nctpy

Release 1.0.0

Linden Parkes, Jason Z. Kim, Jennifer Stiso

Jul 25, 2023

CONTENTS

CHAPTER

ONE

OVERVIEW

Network Control Theory (NCT) is a branch of physical and engineering sciences that treats a network as a dynamical system. Generally, the system is controlled through *control signals* that originate at a control node (or control nodes) and move through the network. In the brain, NCT models each region's activity as a time-dependent internal state that is predicted from a combination of three factors: (i) its previous state, (ii) whole-brain structural connectivity, and (iii) external inputs. NCT enables asking a broad range of questions of a networked system that are highly relevant to network neuroscientists, such as: which regions are positioned such that they can efficiently distribute activity throughout the brain to drive changes in brain states? Do different brain regions control system dynamics in different ways? Given a set of control nodes, how can the system be driven to a specific target state, or switch between a pair of states, by means of internal or external control input?

`nctpy` is a Python toolbox that provides researchers with a set of tools to conduct some of the common NCT analyses reported in the literature. Below, we list select publications that serve as a primer for these tools and their use cases:

1. Gu, S., Pasqualetti, F., Cieslak, M. et al. Controllability of structural brain networks. Nature Communications (2015). https://doi.org/10.1038/ncomms9414

2. Gu, S., Betzel R. F., Mattar, M. G. et al. Optimal trajectories of brain state transitions. NeuroImage (2017). https://doi.org/10.1016/j.neuroimage.2017.01.003

3. Karrer, T. M., Kim, J. Z., Stiso, J. et al. A practical guide to methodological considerations in the controllability of structural brain networks. Journal of Neural Engineering (2020). https://doi.org/10.1088/1741-2552/ab6e8b

4. Kim, J. Z., & Bassett, D. S. Linear dynamics & control of brain networks. arXiv (2019). https://arxiv.org/abs/1902.03309

CHAPTER

TWO

REQUIREMENTS

Currently, `nctpy` works with Python 3.9 and requires the following core dependencies:

- numpy (tested on 1.23.4)
- scipy (tested on 1.9.3)
- tqdm (tested on 4.64.1)

The `utils` module also requires:

- statsmodels (tested on 0.13.2)

The `plotting` module also requires:

- seaborn (tested on 0.12.0)
- nibabel (tested on 4.0.2)
- nilearn (tested on 0.9.2)

There are some additional (optional) dependencies you can install (note, these are only used for i/o and plotting in the Python notebooks located in the *scripts* directory):

- pandas (tested on 1.5.1)
- matplotlib (tested on 3.5.3)
- jupyterlab (tested on 3.4.4)
- sklearn (tested on 0.0.post1)

If you want to install the environment that was used to run the analyses presented in the manuscript, use the environment.yml file.

CHAPTER

THREE

BASIC INSTALLATION

Assuming you have Python 3.9 installed, you can install `nctpy` by opening a terminal and running the following:

```
pip install nctpy
```

CHAPTER

FOUR

QUESTIONS

If you have any questions, please contact Linden Parkes (https://twitter.com/LindenParkes), Jason Kim (https://twitter.com/jason_z_kim), or Jennifer Stiso (https://twitter.com/JenniferStiso).

CHAPTER

FIVE

CONTENTS

5.1 Installation and setup

5.1.1 Basic installation

This package requires Python 3.9. Assuming you have the correct version of Python installed, you can install `nctpy` by opening a terminal and running the following:

```
pip install nctpy
```

5.1.2 GitHub installation

Alternatively, you can install the most up-to-date version of from GitHub:

```
git clone https://github.com/BassettLab/nctpy.git
cd nctpy
pip install .
```

5.2 Getting Started

5.2.1 Introduction to Network Control

Classically, many neuroimaging studies have been interested in how the brain can be guided toward specific, diverse patterns of neural activity. Network control theory (NCT) is a powerful tool from physical and engineering sciences that can provide insight into these questions. NCT provides a specific, dynamical equation that defines how the activity of the brain will spread along white matter connections in response to some input. Important to these tools is the definition of the system (here, the brain) as a network, with nodes (brain regions) connected by edges. This network is stored in the adjacency matrix, $\mathbf{A}$. In neuroimaging, much of the application of NCT is divided into two categories: *controllability statistics* and *control energies*. Controllability statistics are properties of a structural brain network, or node in the network, that summarise information about control to many, nonspecific activity states. Control energies, on the other hand, provide quantification of how easily the network can transition between two specific activity states. For a more detailed introduction, see *Theory*.

