Tag Archives: zero-order hold

MAY 21 2017

1 COMMENT
LINEAR ALGEBRA, MATH, NENGO, PYTHON

Improving neural models by compensating for discrete rather than continuous filter dynamics when simulating on digital systems

This is going to be a pretty niche post, but there is some great work by Aaron Voelker (http://compneuro.uwaterloo.ca/people/aaron-r-voelker.html) from my old lab that has inspired me to do a post. The work is from an upcoming paper, which is all up on Aaron's GitHub (https://github.com/arvoelke/delay2017). It applies to building neural models using the Neural Engineering Framework (NEF) (http://compneuro.uwaterloo.ca/research/nef.html). There's a bunch of material on the NEF out there already, (e.g. the book How to Build a Brain (https://www.amazon.ca/How-Build-Brain-Architecture-Biological/dp/0190262125?SubscriptionId=AKIAILSHYYTFIVPWUY6Q&tag=duckduckgo-d-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0190262125) by Dr. Chris Eliasmith, an online intro (https://pythonhosted.org/nengo/examples/nef_summary.html), and you can also check out Nengo (http://nengo.ca/download), which is neural model development software with some good tutorials on the NEF) so I'm going to assume you already know the basics of the NEF for this post.

Additionally, this is applicable to simulating these models on digital systems, which, probably, most of you are. If you're not, however! Then use standard NEF methods.

And then last note before starting, these methods are most relevant for systems with fast dynamics (relative to simulation time). If your system dynamics are pretty slow, you can likely get away with the continuous time solution if you resist change and learning. And we'll see this in the example point attractor system at the end of the post! But even for slowly evolving systems, I would still recommend at least skipping to the end and seeing how to use the library shortcuts when coding your own models. The example code is also all up on my GitHub (https://github.com/studywolf/blog/blob/master/Nengo%20scripting/Nengo%202/discrete_filter/point_attractor.py).

NEF modeling with continuous lowpass filter dynamics

Basic state space equations for linear time-invariant (LTI) systems (i.e. dynamics can be captured with a matrix and the matrices don't change over time) are:

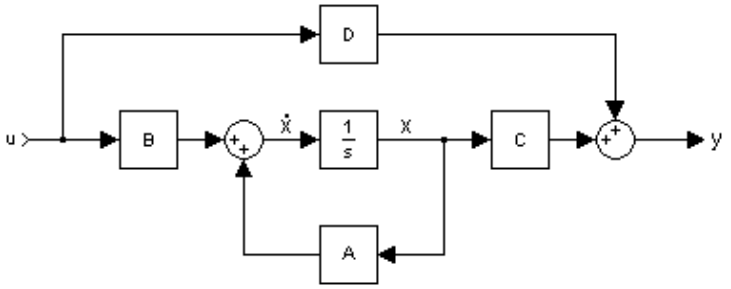
$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)$$

where

- $\mathbf{x}$ is the system state,
- $\mathbf{y}$ is the system output,
- $\mathbf{u}$ is the system input,
- $\mathbf{A}$ is called the state matrix,
- $\mathbf{B}$ is called the input matrix,
- $\mathbf{C}$ is called the output matrix, and
- $\mathbf{D}$ is called the feedthrough matrix,

and the system diagram looks like this:

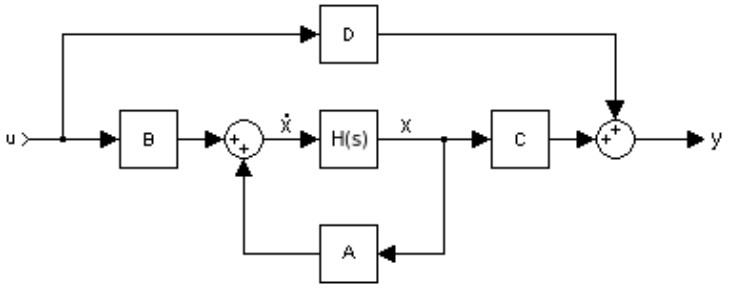


and the transfer function (https://en.wikipedia.org/wiki/Transfer_function), which is written in Laplace space and captures the system output over system input, for the system is

$$\mathbf{F}(s) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}$$

where s is the Laplace variable.

