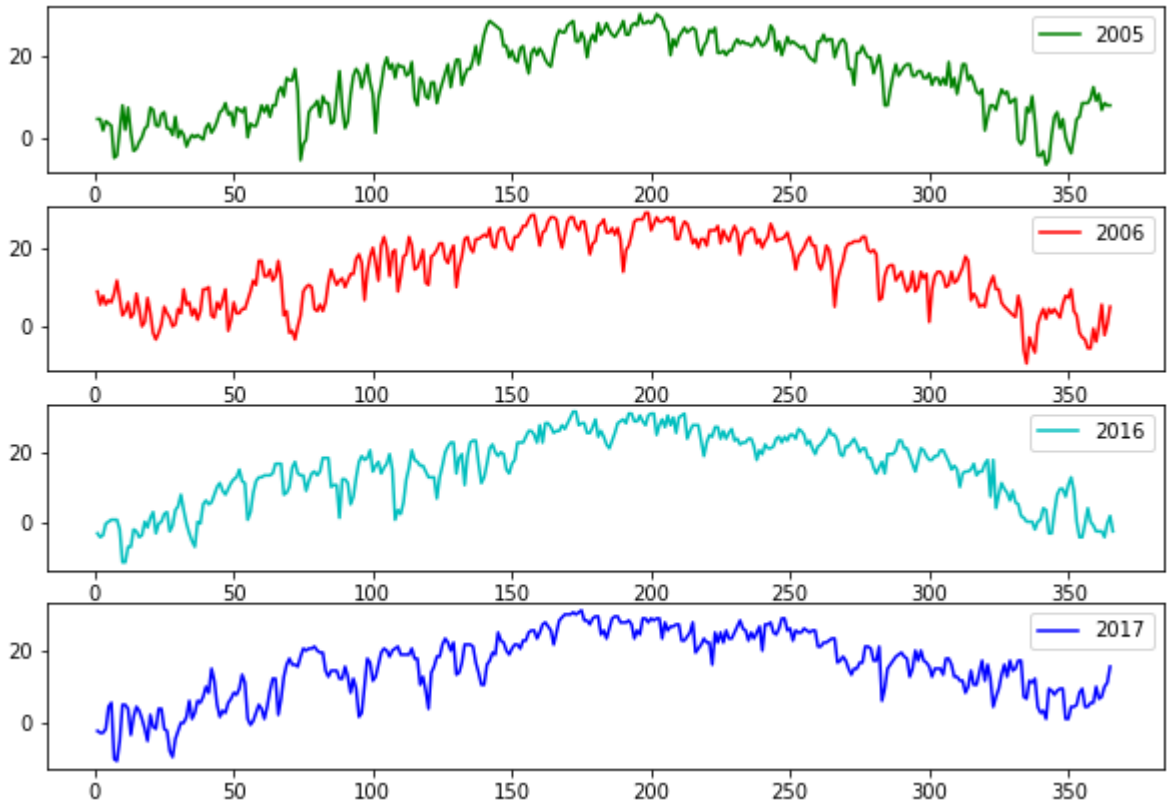
Snow cover is fairly consistent throughout the years available, and following a standard yearly accumulation-melt cycle the relationship between discharge and melting snowpack becomes clear. Notable is the slight reduction in yearly snowcover and by extension SWE in the years 2016 and 2017, the reduction in total discharge in these years is therefore explained. Both the latter two years also fail to recover snowpack as quickly in the winter season with 2017 being particularly lacking in significant snowfall before the turn of the year. What remains unclear is the stange behavior exhibited in the year 2006 as the snowpack and melt seems to be fairly normal but does not match the aforementioned reduced discharge at all like the pattern exhibited by other years.

In [237]:

```
## TEMPERATURE ##

# Creating the figure and axis labels.
fig = plt.figure(figsize=(10,7))

# plot data

ax1 = plt.subplot(411)
plt.plot(x_axis, data05["temp"],'g', label='2005')
plt.legend()

ax2 = plt.subplot(412, sharex=ax1)
plt.plot(x_axis, data06["temp"],'r', label='2006')
plt.legend()

ax3 = plt.subplot(413, sharex=ax1)
plt.plot(x_axisl, data16["temp"],'c', label='2016')
plt.legend()

ax4 = plt.subplot(414, sharex=ax1)
plt.plot(x_axis, data17["temp"],'b', label='2017')
plt.legend()

# Creating placeholder lists for summary statistics.

mean = []
mini = []
argmin = []
maxi = []
argmax = []
stdev = []

for data in datasets:
    mean.append(data["temp"].mean())
    mini.append(data["temp"].min())
    argmin.append(data["temp"].argmin())
    maxi.append(data["temp"].max())
    argmax.append(data["temp"].argmax())
    stdev.append(data["temp"].std())

# Creating a table using pandas to display summary statistics

df = pd.DataFrame({ "Year": years,
                    "Mean": mean,
                    "StDev": stdev,
                    "Minimum": mini,
                    "Minimum Time (doy)": argmin,
                    "Maximum": maxi,
                    "Maximum Time (doy)": argmax
                    })

df
```

Out[237]:

	Year	Mean	StDev	Minimum	Minimum Time (doy)	Maximum	Maximum Time (doy)
0	2005	14.212627	9.117230	-6.666667	341	30.000000	195
1	2006	14.183932	9.041070	-9.444444	334	28.888889	197
2	2016	15.236794	10.098913	-11.666667	9	31.666667	171
3	2017	15.825723	9.242076	-10.555556	7	31.111111	174