`nctpy` is a Python toolbox that implements commonly used analyses from NCT. In this overview, we will walk through some of the basic functions, and important considerations for using NCT. The steps below will help you to get started. If you haven't done so yet, we recommend that you install the package (*Installation and setup*) so you can follow along with the examples.

5.2.2 Defining a dynamical system

We start by initializing a random adjacency matrix ($\mathbf{A}$) that will serve as a proxy for our structural connectivity. The matrix $\mathbf{A}$ defines the possible paths of activity spread for our dynamical system.

```
import numpy as np
# initialize A matrix
np.random.seed(42)  # for reproducibility
n_nodes = 5
A = np.random.rand(n_nodes, n_nodes)
print(A)
```

```
Out:
[[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
 [0.15599452 0.05808361 0.86617615 0.60111501 0.70807258]
 [0.02058449 0.96990985 0.83244264 0.21233911 0.18182497]
 [0.18340451 0.30424224 0.52475643 0.43194502 0.29122914]
 [0.61185289 0.13949386 0.29214465 0.36636184 0.45606998]]
```

In `nctpy`, the dynamics of the system will evolve according to one of two definitions. The first is a discrete-time system that follows this equation:

$$\mathbf{x}(t+1) = A\mathbf{x}(t) + B\mathbf{u}(t).$$

The second is a continuous-time system that follows this equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = A\mathbf{x}(t) + B\mathbf{u}(t).$$

In both equations, $\mathbf{x}$ is the state (or regional neural activity) of the system over time, $\mathbf{u}$ is the input to the system, and B defines which nodes receive input.

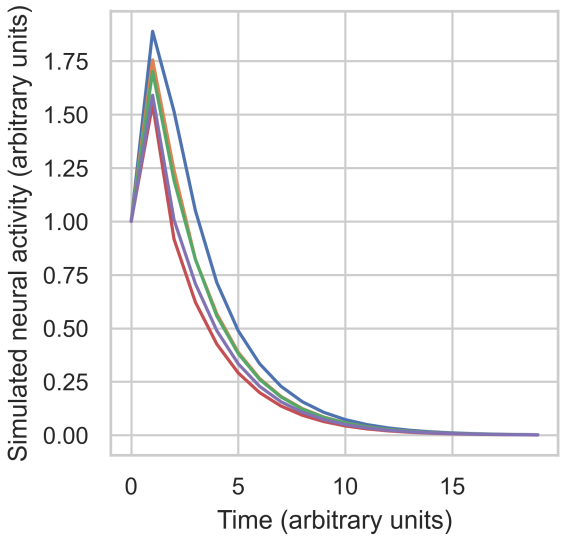
Before applying one of the above models, most work will first normalize $\mathbf{A}$ to be stable. In response to adding a single unit of input, the activity in a stable matrix will spread but eventually die down, and the system will reach a stable state. By contract, an unstable system will exhibit states that continue growing forever. Practically, this normalization is done differently depending on whether we analyze a discrete-time or a continuous-time system. For discrete-time systems, all eigenvalues must be less than 1 for the system to be stable, and for continuous-time systems all eigenvalues must be less than 0. Unless otherwise noted, the functions of `nctpy` will work across both time systems. Here, we will define our system to be in discrete time, which can be achieved using the `nctpy.utils.matrix_normalization` function:

```
from nctpy.utils import matrix_normalization
system = 'discrete'
A_norm = matrix_normalization(A=A, c=1, system=system)
print(A_norm)
```

```
Out:
[[0.11828952 0.30026034 0.23118275 0.18907194 0.04927475]
 [0.04926713 0.01834432 0.27356099 0.18984778 0.22362776]
 [0.00650112 0.30632279 0.26290707 0.06706222 0.05742506]
 [0.05792392 0.09608762 0.16573175 0.13641949 0.09197775]
 [0.19323908 0.04405579 0.09226689 0.11570661 0.14403878]]
```