Now, because it's a neural system we don't have a perfect integrator in the middle, we instead have a synaptic filter, $H(s)$, giving:



So our goal is: given some synaptic filter $H(s)$, we want to generate some modified transfer function, $\mathbf{F}'$, such that $\mathbf{F}'(H(s))$ has the same dynamics as our desired system, $\mathbf{F}(s)$. In other words, find an $\mathbf{F}'$ such that

$$\mathbf{F}'\left(\frac{1}{H(s)}\right) = \mathbf{F}(s).$$

Alrighty. Let's do that.

The transfer function for our neural system is

$$\mathbf{F}(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(H(s)^{-1}\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}.$$

The effect of the synapse is well captured by a lowpass filter, $H(s) = \frac{1}{\tau s+1}$, making our equation

$$\mathbf{F}(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}((\tau s + 1)\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D},$$

$$\mathbf{F}(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(\tau s\mathbf{I} + \mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}.$$

To get this into a form where we can start to modify the system state matrices to compensate for the filter effects, we have to isolate $s\mathbf{I}$. To do that, we can do the following basic math jujitsu

$$\mathbf{F}(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(\tau(s\mathbf{I} + \tfrac{1}{\tau}(\mathbf{I} - \mathbf{A}))^{-1}\mathbf{B} + \mathbf{D}.$$

$$\mathbf{F}(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(s\mathbf{I} + \tfrac{1}{\tau}(\mathbf{I} - \mathbf{A})^{-1}\tfrac{1}{\tau}\mathbf{B} + \mathbf{D}.$$

OK. Now, we can make $\mathbf{F}'$ by substituting for our $\mathbf{A}$ and $\mathbf{B}$ matrices with

$$\mathbf{A}' = \tau\mathbf{A} + \mathbf{I}$$

$$\mathbf{B}' = \tau\mathbf{B}$$

then we get

$$\mathbf{F}'(H(s)) = \frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \mathbf{C}(s\mathbf{I} + \tfrac{1}{\tau}(\mathbf{I} - \mathbf{A}')^{-1}\tfrac{1}{\tau}\mathbf{B}' + \mathbf{D}.$$

$$= \mathbf{C}(s\mathbf{I} + \tfrac{1}{\tau}(\mathbf{I} - (\tau\mathbf{A} + \mathbf{I}))^{-1}\tfrac{1}{\tau}(\tau\mathbf{B}) + \mathbf{D}.$$

$$= \mathbf{C}(s\mathbf{I} + \tfrac{1}{\tau}(\tau\mathbf{A}))^{-1}\mathbf{B} + \mathbf{D}.$$

$$= \mathbf{C}(s\mathbf{I} + \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}.$$

and voila! We have created an $\mathbf{F}'$ such that $\mathbf{F}'(H(s)) = \mathbf{F}(s)$. Said another way, we have created a system $(\mathbf{A}', \mathbf{B}', \mathbf{C}' = \mathbf{C}, \mathbf{D}' = \mathbf{D})$ that compensates for the synaptic filtering effects to achieve our desired system dynamics!

So, to compensate for the continuous lowpass filter, we use $\mathbf{A}' = \tau\mathbf{A} + \mathbf{I}$ and $\mathbf{B}' = \tau\mathbf{B}$ when implementing our model and we're golden.

And so that's what we've been doing for a long time when building our models. Assuming a continuous lowpass filter and going along our merry way. Aaron, however, shrewdly noticed that computers are digital, and thusly that the standard NEF methods are not a fully accurate way of compensating for the filter that is actually being applied in simulation.