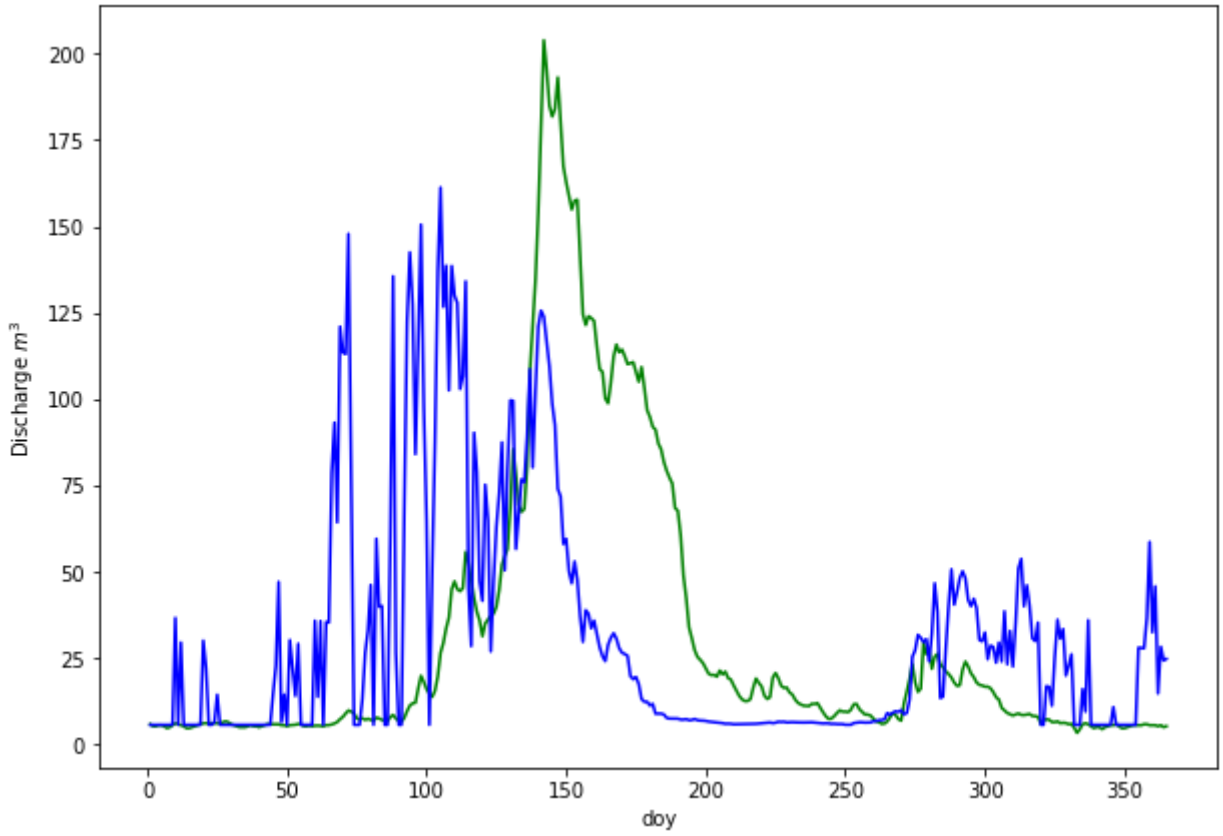


There is no obvious disparity here between the data collected in 2006 and the remaining years other than a slightly lower peak temperature. The temperature data appears to be fairly uniform and reflective of a normal mid-mountainous climate that could be expected of the Colorado site.

3.5 Creating the Model:

In [238]:
```
# Creating the model in python from the mathematical instructions given in section 2.
1

def flow_model(t_thresh, data):

    """
    Function to model discharge given a dataset of discharge, temperature, and snow c
over.
    Arguments:

    t_thresh - The temperature threshold value, at which water can be released from t
he snowpack into the system.

    data - The dictionary containing the arrays of input variables for the given year
s.

    Returns:

    flow - modelled discharge over the given year.

    """

    d = data["flow"]
    p = data["snow"]
    t = data["temp"]

    # Averaging the January discharge to approximate base flow where melt should be m
inimal due to lowest temperatures.

    base_flow = d[:31].mean()
    nonbase_flow = d - base_flow

    k_p = (t - t_thresh)/(t.max() - t_thresh)

    # Verbatim from Chapter 8 practical instructions part 2.
    # Setting up a check for maximum in this way allows an effective on/off for flow
 where temperature breaches threshold.
    k_p = np.max([np.zeros_like(k_p),k_p],axis=0)

    swe_melt  = p * p * k_p

    k = nonbase_flow.sum() / swe_melt.sum()

    flow = swe_melt * k + base_flow

    return flow

# Testing the model with the preliminary t_thresh value given in literature.

model05 = flow_model(5.5, data05)

plt.figure(figsize=(10,7))
plt.xlabel('doy')
plt.ylabel('Discharge $m^3$')

# plot data
plt.plot(x_axis, data05["flow"],'g',label='Observed Discharge')
plt.plot(x_axis, model05,'b',label='Modelled Discharge')
```

```
Out[238]: [<matplotlib.lines.Line2D at 0x7fef7ff8c2e8>]
```



3.6 Adding Delay to Model Output:

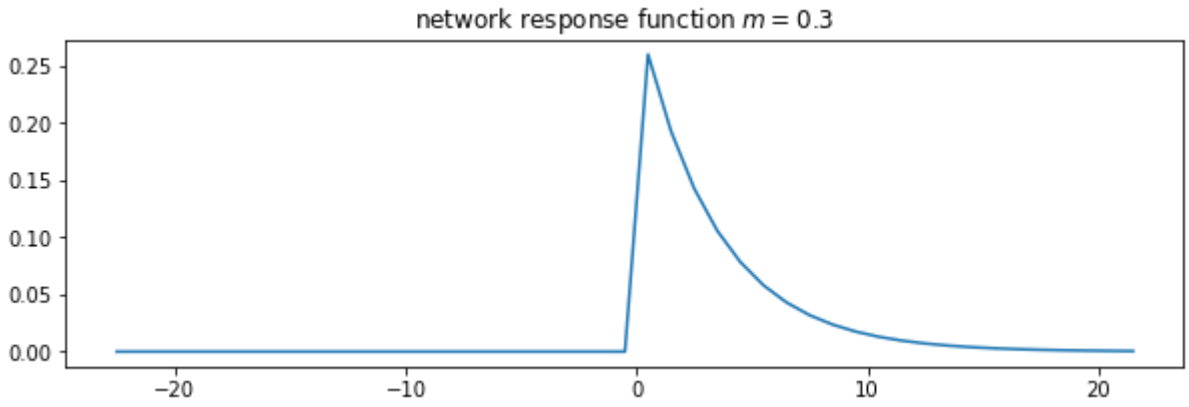
Whilst the basic shape of the above model is correct there is a problem with how the model releases water, upon breach of the threshold value water is very quickly released in spikes which is rather unlike the more steady melt exhibited in section 3.4. As given in the practical instructions a NRF function can be built into the model to effectively delay the effect of water entering the system as well as applying a smoothing effect. As the model only has so much water to work with in this case the sharp release of water at the beginning of spring/summer leaves little to compensate for where later pulses should occur.

The network response function (nrf) pictured below is given by the equation:

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

If the snowmelt runoff is an input to the time series then convolution with the NRF will produce a corrected downstream hydrograph, ignoring this step is effectively cancelling out the convective delay of water in a reach. Problems with this convective factor increase with the size of catchment assuming a centrality of the collection point - more in-depth studies have sought to use a spatially distributed NRF over high-resolution data to account for the difference in transport time over the study area (Gong et al, 2009) (https://doi.org/10.1016/j.jhydrol.2009.02.007).

In [241]:
```
# Creating a function to add delay to the model as given in practical instructions.

def decay(m, flow, plot=False):

    """
    A Function add network convective response to modelled discharge.

    Arguments:

    m - Decay parameter, to be optimised.

    flow - Modelled discharge.

    Returns:

    model_flow_nrf - Convolved modelled discharge.

    """
    # decay parameter
    #nrf = exp(-m x)
    #m  = 0.03

    # window size
    ndays = 15 * int(1/m)
    nrf_x = np.arange(ndays) - ndays/2

    # function for nrf
    nrf = np.exp(-m*nrf_x)
    nrf[nrf_x<0] = 0

    # normalise so that sum is 1
    nrf = nrf/nrf.sum()

    if plot == True:
        plt.figure(figsize=(10,3))
        # plot
        plt.plot(nrf_x,nrf)
        plt.title(f'network response function $m={m}$')

    # convolve NRF with data
    model_flow_nrf = scipy.ndimage.filters.convolve1d(flow, nrf)

    return model_flow_nrf

NRF_model = decay(0.3, model05, plot=True)
```



