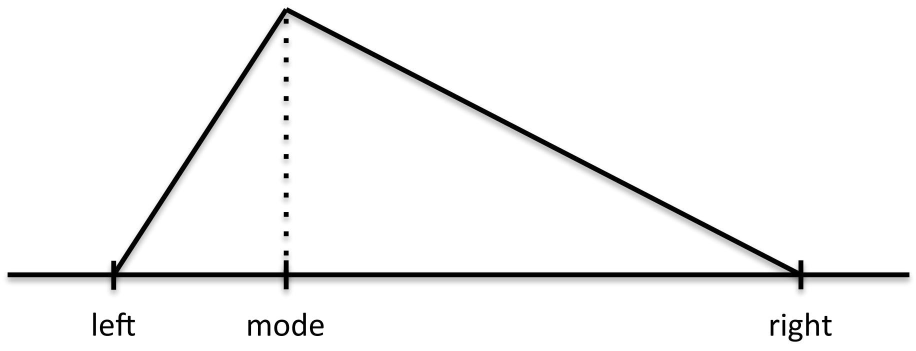
Homework 7 **PSY4219/6219**
Due November 2 **Fall 2022**
18 points

As with all homework assignments, we expect you to use comments, use docstrings for functions, use meaningful variable and function names, use vectorized operations when possible, create functions to modularize your code, make sure figures are labeled thoroughly and formatted appropriately, use the Pythonic (object-oriented) rather than Matlab (stateful) approach to creating figures and axes, and follow the other guidelines and best practices discussed in class.

You can complete this homework assignment in a Jupyter notebook or a Python .py file. Please use the HW7.ipynb or HW7.py files provided on Brightspace as a starting point for your work, following the structure embedded in those files.

Q1. This question will ask you to implement in Python some of the random number generation methods we discussed in class. In class, we talked about random number generators for uniform distributions, normal distributions, and exponential distributions. We also talked about creating random number generators for arbitrary distributions. The triangular distribution is just one possible distribution out there in the universe of possible probability distributions.



left defines the lower limit, right defines the upper limit, and mode defines the peak of the distribution. Like every probability distribution, the area under the distribution is 1.0.

Use the following parameters for this assignment, but make sure your code works for any values of these parameters; your code must be organized so that I only need to change left, mode, and right in one place in your code:

```
left = 1
mode = 2
right = 5
```

Your code will be tested by trying different values of `left`, `mode`, and `right`.

(a) (2 point) Plot the triangular distribution (its probability density function, pdf, as illustrated above). The plot should be done within a function. Please use the Pythonic (object-oriented) rather than Matlab (stateful) approach to creating figures and axes. Again, remember that your code must work for any values of `left`, `mode`, and `right`.

To create this plot, you will need to write a function that returns the value of the pdf given a particular value (this function will also be useful in the parts c and d). You can try to figure out the equations for the pdf from the geometry on your own so that the area under the probability density function (pdf) is equal to 1, or you can simply find the equations for the probability density function (pdf) for a triangular distribution online (Wikipedia or elsewhere); note that there is a built-in pdf function in scipy in Python for the triangular distribution, but it assumes `left = 0` and `right = 1`.

(b) (1 point) Look within the numpy documentation to find the built-in random number generator for the triangular distribution. Generate 1000 random samples from the triangular distribution. Create a histogram from these samples following the examples shared in class; creating the histogram should be done by a function that can be used in parts c and d below. The histogram should look a little like the plot of the probability distribution you created for part 1 (you might confirm that for yourself by running it with 10x or 100x the number samples, but I only want to run your code for 1000 when we test your code).

(c) (5 points) Now create your own random number generator for the triangular distribution using the rejection sampling method we discussed in class. Generate 1000 random samples, create a histogram, and a plot, just as you did for part b. Again, make sure your code works for any valid values of left, mode, and right. This should be implemented as a function modeled after the built-in numpy prng used in (b), with arguments defining the shape of the triangular distribution and the number of random numbers to generate, returning an array of random samples as a numpy array.

(d) (5 points) Now create your own random number generator for the triangular distribution using the Metropolis (MCMC) algorithm we discussed in class. Generate 1000 random samples, create a histogram, and a plot, just as you did for parts b and c. Again, make sure your code works for any valid values of left, mode, and right. This should be implemented as a function modeled after the built-in numpy prng used in (b), with arguments defining the shape of the triangular distribution and the number of random numbers to generate, returning an array of random samples as a numpy array; it will need to include optional arguments specifying the burn-in and thinning (described below).

I outlined the Metropolis algorithm in class. I will formalize it here so as to roughly follow the text that is on the Wikipedia page for Metropolis-Hastings (a generalization of the Metropolis algorithm).

Metropolis algorithm (symmetric proposal distribution)

Let *p(x)* be the desired probability distribution, which in this case is the triangular distribution (note that the Wikipedia page formalizes using *f(x)*, a function proportional to *p(x)*, which is the approach used, for example, to generate random numbers for a Bayesian posterior distribution).

1. Initialization:
   - Choose an arbitrary point *x0* to be the first sample. The first point in the chain *xt* will be equal to *x0*. Since you know the shape of the triangular distribution, it makes sense to pick a first sample *x0* that is within the range of the distribution.
   - Choose a proposal distribution *g(xp|xt)*, where *xt* is the current point in the chain and *xp* is the proposed point in the chain. For the Metropolis algorithm, the proposal distribution needs to be symmetric (Metropolis-Hastings generalizes to allow for non-symmetric proposal distributions). The simplest choice for *g(xp|xt)* is to assume that *xp* is drawn from a normal distribution with mean *xt* and standard deviation *sigma*; *sigma* should be small, but not too small.