To convert our continuous system state equations to discrete state equations we need to make two changes: 1) the first is a variable change to denote the that we're in discrete time, and we'll use z instead of s, and 2) we need to calculate the discrete version our system, i.e. transform $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}) \rightarrow (\mathbf{A}_d, \mathbf{B}_d, \mathbf{C}_d, \mathbf{D}_d)$.

The first step is easy, the second step more complicated. To discretize the system we'll use the zero-order hold (ZOH) method (also referred to as discretization assuming zero-order hold).

Zero-order hold discretization

Zero-order hold (ZOH) systems (https://www.mathworks.com/help/simulink/slref/zeroorderhold.html?requestedDomain=www.mathworks.com) simply hold their input over a specified amount of time. The use of ZOH here is that during discretization we assume the input control signal stays constant until the next sample time.

There are good write ups on the derivation of the discretization both on wikipedia (https://en.wikipedia.org/wiki/Discretization#discrete_function) and in these course notes from Purdue (http://www.engr.iupui.edu/~skoskie/ECE595_f05/handouts/discretization.pdf). I mostly followed the wikipedia derivation, but there were a few steps that get glossed over, so I thought I'd just write it out fully here and hopefully save someone some pain. Also for just a general intro I found these slides (http://www.pages.drexel.edu/~pyo22/mem640/lecture03-Discrete/mem640Lecture-Discrete.pdf) from Paul Oh at Drexel University really helpful.

OK. First we'll solve an LTI system, and then we'll discretize it.

So, you've got yourself a continuous LTI system

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$$

and you want to solve for $\mathbf{x}(t)$. Rearranging things to put all the $\mathbf{x}$ on one side gives

$$\dot{\mathbf{x}}(t) - \mathbf{A}\mathbf{x}(t) = \mathbf{B}\mathbf{u}(t).$$

Looking through our identity library to find something that might help us here (after a long and grueling search) we come across:

$$\frac{\partial}{\partial t}\mathrm{e}^{\mathbf{A}t} = \mathbf{A}\mathrm{e}^{\mathbf{A}t} = \mathrm{e}^{\mathbf{A}t}\mathbf{A}.$$

We now left multiply our system by $\mathrm{e}^{-\mathbf{A}t}$ (note the negative in the exponent)

$$\mathrm{e}^{-\mathbf{A}t}\dot{\mathbf{x}}(t) - \mathrm{e}^{-\mathbf{A}t}\mathbf{A}\mathbf{x}(t) = \mathrm{e}^{-\mathbf{A}t}\mathbf{B}\mathbf{u}(t).$$

Looking at this carefully, we identify the left-hand side as the result of a chain rule, so we can rewrite it as

$$\frac{\partial}{\partial t}(\mathrm{e}^{-\mathbf{A}t}\mathbf{x}(t)) = \mathrm{e}^{-\mathbf{A}t}\mathbf{B}\mathbf{u}(t).$$

From here we integrate both sides, giving

$$\mathrm{e}^{-\mathbf{A}t}\mathbf{x}(t) - \mathrm{e}^{0}\mathbf{x}(0) = \int_0^t \mathrm{e}^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau,$$

$$\mathrm{e}^{-\mathbf{A}t}\mathbf{x}(t) = \int_0^t \mathrm{e}^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau + \mathbf{x}(0).$$

To isolate the $\mathbf{x}(t)$ term on the left-hand side now multiply (https://en.wikipedia.org/wiki/Exponentiation#Identities_and_properties) by $\mathrm{e}^{\mathbf{A}t}$:

$$e^{\mathbf{A}t}e^{-\mathbf{A}t}\mathbf{x}(t) = e^{\mathbf{A}t}\int_0^t e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau + e^{\mathbf{A}t}\mathbf{x}(0),$$

$$e^{\mathbf{A}t-\mathbf{A}t}\mathbf{x}(t) = e^{\mathbf{A}t}\int_0^t e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau + e^{\mathbf{A}t}\mathbf{x}(0),$$

$$\mathbf{x}(t) = e^{\mathbf{A}t}\int_0^t e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau + e^{\mathbf{A}t}\mathbf{x}(0).$$

OK! We solved for $\mathbf{x}(t)$.