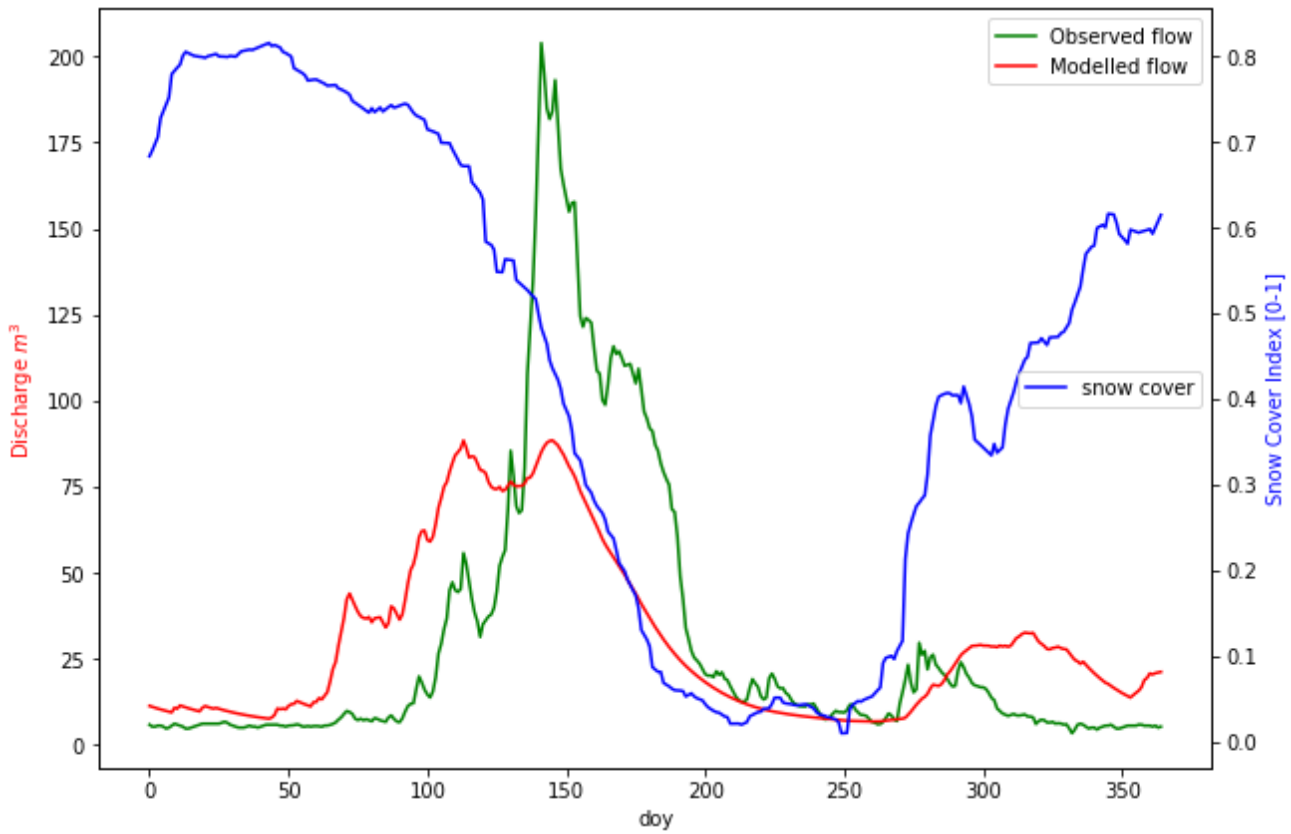
3.7 Combining Flow Model With NRF:

In [239]:
```
def combined_model(t_thresh, m, data):
    """
    A Function to combine the flow_model and decay functions into one.

    Arguments:

    t_thresh - The temperature threshold value, to be optimised.

    m - Decay parameter, to be optimised.

    data - The original data dictionary containing arrays of data.

    Returns:

    cmodel - Convolved modelled discharge.

    """
    flow = flow_model(t_thresh, data)
    decay_flow = decay(m, flow)

    return cmodel
```

3.8 Model Testing:

In [252]:
```
# Creating an initialisation for the model using 2005 as the calibration year.
# Setting parameters for t_thresh (given in literature) and m.

model05 = combined_model(5.5,  0.05, data05)

#plt.figure(figsize=(10,7))

fig, ax1 = plt.subplots(figsize=(10, 7))

ax1.set_xlabel('doy')
ax1.set_ylabel('Discharge $m^3$', color='r')

ax1.plot(x, data05["flow"],'g',label='Observed flow')
ax1.plot(x, model05,'r',label='Modelled flow')
plt.legend(loc='best')

ax2 = ax1.twinx()

ax2.set_ylabel('Snow Cover Index [0-1]', color='b')
ax2.plot(x, data05["snow"],'b',label='snow cover')

plt.legend(loc='right')
```

Out[252]: <matplotlib.legend.Legend at 0x7fef7f47e470>



At this point the NRF has smoothed the data significantly and has moved the incidence of snowmelt to be slightly more accurately aligned with the observations. However, what is needed is the optimisation of the input parameters as even with given values from literature it is very difficult to approximate the values which will give the best results.

3.9 Cost Function & Optimization:

In [257]:
```
# Starting values of T-thresh = 5.5, m = 0.05

t_thresh = 5.5
m = 0.05

# Creating an array for the parameters to sit in, this is important to build compatab
ility with the scy.py optimisation
# procedure.

parameters = (t_thresh, m)
p = np.asarray(parameters)

# combined_model(t_thresh, m, discharge06, temp06, snow06)

def cost_function(p, data, func=combined_model):

    """
    A Function to ascertain the total residual value between model and observations.

    Arguments:

    p - Parameter array.

    data - The original data dictionary containing arrays of data.

    Func - The combined model function.

    Returns:

    cost - The cost of the model in regards to modelled - observations.

    """
    # Creating the cost function using the mathematics laid out in section 2.2:

    y_pred = func(p[0], p[1], data)
    y_obs = data["flow"]
    cost = ((y_obs - y_pred)**2)*0.5

    return cost.sum()

cost = cost_function(p, data05)

print(f"The value of the cost function with T_thresh({p[0]}) & m({p[1]}): {cost}")
```

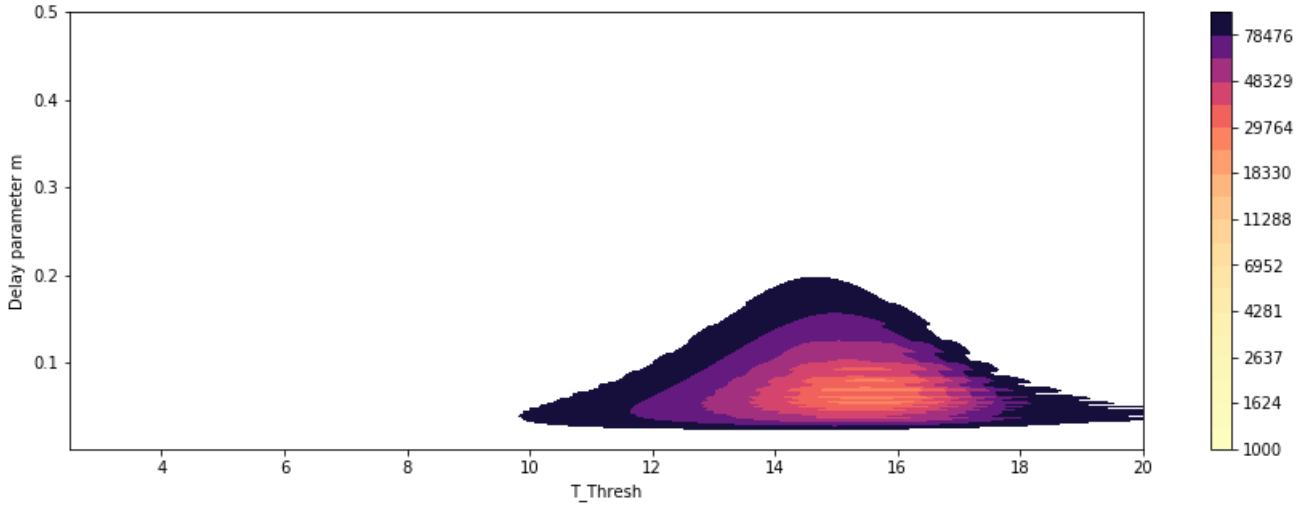
```
The value of the cost function with T_thresh(5.5) & m(0.05): 172119.55544918997
```