2. For each iteration:
   - Generate: Generate a candidate *xp* from *g(xp|xt)* using the above.
   - Calculate: Calculate the acceptance ratio *A = p(xp)/p(xt)*, which is used to decide whether to accept or reject the proposed candidate.
   - Accept or Reject:
     - Generate a uniform random number *U* on [0,1).
     - If *U <= A* accept the proposed candidate by setting *xt = xp*.
     - If *U > A* reject the proposed candidate.

Each new *xt* produced on each iteration is a candidate random number produced by the Metropolis algorithm.

Use a burn-in of 100 samples. That means that you throw away the first 100 iterations of the Metropolis algorithm.

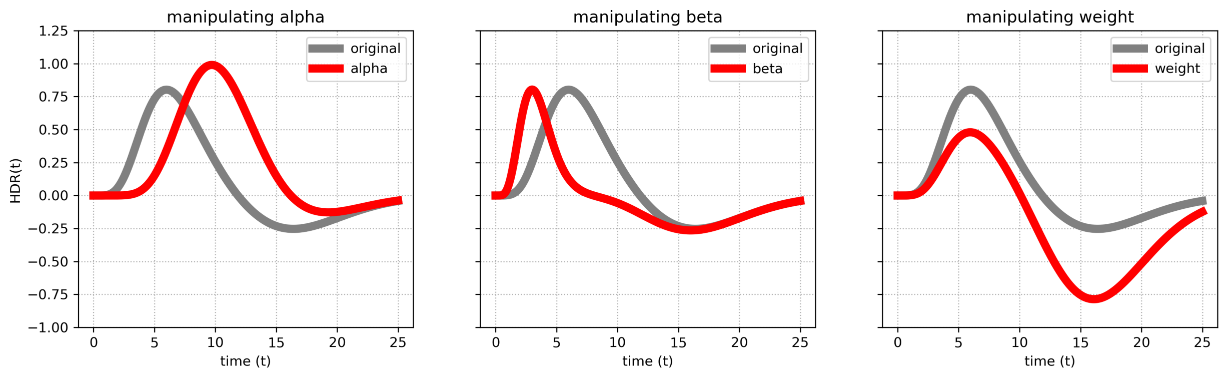
Use the Metropolis algorithm to generate 1000 random samples from the triangular distribution. Because the Metropolis algorithm produces highly correlated samples (it is a random walk) I want you to thin, as discussed in class, such that you keep only 1 of every 50 *xt* values produced by the Metropolis algorithm (you throw away 49 of every 50 sample). Your value of thinning should be a variable that can be adjusted in your code.

Q2 (5 points). Create a figure that is an array of plots using matplotlib to illustrate how changing values of the parameters α_1, β_1, and w (holding α_2 and β_2 fixed) changes the shape of the Hemodynamic Response (HDR) function you used in past homework assignments.

$$HDR(t) = w\left(\frac{\beta_1^{\alpha_1} t^{\alpha_1} e^{-t\beta_1}}{\Gamma(\alpha_1)}\right) - (1-w)\left(\frac{\beta_2^{\alpha_2} t^{\alpha_2} e^{-t\beta_2}}{\Gamma(\alpha_2)}\right)$$

Obviously, you will need to create a Python function that computes the HDR function. For the default (original) values for the parameters, assume the following:
$\alpha_1 = 6$
$\beta_1 = 1$
$\alpha_2 = 16$
$\beta_2 = 1$
$w = 5/6$
Your Python function should allow for default arguments (as discussed in class) that default to these specific values if no argument values are passed.

Your resulting array of plots should look as close as possible to this:



Each individual plot within array of three plots will need to be created using the same function (in other words, that function will need to be called three times). That function will need to take as arguments the numpy arrays defining the two curves within the plot as well as any other information you need to properly label and format each subplot (you will also need to pass the matplotlib axis on which to place the plot). I do not want you to simply copy and paste the same code for each plot. Please use the Pythonic (object-oriented) rather than Matlab (stateful) approach to creating figures and axes.

The first plot shows the HDR function using original values as a grey line and the HDR using a different value of α_1 as a red line. The second plot shows the same HDR using original values as a grey line and the HDR using a different value of β_1. The third plot shows the original and a different value of w.

You will need to use `subplots()` to create the layout (and adjust some other settings of the figure to get the proportions right).

To reproduce this plot, you will need to look up how to set the axes ticks to particular values (rather than the default) and add a grid. Note that the ticks and the extent of the y axes (and hence the grid lines as well) are the same in all three plots (you will need to make sure to specify the axis ranges). Also note that the y axis labels are intentionally missing from the 2nd and 3rd plots.

You will need to play around with values of α_1, β_1, and w to find values that give a reasonably close match to the shapes of the functions shown in these plots (they don't need to be perfect, but they should be visually close). I suggest doing this after you have gotten the plots (and the array of plots) formatted appropriately.

Unexcused late assignments will be penalized 10% for every 24 hours late, starting from the time class ends, for a maximum of two days, after which they will earn a 0.