To discretize our solution we're going to assume that we're sampling the system at even intervals, i.e. each sample is at kT for some time step T, and that the input $\mathbf{u}(t)$ is constant between samples (this is where the ZOH comes in). To simplify our notation as we go, we also define

$$\mathbf{x}[k] = \mathbf{x}(kT).$$

So using our new notation, we have

$$\mathbf{x}[k] = e^{\mathbf{A}kT}\mathbf{x}(0) + e^{\mathbf{A}kT}\int_0^{kT} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau.$$

Now we want to get things back into the form:

$$\mathbf{x}[k+1] = \mathbf{A}_d\mathbf{x}[k] + \mathbf{B}_d\mathbf{u}[k].$$

To start, let's write out the equation for $\mathbf{x}[k+1]$

$$\mathbf{x}[k+1] = e^{\mathbf{A}(k+1)T}\mathbf{x}(0) + e^{\mathbf{A}(k+1)T}\int_0^{(k+1)T} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau.$$

We want to relate $\mathbf{x}[k+1]$ to $\mathbf{x}[k]$. Being incredibly clever, we see that we can left multiply $\mathbf{x}[k]$ by $e^{\mathbf{A}T}$, to get

$$e^{\mathbf{A}T}\mathbf{x}[k] = e^{\mathbf{A}(k+1)T}\mathbf{x}(0) + e^{\mathbf{A}(k+1)T}\int_0^{kT} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau,$$

and can rearrange to solve for a term we saw in $\mathbf{x}[k+1]$:

$$e^{\mathbf{A}(k+1)T}\mathbf{x}(0) = e^{\mathbf{A}T}\mathbf{x}[k] - e^{\mathbf{A}(k+1)T}\int_0^{kT} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau.$$

Plugging this in, we get

$$\mathbf{x}[k+1] = e^{\mathbf{A}T}\mathbf{x}[k] - e^{\mathbf{A}(k+1)T}\left(\int_0^{kT} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau + \int_0^{(k+1)T} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau\right),$$

$$\mathbf{x}[k+1] = e^{\mathbf{A}T}\mathbf{x}[k] - e^{\mathbf{A}(k+1)T}\int_{kT}^{(k+1)T} e^{-\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau.$$

OK, we're getting close.

At this point we've got things in the right form, but we can still clean up that second term on the right-hand side quite a bit. First, note that using our starting assumption (that $\mathbf{u}(t) \in [k, kT)$ is constant), we can take $\mathbf{B}\mathbf{u}(t)$ outside the integral.

$$\mathbf{x}[k+1] = e^{\mathbf{A}T}\mathbf{x}[k] - e^{\mathbf{A}(k+1)T}\int_{kT}^{(k+1)T} e^{-\mathbf{A}\tau}d\tau\mathbf{B}\mathbf{u}[k].$$

Next, bring that $\mathrm{e}^{\mathbf{A}(k+1)T}$ term inside the integral:

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] - \int_{kT}^{(k+1)T} \mathrm{e}^{\mathbf{A}((k+1)T-\tau)} d\tau \mathbf{B}\mathbf{u}[k].$$

And now we're going to simplify the integral using variable substitution. Let $v = (k+1)T - \tau$, which means also that $\frac{dv}{d\tau} = -1 \rightarrow d\tau = -dv$. Evaluating the upper and lower bounds of the integral, when $\tau = (k+1)T$ then $v = 0$ and when $\tau = kT$ then $v = T$. With this, we can rewrite our equation:

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] - \int_{T}^{0} \mathrm{e}^{\mathbf{A}v} dv \mathbf{B}\mathbf{u}[k].$$

The astute will notice our integral integrates from T to 0 instead of 0 to T. Fortunately for us, we know $\int_a^b x = -\int_b^a$. We can just swap the bounds and multiply by -1, giving:

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] + \int_{0}^{T} \mathrm{e}^{\mathbf{A}v} dv \mathbf{B}\mathbf{u}[k].$$

And finally, we can evaluate our integral by recalling that $\frac{d}{dt}\mathrm{e}^{\mathbf{A}t} = \mathbf{A}\mathrm{e}^{\mathbf{A}t}$ and assuming that $\mathbf{A}$ is invertible:

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] + \mathbf{A}^{-1}\mathrm{e}^{\mathbf{A}v}|_{v=0}^{T}\mathbf{B}\mathbf{u}[k].$$

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] + \mathbf{A}^{-1}(\mathrm{e}^{\mathbf{A}T} - \mathrm{e}^{0})\mathbf{B}\mathbf{u}[k].$$

$$\mathbf{x}[k+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[k] + \mathbf{A}^{-1}(\mathrm{e}^{\mathbf{A}T} - \mathbf{I})\mathbf{B}\mathbf{u}[k].$$

We did it! The state and input matrices for our digital system are:

$$\mathbf{A}_d = \mathrm{e}^{\mathbf{A}T}$$

$$\mathbf{B}_d = \mathbf{A}^{-1}(\mathrm{e}^{\mathbf{A}T} - \mathbf{I})\mathbf{B}$$

And that's the hard part of discretization, the rest of the system is easy:
where, fortunately for us

$$\mathbf{C}_d = \mathbf{C},$$

$$\mathbf{D}_d = \mathbf{D}.$$

This then gives us our discrete system transfer function:

$$\mathbf{F}(z) = \frac{\mathbf{Y}_d(z)}{\mathbf{U}_d(z)} = \mathbf{C}_d(z\mathbf{I} - \mathbf{A}_d)^{-1}\mathbf{B}_d + \mathbf{D}_d.$$

NEF modeling with continuous lowpass filter dynamics

Now that we know how to discretize our system, we can look at compensating for the lowpass filter dynamics in discrete time. The equation for the discrete time lowpass filter is

$$H(z) = \frac{1-a}{z-a},$$

where $a = \mathrm{e}^{-\frac{dt}{\tau}}$.

Plugging that in to the discrete transfer fuction gets us

$$\mathbf{F}(H(z)) = \mathbf{C}_d(H(z)^{-1}\mathbf{I} - \mathbf{A}_d)^{-1}\mathbf{B}_d + \mathbf{D}_d.$$

$$\mathbf{F}(H(z)) = \mathbf{C}_d(\tfrac{z-a}{1-a}\mathbf{I} - \mathbf{A}_d)^{-1}\mathbf{B}_d + \mathbf{D}_d,$$

$$\mathbf{F}(H(z)) = \mathbf{C}_d(\tfrac{z\mathbf{I}}{1-a} - \tfrac{a\mathbf{I}-(1-a)\mathbf{A}_d}{1-a})^{-1}\mathbf{B}_d + \mathbf{D}_d,$$

$$\mathbf{F}(H(z)) = \mathbf{C}_d(z\mathbf{I} - a\mathbf{I} - (1-a)\mathbf{A}_d)^{-1}(1-a)\mathbf{B}_d + \mathbf{D}_d.$$

and we see that if we choose

$$\mathbf{A}'_d = \tfrac{1}{1-a}(\mathbf{A} + a\mathbf{I}),$$

$$\mathbf{B}'_d = \tfrac{1}{1-a}\mathbf{B},$$

then we get

$$\mathbf{F}'(H(z)) = \mathbf{C}_d(z\mathbf{I} - a\mathbf{I} - (1-a)\mathbf{A}'_d)^{-1}(1-a)\mathbf{B}'_d + \mathbf{D}_d,$$

$$= \mathbf{C}_d(z\mathbf{I} - a\mathbf{I} - (1-a)(\tfrac{1}{1-a}(\mathbf{A} + a\mathbf{I})))^{-1}(1-a)(\tfrac{1}{1-a}\mathbf{B}) + \mathbf{D}_d,$$

$$= \mathbf{C}_d(z\mathbf{I} - a\mathbf{I} - \mathbf{A} + a\mathbf{I})^{-1}\mathbf{B}) + \mathbf{D}_d,$$

$$= \mathbf{C}_d(z\mathbf{I} - \mathbf{A})^{-1}\mathbf{B}) + \mathbf{D}_d,$$

And now congratulations are in order. Proper compensation for the discrete lowpass filter dynamics has finally been achieved!