3.10 Searching Parameter Space:

With the above cost function built we can search parameter space for a solution, this can be guided using techniques like Markov-Chain Monte-Carlo to search for a solution directly but the below method will search an area unguided instead:

In [267]:
```
# Using method from Chapter 5
# Creating an extremely large grid of 250x250 as this operation will only be performe
d once at this scale
# and it will be much more interesting at higher resolutions

# Define a 2D array for the sum of squares (sos)
sos = np.zeros((250, 250))


# first loop is over parameter 0 (tt), 250 steps between 2.5 (lower bound of literatu
re) and 20.
for ii, p0 in enumerate(np.linspace(2.5, 20, 250)):
    # 2nd loop is over parameter 1 (m), 250 steps between 0.001 and 0.5.
    for jj, p1 in enumerate(np.linspace(0.001, 0.5, 250)):
        # for the current values of p calculate the residual
        parameters = (p0, p1)
        p = np.asarray(parameters)
        residual = cost_function(p, data05)

        sos[jj, ii] = residual

# Plotting results:

plt.figure(figsize=(15, 5))
yy = np.linspace(2.5, 20, 250)
xx = np.linspace(0.001, 0.5, 250)

c = plt.contourf(yy, xx, sos, np.logspace(3, 5, 20), cmap=plt.cm.magma_r)

plt.xlabel('T_Thresh')
plt.ylabel('Delay parameter m')
plt.colorbar()
```

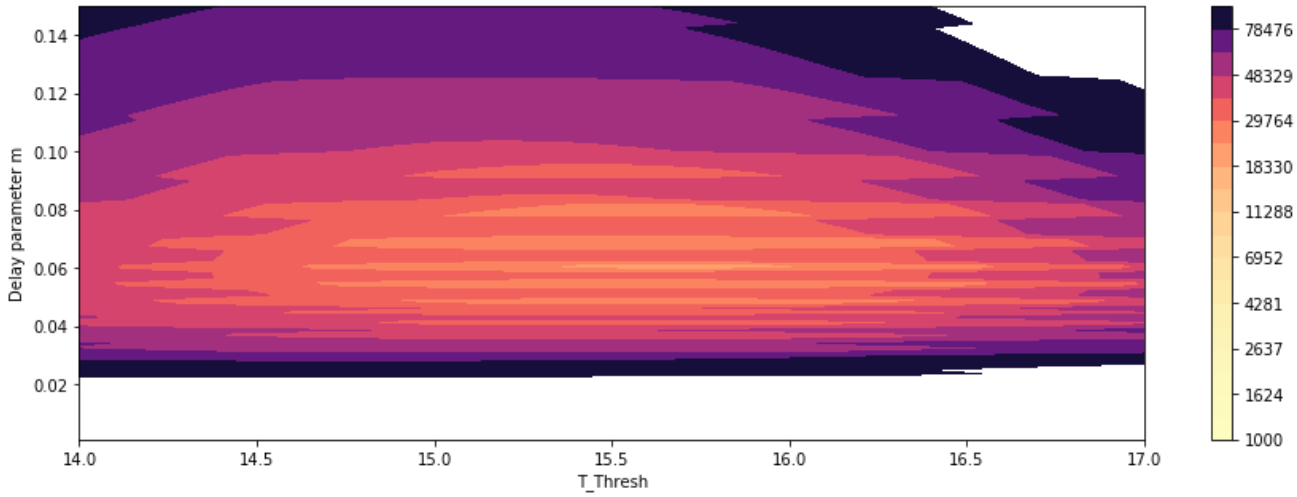
Out[267]:
```
<matplotlib.colorbar.Colorbar at 0x7fef7ff8d470>
```



So it appears that the solution lies within the bounds 14-17 T_thresh and 0-0.15 m. Searching the space within these bounds should give a quality estimate for the optimal parameters:

In [268]:
```
# Define a 2D array for the sum of squares (sos)
sos = np.zeros((100, 100))

# first loop is over parameter 0 (tt), 100 steps between 2.5 (lower bound of literatu
re) and 20.
for ii, p0 in enumerate(np.linspace(14, 17, 100)):
    # 2nd loop is over parameter 1 (m), 100 steps between 0.001 and 0.5.
    for jj, p1 in enumerate(np.linspace(0.001, 0.15, 100)):
        # for the current values of p calculate the residual
        parameters = (p0, p1)
        p = np.asarray(parameters)
        residual = cost_function(p, data05)

        sos[jj, ii] = residual

# Plotting results:

plt.figure(figsize=(15, 5))
yy = np.linspace(14, 17, 100)
xx = np.linspace(0.001, 0.15, 100)

c = plt.contourf(yy, xx, sos, np.logspace(3, 5, 20), cmap=plt.cm.magma_r)

plt.xlabel('T_Thresh')
plt.ylabel('Delay parameter m')
plt.colorbar()
```

Out[268]: <matplotlib.colorbar.Colorbar at 0x7fef84f52198>



At this magnified resolution it appears as though the solution lies at 15.75 T_thresh and 0.06 m.