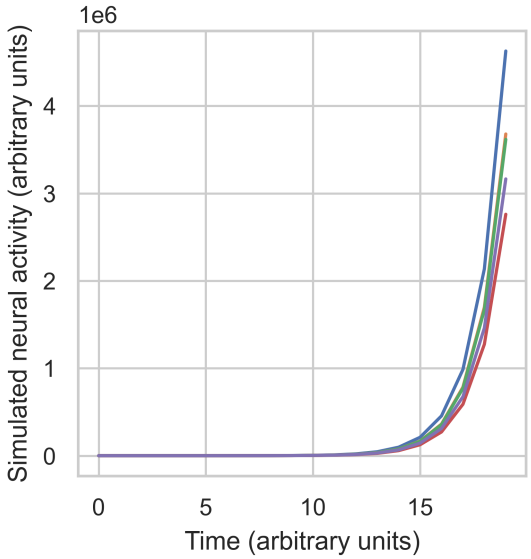
We can visualize the behavior of a stable and unstable matrix by simulating the systems response to small amount of input. These simulations can be run using the `nctpy.energies.sim_state_eq` function. First, let's visualize the stable matrix we create above, `A_norm`:

```
from nctpy.energies import sim_state_eq
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid', context='paper', font_scale=1)

T = 20  # time horizon
U = np.zeros((n_nodes, T))  # the input to the system
U[:, 0] = 1  # impulse, 1 input at the first time point delivered to all nodes
B = np.eye(n_nodes)  # uniform full control set
x0 = np.ones((n_nodes, 1))  # initial state, all nodes set to 1 unit of neural activity
x = sim_state_eq(A_norm=A_norm, B=B, x0=x0, U=U, system=system)

# plot
f, ax = plt.subplots(1, 1, figsize=(3, 3))
ax.plot(x.T)
ax.set_ylabel('Simulated neural activity (arbitrary units)')
ax.set_xlabel('Time (arbitrary units)')
f.savefig('A_stable.png', dpi=600, bbox_inches='tight', pad_inches=0.01)
plt.show()
```



The above figure shows that our system nodes all started with 1 unit of activity. Then, an impulse was deliver which increased their activity by different amounts. This happened because of the connectivity between nodes. Finally, following the impulse, all activity decayed to 0 over time. Next, let's see what happens when we use the unstable matrix, `A`:

```
# unstable A
x = sim_state_eq(A_norm=A, B=B, x0=x0, U=U, system=system)

# plot
f, ax = plt.subplots(1, 1, figsize=(3, 3))
ax.plot(x.T)
ax.set_ylabel('Simulated neural activity (arbitrary units)')
ax.set_xlabel('Time (arbitrary units)')
f.savefig('A_unstable.png', dpi=600, bbox_inches='tight', pad_inches=0.01)
plt.show()
```



In the unstable case, we can see that activity has exploded over time, exceeding 3×10^6 for most nodes.

5.2.3 Calculating controllability statistics

Now that we have a stable matrix, we're ready to calculate some NCT metrics. The first metric included in `nctpy` is *average controllability*. Average controllability measures the impulse response of the system, and can be thought of as an indicator of a node's general capacity to control dynamics. Additionally, this metric represents an upper bound on the energy required to transition between any two states. Average controllability can be calculated using `nctpy.metrics.ave_control`:

```
from nctpy.metrics import ave_control
ac = ave_control(A_norm=A_norm, system=system)
print(ac)
```

```
Out:
[1.28964075 1.18649349 1.18014308 1.10255958 1.13759366]
```

We can see that node 1 has the highest average controllability.

5.2.4 Calculating control energies

Now, let's say instead that we want to know how well our system can transition between two specific neural states. To achieve this, we can calculate the amount of *control energy* that would need to be input into our system to transition it between an initial state ($x0$) and a target state (xf). This is done using `nctpy.energies.get_control_inputs` and `nctpy.energies.integrate_u`. For the sake of demonstration, let's switch to a continuous time system here. Note, `ave_control`, `get_control_inputs`, and `integrate_u` can all be used for both discrete-time and continuous-time systems:

```
system = 'continuous'
A_norm = matrix_normalization(A=A, c=1, system=system)
print(A_norm)
```

```
Out:
[[-0.88171048  0.30026034  0.23118275  0.18907194  0.04927475]
 [ 0.04926713 -0.98165568  0.27356099  0.18984778  0.22362776]
 [ 0.00650112  0.30632279 -0.73709293  0.06706222  0.05742506]
 [ 0.05792392  0.09608762  0.16573175 -0.86358051  0.09197775]
 [ 0.19323908  0.04405579  0.09226689  0.11570661 -0.85596122]]
```

```
from nctpy.energies import get_control_inputs, integrate_u

# define initial and target states as random patterns of activity
np.random.seed(42)  # for reproducibility
x0 = np.random.rand(n_nodes, 1)  # initial state
xf = np.random.rand(n_nodes, 1)  # target state

# set parameters
T = 1  # time horizon
rho = 1  # mixing parameter for state trajectory constraint
S = np.eye(n_nodes)  # nodes in state trajectory to be constrained

# get the state trajectory (x) and the control inputs (u)
x, u, n_err = get_control_inputs(A_norm=A_norm, T=T, B=B, x0=x0, xf=xf, system=system,␣
↪rho=rho, S=S)
```

`get_control_inputs` outputs the neural activity of the system (`x`, state trajectory), the control signals that drove the system to transition between the initial and target states (`u`), and a pair of numerical errors (`n_err`). Let's unpack all this!