Point attractor example

What difference does this actually make in modelling? Well, everyone likes examples, so let's have one.

Here are the dynamics for a second-order point attractor system:

$$\ddot{x} = \alpha(\beta(x^* - x) - \dot{x})$$

with $x, \dot{x},$ and $\ddot{x}$ being the system position, velocity, and acceleration, respectively, x^* is the target position, and $\alpha,$ and β are gain values. So the acceleration is just going to be set such that it drives the system towards the target position, and compensates for velocity.

Converting this from a second order system to a first order system (https://www.youtube.com/watch?v=cq3bPBePE8E) we have

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\alpha\beta & -\alpha \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ \alpha\beta \end{bmatrix} \begin{bmatrix} 0 \\ x^* \end{bmatrix}$$

which we'll rewrite compactly as

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

OK, we've got our state space equation of the dynamical system we want to implement.

Given a simulation time step dt, we'll first calculate the discrete state matrices:

$$\mathbf{A}_d = \mathrm{e}^{\mathbf{A}dt},$$

$$\mathbf{B}_d = \mathbf{A}^{-1}(\mathrm{e}^{\mathbf{A}dt} - \mathbf{I})\mathbf{B}.$$

Great! Easy. Now we can calculate the state matrices that will compensate for the discrete lowpass filter:

$$\mathbf{A}'_d = \tfrac{1}{1-a}(\mathbf{A}_d + a\mathbf{I}),$$

$$\mathbf{B}'_d = \tfrac{1}{1-a}\mathbf{B}_d,$$

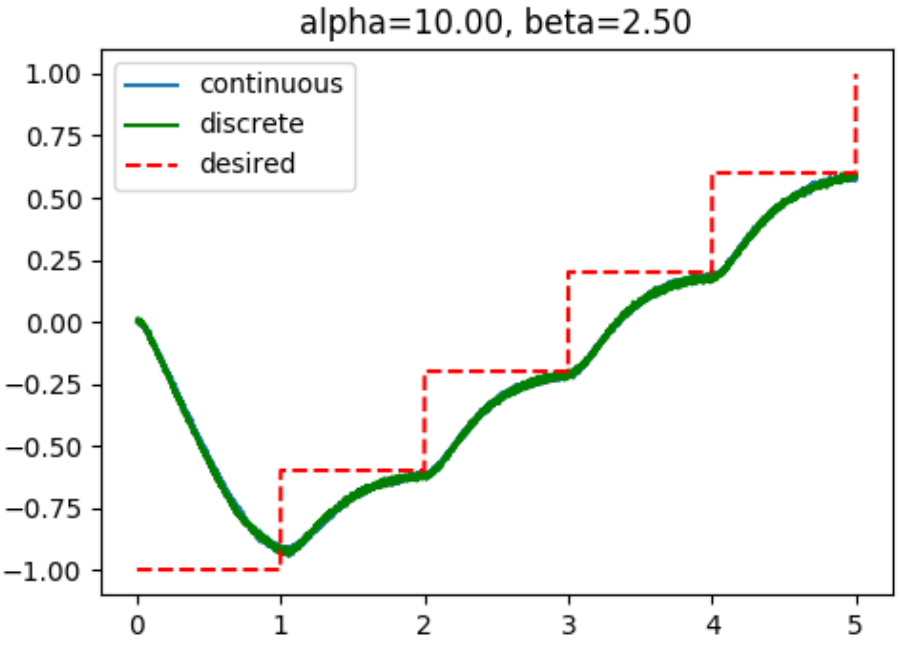
where $a = \mathrm{e}^{-\frac{dt}{\tau}}$.