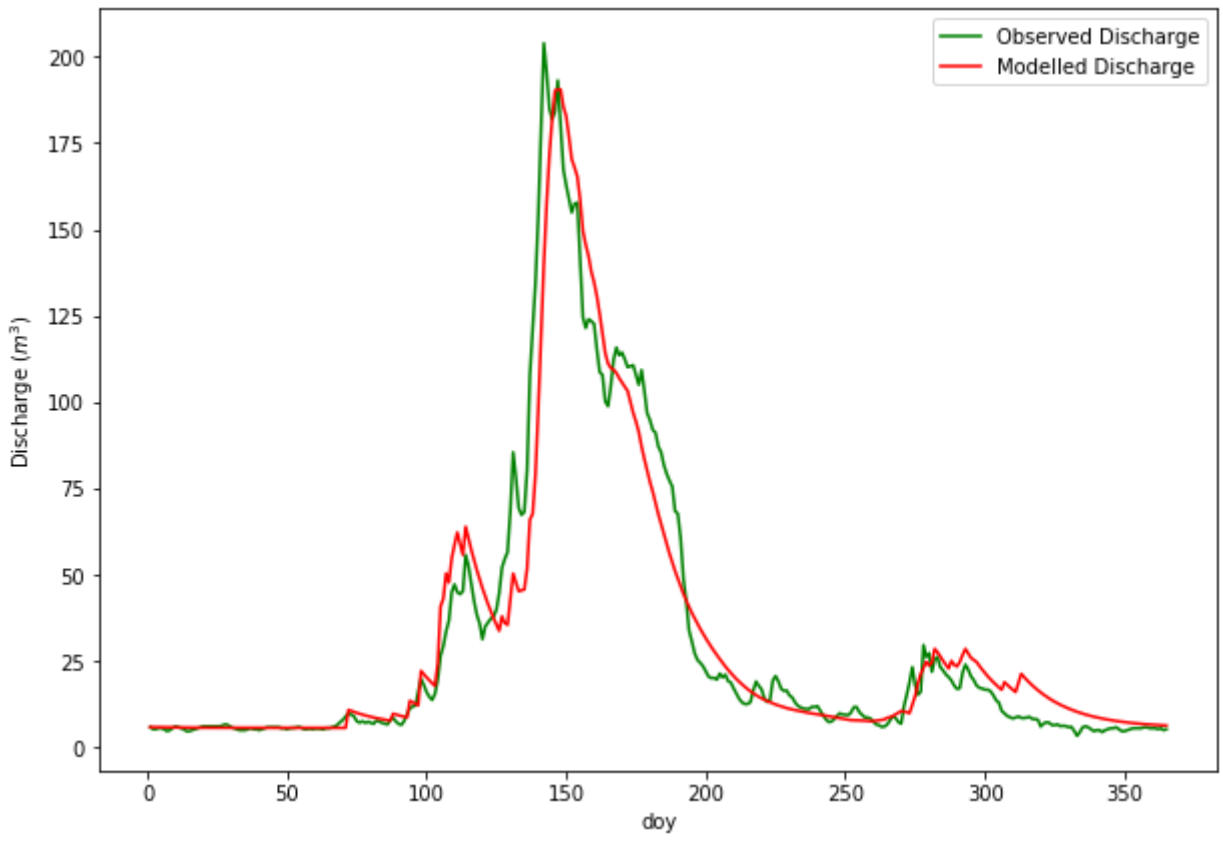
3.11 Optimization of Parameters:

In [275]:
```
# Using the starting values of the parameters roughly resolved above:

t_thresh = 15.75
m = 0.06

parameters = (t_thresh, m)
p = np.asarray(parameters)

# Creating bounds for the parameters before minimization.
# This is important because the function will collapse if m is allowed to become nega
tive or greater than 1.
bounds = ((None, None), (0.0001, 0.999))

# Using the scy.py minimize function to find the solution.
solution = minimize(cost_function, p, args=(data05), bounds=bounds)

print(f'The cost value at the solution is: {solution.fun}')
print(f'The parameter values at the solution are: {solution.x}')
```

```
The cost value at the solution is: 22609.357213800802
The parameter values at the solution are: [15.69761153  0.06072956]
```

In [277]:
```
# Creating the figure and axis labels.
plt.figure(figsize=(10,7))
plt.xlabel('doy')
plt.ylabel('Discharge ($m^3$)')

# plot data
plt.plot(x_axis, data05["flow"],'g',label='Observed Discharge')
plt.plot(x_axis, combined_model(15.69761153, 0.06072956, data05),'r',label='Modelled
 Discharge')


plt.legend()
```

Out[277]:
```
<matplotlib.legend.Legend at 0x7fef7f9015c0>
```



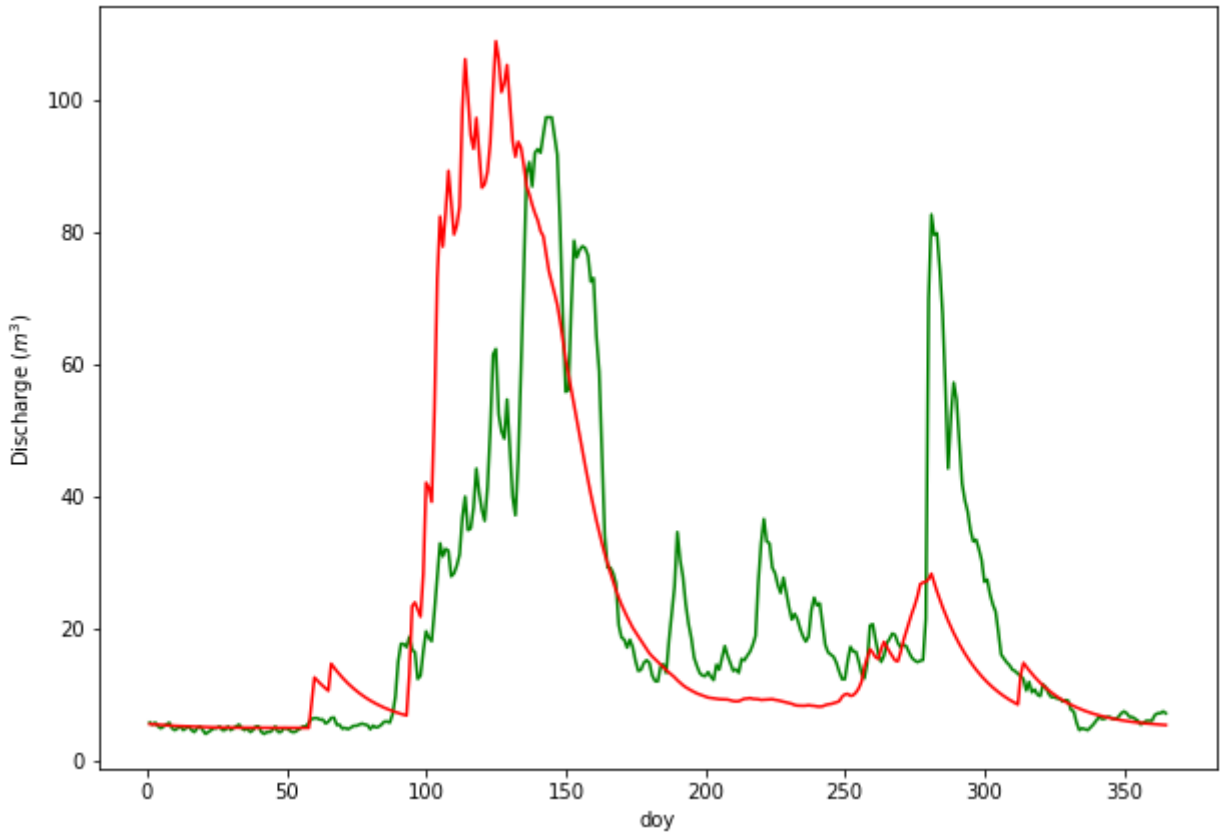
The optimised parameters show a really accurate fit to the data, although not perfect with the simplicity of the model this is a very promising result.