To complete the state transition, `get_control_inputs` utilizes a cost function that constrains both the magnitude of the control signals (`u`) as well as the magnitude of the state trajectory (`x`). That is, we are primarily trying to find the control signals that achieve a desired state transition with the lowest amount of input, while also constraining the level of neural activity in the state trajectory; we don't want wild fluctuations in neural activity! The input parameter `rho` allows researchers to tune the mixture of these two costs while finding the control signals. Specifically, `rho=1` places equal cost over the magnitude of the control signals and the state trajectory. Reducing `rho` below 1 increases the extent to which the state trajectory adds to the cost function alongside the control signals. Conversely, increasing `rho` beyond 1 reduces the state trajectory contribution, thus increasing the relative prioritization of the control signals. Lastly, `S` takes in an $N \times N$ matrix whose diagonal elements define which nodes' activity will be constrained in the state trajectory. In summary, `S` designates which nodes' neural activity will be constrained while `rho` determines by how much (relative to the control signals). Here, by setting `rho=1` and `S=np.eye(n_nodes)`, we are implementing what we refer to as *optimal control*. If `S` is set to an $N \times N$ matrix of zeros, then the state trajectory is completely unconstrained; we refer to this setup as *minimum control*. In this case, `rho` is ignored.

Next, let's consider those numerical errors. The first error is the *inversion error*, which measures the conditioning of the optimization problem. If this error is small, then solving for the control signals was well-conditioned. The second error is the *reconstruction error*, which is a measure of the distance between the target state (`xf`) and the state trajectory at time *T*. If this error is small, then the state transition completed successfully; that is, the neural activity at the end of the simulation was equivalent to the neural activity encoded by `xf`. We consider errors $< 1 \times 10^{-8}$ as adequately small:

```
# print errors
thr = 1e-8

# the first numerical error corresponds to the inversion error
print('inversion error = {:.2E} (<{:.2E}={:})'
      .format(n_err[0], thr, n_err[0] < thr))
```

(continues on next page)

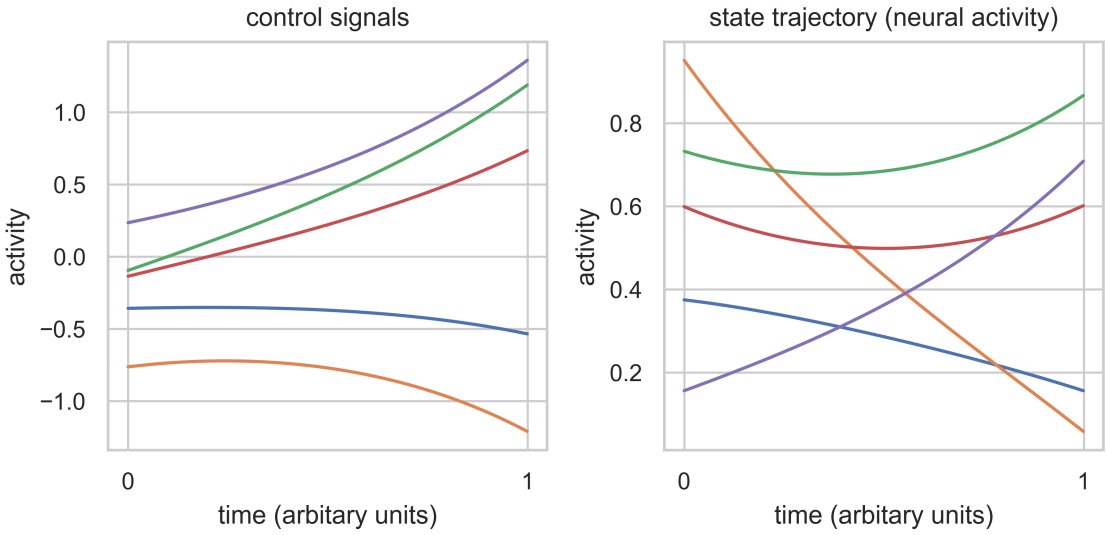
(continued from previous page)

```
# the second numerical error corresponds to the reconstruction error
print('reconstruction error = {:.2E} (<{:.2E}={:})'
      .format(n_err[1], thr, n_err[1] < thr))
```

```
Out:
inversion error = 1.31E-16 (<1.00E-08=True)
reconstruction error = 5.45E-14 (<1.00E-08=True)
```