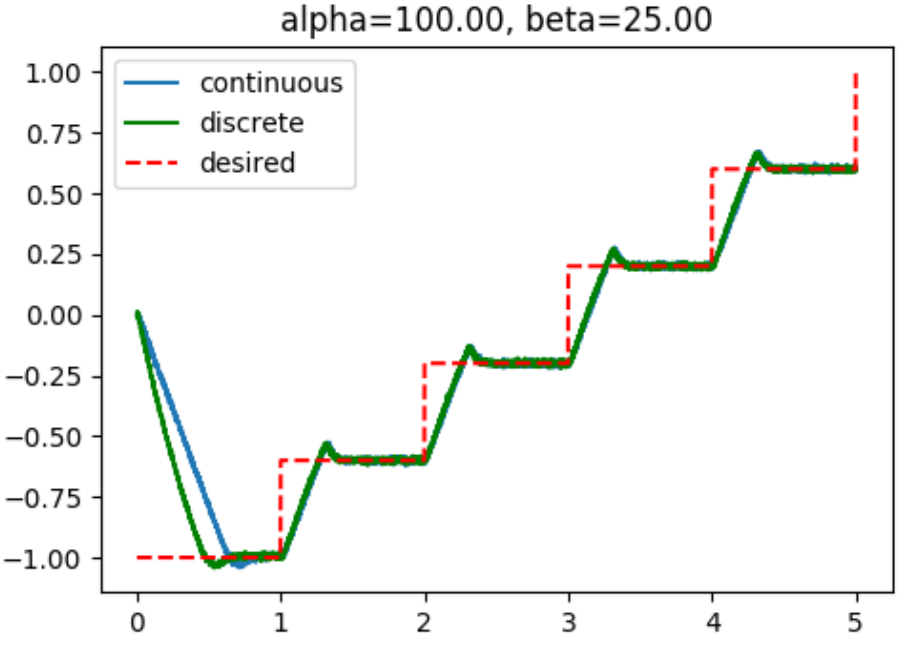
Alright! So that's our system now, a basic point attractor implementation in Nengo 2.3 looks like this:

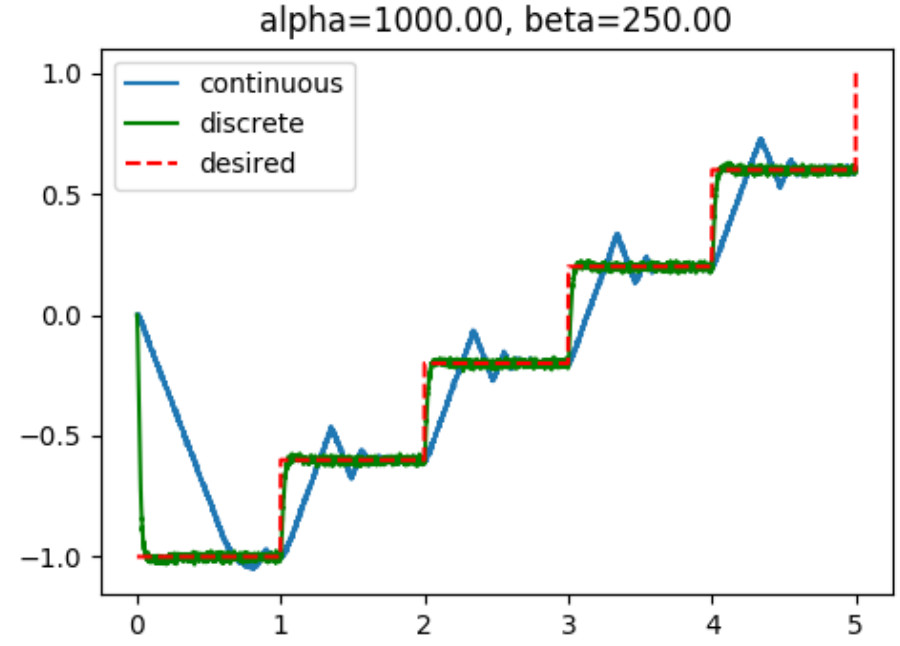
```
tau = 0.1  # synaptic time constant

# the A matrix for our point attractor
A = np.array([[0.0, 1.0],
              [-alpha*beta, -alpha]])

# the B matrix for our point attractor
B = np.array([[0.0], [alpha*beta]])

# account for discrete lowpass filter
a = np.exp(-dt/tau)
if analog:
    A = tau * A + np.eye(2)
    B = tau * B
else:
    # discretize
    Ad = expm(A*dt)
    Bd = np.dot(np.linalg.inv(A), np.dot((Ad - np.eye(2)), B))
    A = 1.0 / (1.0 - a) * (Ad - a * np.eye(2))
    B = 1.0 / (1.0 - a) * Bd

net = nengo.Network(label='Point Attractor')
net.config[nengo.Connection].synapse = nengo.Lowpass(tau)

with config, net:
    net.ydy = nengo.Ensemble(n_neurons=n_neurons, dimensions=2,
        # set it up so neurons are tuned to one dimensions only
        encoders=nengo.dists.Choice([[1, 0], [-1, 0], [0, 1], [0, -1]]))
    # set up Ax part of point attractor
    nengo.Connection(net.ydy, net.ydy, transform=A)

    # hook up input
    net.input = nengo.Node(size_in=1, size_out=1)
    # set up Bu part of point attractor
    nengo.Connection(net.input, net.ydy, transform=B)

    # hook up output
    net.output = nengo.Node(size_in=1, size_out=1)
    # add in forcing function
    nengo.Connection(net.ydy[0], net.output, synapse=None)
```

Note that for calculating `Ad` we're using `expm` which is the matrix exp function from `scipy.linalg` package. The `numpy.exp` does an elementwise `exp`, which is definitely not what we want here, and you will get some confusing bugs if you're not careful.

Code for implementing and also testing under some different gains is up on my GitHub (https://github.com/studywolf/blog/blob/master/Nengo%20scripting/Nengo%202/discrete_filter/point_attractor.py), and generates the following plots for dt=0.001:







In the above results you can see that the when the gains are low, and thus the system dynamics are slower, that you can't really tell a difference between the continuous and discrete filter compensation. But! As you

get larger gains and faster dynamics, the resulting effects become much more visible.

If you're building your own system, then I also recommend using the `ss2sim` function from Aaron's `nengolib` (https://github.com/arvoelke/nengolib) library. It automatically handles compensation for any synapses and generates the matrices that account for discrete or continuous implementations automatically. Using the library looks like:

```
tau = 0.1  # synaptic time constant
synapse = nengo.Lowpass(tau)

# the A matrix for our point attractor
A = np.array([[0.0, 1.0],
              [-alpha*beta, -alpha]])

# the B matrix for our point attractor
B = np.array([[0.0], [alpha*beta]])

from nengolib.synapses import ss2sim
C = np.eye(2)
D = np.zeros((2, 2))
linsys = ss2sim((A, B, C, D),
                synapse=synapse,
                dt=None if analog else dt)
A = linsys.A
B = linsys.B
```

Conclusions

So there you are! Go forward and model free of error introduced by improperly accounting for discrete simulation! If, like me, you're doing anything with neural modelling and motor control (i.e. systems with very quickly evolving dynamics), then hopefully you've found all this work particularly interesting, as I did.

There's a ton of extensions and different directions that this work can be and has already been taken, with a bunch of really neat systems developed using this more accurate accounting for synaptic filtering as a base. You can read up on this and applications to modelling time delays and time cells and lots lots more up on Aaron's GitHub (https://github.com/arvoelke/delay2017), and hisrecent papers, which are listed on his lab webpage.

Tagged discrete time, filtering, Nengo, neural modelling, simulation, zero-order hold