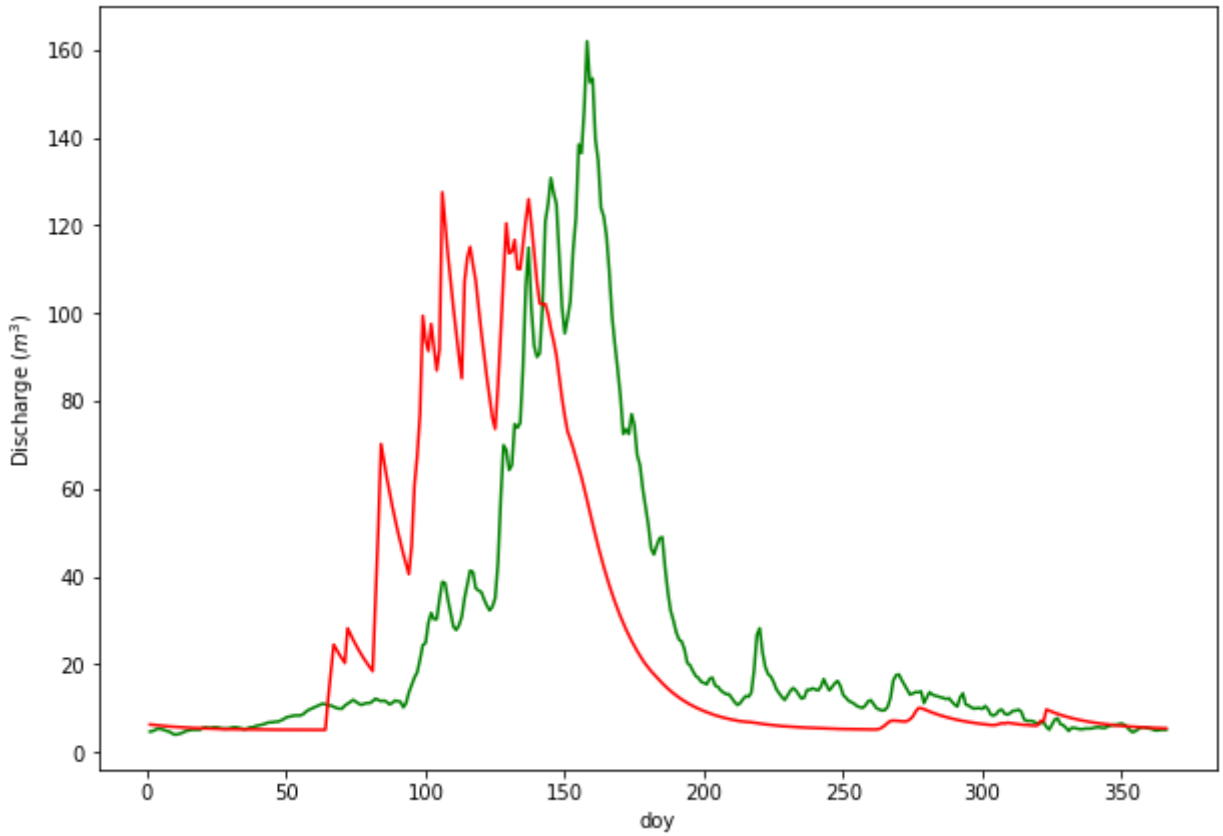
3.12 Validation of Model & Results:

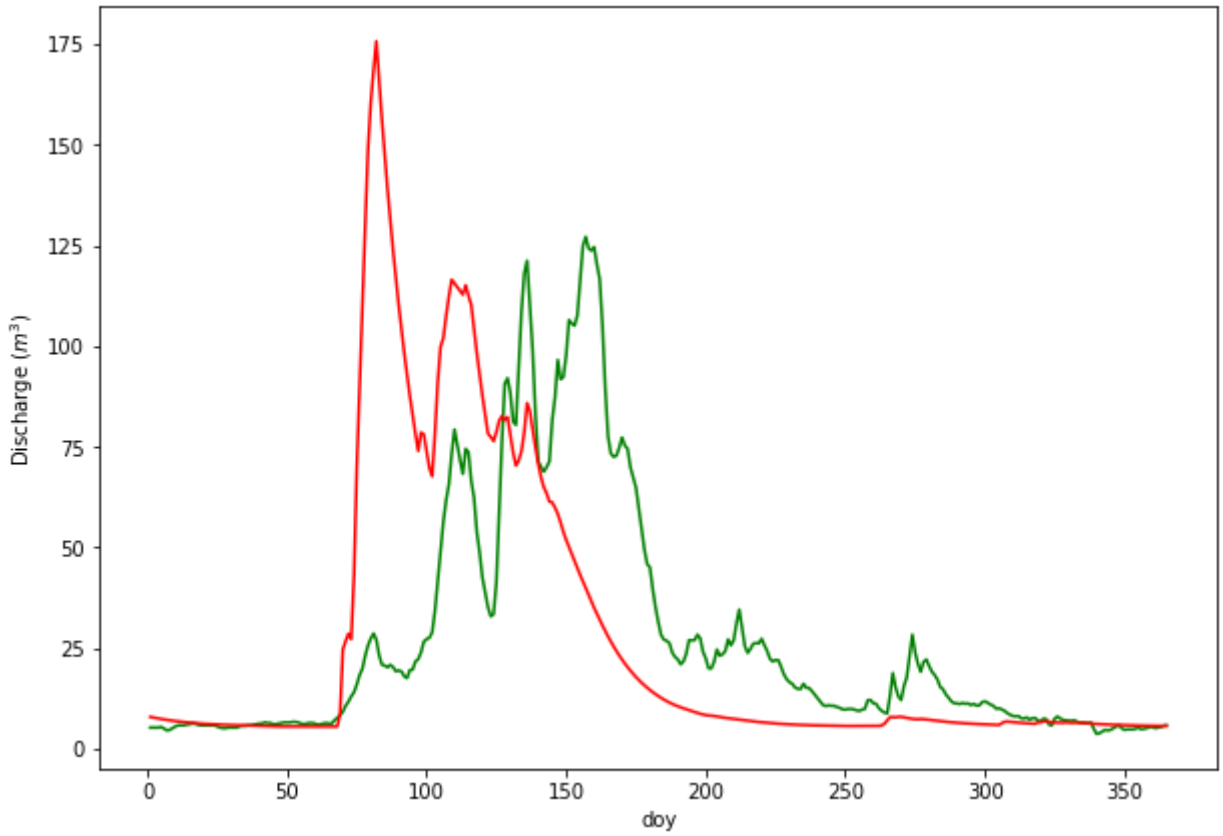
With the optimised parameters now available it is possible to use these to calculate the effectiveness of the model for the validation years.

In [279]:
```
t_thresh = 15.69761153
m = 0.06072956

parameters = (t_thresh, m)
p = np.asarray(parameters)

def validation(data):
    model = combined_model(t_thresh, m, data)
    cost = cost_function(p, data)
    print(f'The cost value is: {cost}')

    # Creating the figure and axis labels.
    plt.figure(figsize=(10,7))

    plt.xlabel('doy')
    plt.ylabel('Discharge ($m^3$)')
    if len(data["flow"]) < 366:
        # plot data
        plt.plot(x_axis, data["flow"],'g',label='Observed Discharge')
        plt.plot(x_axis, model,'r',label='Modelled Discharge')
    else:
        # plot data
        plt.plot(x_axisl, data["flow"],'g',label='Observed Discharge')
        plt.plot(x_axisl, model,'r',label='Modelled Discharge')

validation(data06)
validation(data16)
validation(data17)
```

```
The cost value is: 69284.26083246406
The cost value is: 158335.13517048932
The cost value is: 225413.02105678953
```







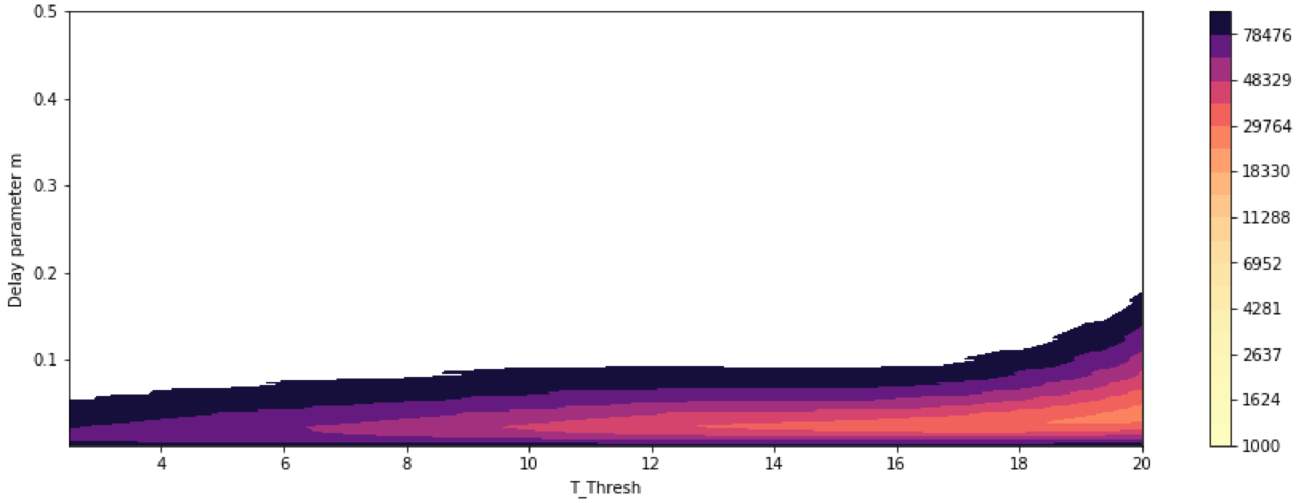
The parameters do not give a particularly accurate model for any of the other years although the model fits 2006 the best, considering how this year differed in behavior so drastically from the others this is surprising.

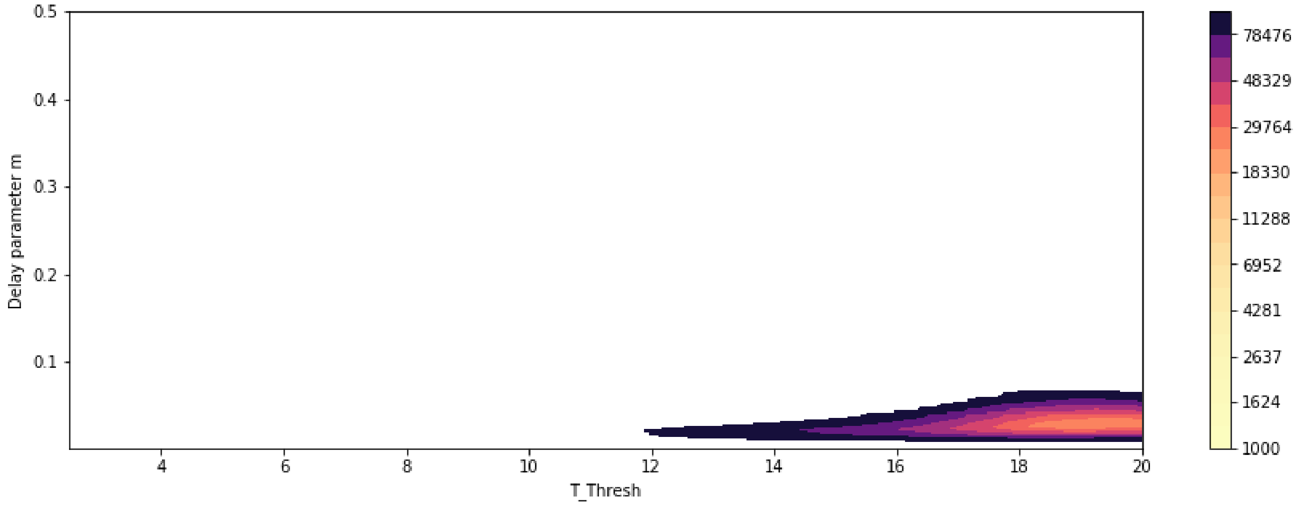
Before moving to analysis I will use the previously constructed functions to resolve the parameter solutions for different validation years to ascertain what the difference may be.

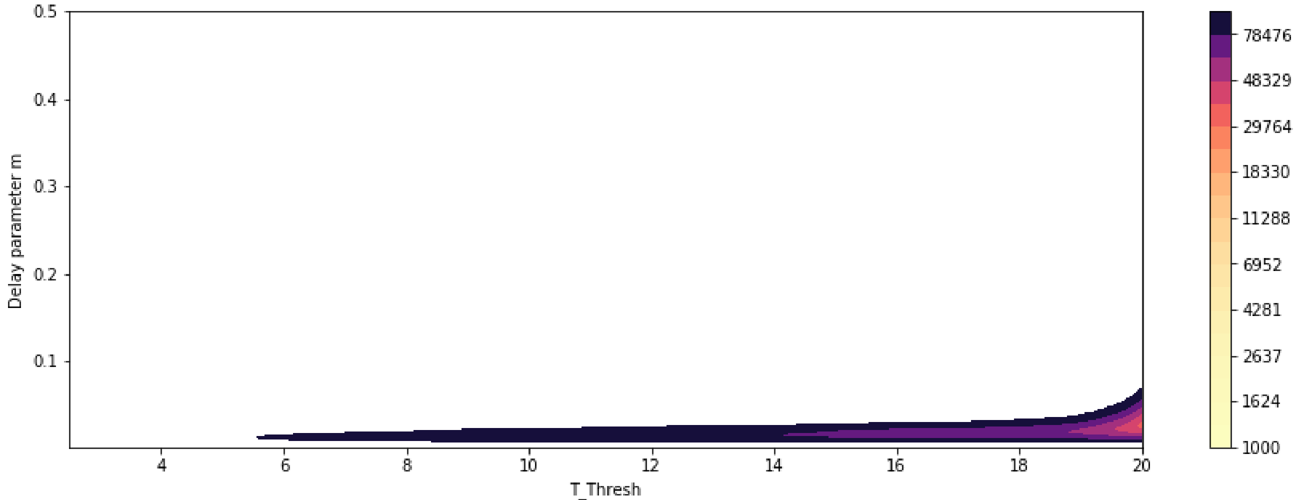
In [295]:
```
bounds = ((0, 20), (0.0001, 0.999))

t_thresh = 15.69761153
m = 0.06072956

parameters = (t_thresh, m)
p = np.asarray(parameters)

# Performing operations
#solution06 = minimize(cost_function, p, args=(data06), bounds=bounds)
#solution16 = minimize(cost_function, p, args=(data16), bounds=bounds)
solution17 = minimize(cost_function, p, args=(data17), bounds=bounds)

# Printing results
print("2006:")
print(f'Was the operation successful: {solution06.success}')
print(f'The cost value at the solution is: {solution06.fun}')
print(f'The parameter values at the solution are: {solution06.x}')
print("**************************************************")
print("2016:")
print(f'Was the operation successful: {solution16.success}')
print(f'The cost value at the solution is: {solution16.fun}')
print(f'The parameter values at the solution are: {solution16.x}')
print("**************************************************")
print("2017:")
print(f'Was the operation successful: {solution17.success}')
print(f'The cost value at the solution is: {solution17.fun}')
print(f'The parameter values at the solution are: {solution17.x}')
```

```
2006:
Was the operation successful: False
The cost value at the solution is: 27306.76692321018
The parameter values at the solution are: [20.          0.03748531]
**************************************************
2016:
Was the operation successful: False
The cost value at the solution is: 23421.98195429478
The parameter values at the solution are: [19.29757334  0.02941674]
**************************************************
2017:
Was the operation successful: True
The cost value at the solution is: 164874.6080751947
The parameter values at the solution are: [1.16666685e+01 1.00000000e-04]
```

In [299]:
```
# Creating a function using the parameter space exploration technique above to check
 why a solution
# is not reached in the above minimisations.

def parspace(data):
    # Define a 2D array for the sum of squares (sos)
    sos = np.zeros((250, 250))


    # first loop is over parameter 0 (tt), 250 steps between 2.5 (lower bound of lite
rature) and 20.
    for ii, p0 in enumerate(np.linspace(2.5, 20, 250)):
        # 2nd loop is over parameter 1 (m), 250 steps between 0.001 and 0.5.
        for jj, p1 in enumerate(np.linspace(0.001, 0.5, 250)):
            # for the current values of p calculate the residual
            parameters = (p0, p1)
            p = np.asarray(parameters)
            residual = cost_function(p, data)

            sos[jj, ii] = residual

    # Plotting results:

    plt.figure(figsize=(15, 5))
    yy = np.linspace(2.5, 20, 250)
    xx = np.linspace(0.001, 0.5, 250)

    c = plt.contourf(yy, xx, sos, np.logspace(3, 5, 20), cmap=plt.cm.magma_r)

    plt.xlabel('T_Thresh')
    plt.ylabel('Delay parameter m')
    plt.colorbar()

parspace(data06)
parspace(data16)
parspace(data17)
```







4.0 Discussion & Conclusions:

In regards to the validation procedure of 3.12 it appears that the model itself had a mediocre performance, whilst the cost value was kept low for 2006 the model was quite inaccurate for the years 2016 and 17. The resolved parameters are quite suspect as the T_thresh value at ~15 is much higher than what should be considered scientifically accurate and does not match those taken from previous studies which were in the region of 5 degrees (Hock, 2003) (https://doi.org/10.1016/S0022-1694(03)00257-9). The simplicity of the model has left optimisation open to this sort of failiure as although the parameters fit the training set well they do not translate to validation particularly convincingly suggesting that some of the core system dynamics have been lost.

In the latter years the functionality of the model was broken as both saw discharge peak far too early suggesting that the error lay in the m value calculated poorly manipulating the NRF convolution. Gong et al. (2009) (https://doi.org/10.1016/j.jhydrol.2009.02.007) writes that the NRF convolution when used correctly has the ability to fit data very precisely, however this study utilised spatial adaption and tried various algorithms to find an approach which accurately represented runoff in the catchment. It is likely that the lack of resolution in the approach of this report meant that the results did not translate accurately, if nrf was calculated with the distribution of the snow cover in the catchment combined with the relative transport time to the data point there may have been an increased level of accuracy. It is likely that the sensitivity of the NRF technique has had some effect, as the optimal parameter given is such a small number the parameter holds a high degree of uncertainty as a little change can have drastic effects on the results. The resolved solutions for each of the validation years although not successful are less than half that of the optimised value from 2005 (1.00000000e-04, 0.02941674, 0.03748531 compared to 0.06072956).

There is also the possibility that the data itself for these given years did not lend itself kindly to the technique, searching parameter space for optimisations for each of the validation years was inconclusive. Cost never really dropped within the set bounds for any of the years suggesting (in the case of 16', 17') an error in data preparation of a lack of comprehensive coverage in the model for the specific scenarios the data presents. It must also be noted that there is a difference in 10 years between the validation year of 2005 and then calibration datasets from 2016, 2017, mountainous regions are particularly susceptible to changes in climate there is some possibility that weather patterns have been altered in the area between this time or larger weather patterns such as ENSO may be in play.

There is a likelihood that the seasonality and variability in the data for each year has had some effect on the propensity of the model to cope with annual changes when given set parameters. There are certainly variables which are unaccounted for such as the actual snowpack water equivalent is unknown and measurements of snow cover can be misleading as there is no indication of depth or composition, ice storing a larger volume of water in space and melting at a higher threshold will also have some effect. Energy and temperature variations in snowpack can be very complex and without accurate controls on contributing factors such as the temporal and spatial variability of snow depth (which in practicality is not always available) modelling may fall short (Zeinivand, 2009) (https://doi.org/10.1007/s11269-008-9381-2). Additionally, precipitation events go unaccounted for which could influence both the training dataset and the discharge totals used in each subsequent model.

Overall, there are a large number of contributing factors which have upset model performance although calibration was successful for the training dataset it is clear that the resolved parameters are overfit to this data and do not provide a good performence in validation. It is the inherent uncertainty in the m parameter combined with a lack of appropriate scientific transformation in regards to the T thresh value which make this model unable to handle multiple datasets well with fixed parameters.

References:

(In order of appearance)

1. Shalamu Abudu, Chun-liang Cui, Muattar Saydi, James Phillip King, Application of snowmelt runoff model (SRM) in mountainous watersheds: A review, Water Science and Engineering, Volume 5, Issue 2, 2012, Pages 123-136, ISSN 1674-2370, https://doi.org/10.3882/j.issn.1674-2370.2012.02.001 (https://doi.org/10.3882/j.issn.1674-2370.2012.02.001).

2. R. I. Ferguson (1999). Snowmelt runoff models. Progress in Physical Geography: Earth and Environment, 23(2), 205–227. https://doi.org/10.1177/030913339902300203 (https://doi.org/10.1177/030913339902300203)

3. Regine Hock, Temperature index melt modelling in mountain areas, Journal of Hydrology, Volume 282, Issues 1–4, 2003, Pages 104-115, ISSN 0022-1694, https://doi.org/10.1016/S0022-1694(03)00257-9 (https://doi.org/10.1016/S0022-1694(03)00257-9).

4. F. Cazorzi, G. Dalla Fontana, Snowmelt modelling by combining air temperature and a distributed radiation index, Journal of Hydrology, Volume 181, Issues 1–4, 1996, Pages 169-187, ISSN 0022-1694, https://doi.org/10.1016/0022-1694(95)02913-3 (https://doi.org/10.1016/0022-1694(95)02913-3).

5. L. Gong, E. Widén-Nilsson, S. Halldin, C.-Y. Xu, Large-scale runoff routing with an aggregated network-response function, Journal of Hydrology, Volume 368, Issues 1–4, 2009, Pages 237-250. https://doi.org/10.1016/j.jhydrol.2009.02.007 (https://doi.org/10.1016/j.jhydrol.2009.02.007).

6. Zeinivand, H. & De Smedt, F. Water Resour Manage (2009) 23: 2271. https://doi.org/10.1007/s11269-008-9381-2 (https://doi.org/10.1007/s11269-008-9381-2)