Now that we've unpacked all that, let's plot the state trajectory (`x`) and the control signals (`u`) to see what they're doing:

```
# plot x and u
f, ax = plt.subplots(1, 2, figsize=(6, 3))
# plot control signals for initial state
ax[0].plot(u)
ax[0].set_title('control signals')

# plot state trajectory for initial state
ax[1].plot(x)
ax[1].set_title('state trajectory (neural activity)')

for cax in ax.reshape(-1):
    cax.set_ylabel("activity")
    cax.set_xlabel("time (arbitary units)")
    cax.set_xticks([0, x.shape[0]])
    cax.set_xticklabels([0, T])

f.tight_layout()
f.savefig('plot_xu.png', dpi=600, bbox_inches='tight', pad_inches=0.01)
plt.show()
```



Finally, we'll integrate `u` to compute *control energy*:

```
# integrate control inputs to get control energy
node_energy = integrate_u(u)
```

(continues on next page)

(continued from previous page)

```
print('node energy =', node_energy)

# summarize nodal energy to get control energy
energy = np.sum(node_energy)
print('energy = {:.2F}'.format(np.round(energy, 2)))
```

```
Out:
node energy = [159.35334645 728.32771143 349.67802113 120.56428349 563.2983561 ]
energy = 1921.22
```

The *control energy* associated with our state transition was 1921.22.

That concludes the *getting started* section.

5.3 Theory

Note: Relevant publication: Kim et al. 2020 Neural Engineering. Much of the inspiration for the derivations came from Dr. George Pappas' course ESE 500 on Linear Systems Theory at UPenn.

When we talk about the "control" of a system, we broadly refer to some input or change to the system that alters its behavior in a desired way. To more precisely discuss control, we first have to discuss the object that is being controlled: the system. In the network control framework, what is being controlled is a **dynamical system**.

5.3.1 What is a Dynamical System?

As per the Wikipedia article on dynamical systems: "a dynamical system is a system in which a function describes the time dependence of a point in geometrical space." We're going to reword this sentence a bit to say: "a **dynamical** system is a system whose **states** evolve **forward in time** in **geometric space** according to a **function**." Let's unpack this sentence a bit.

"A dynamical system is a system"

The word "dynamic" (change) is used in contrast to "static" (no change). If a system is dynamic, it changes over time. That's all well and good, but what exactly is changing about the system? The answer is: the **states**.

"whose states"

A state is just a complete description of a system. Take for example a train. If I know the position x of the train from some station, then I know exactly where the train is. It can only move forwards or backwards along the track, and every position along that track has an associated position x, or *state*. As another example, in a computer, if I know the voltage of every single transistor, then I have a complete description of the computer, and I can unambiguously describe all possible states of the computer using these transistor voltages (states).

"evolve forward in time"

When we say dynamic (states are changing), we mean to say that the states are changing forward in time. Hence, *time* is a very important concept in dynamical systems. The field of dynamical systems broadly concerns itself with the question "how do the system states evolve forward in time?"

"in geometric space"

When we say "geometric," we *usually* mean the colloquial usage of the word. That is, Euclidean space. We live in 3-dimensional Euclidean space, where every point is described by 3 coordinates: (x, y, z). In a dynamical system, there is no need to restrict ourselves to 3 dimensions. In the train example, we can represent the position of the train along the track, x, on a number line. Even if the track itself is not straight, we can "straighten out" the track to form a 1-dimensional line. As another exmple, the FitzHugh-Nagumo model is a 2-dimensional simplification of a Hodgkin-Huxley neuron, with two states, v and w. We can plot these states separately over time (left,center), or we can plot them together in a 2-dimensional geometric space, where each axis represents either v or w (right).

"according to a function."

Now, just because a system has states and evolves forward in time does not make it a dynamical system. For a system to be dynamical, it must evolve forward in time according to a function. This requirement is precisely where *differential equations* enters the fray. Specifically, the function that the dynamical system uses to evolve forward in time is a differential equation. For example, the FitzHugh-Nagumo model evolves according to the functions

$$\frac{\mathrm{d}v}{\mathrm{d}t} = v - \frac{v^3}{3} - w + 0.5$$
$$\frac{\mathrm{d}w}{\mathrm{d}t} = \frac{1}{12.5}(v + 0.8 - 0.7w)$$

5.3.2 What is a Differential Equation?

As per the Wikipedia definition: "a differential equation is an equation that relates one or more functions and their derivatives." Let's break down this sentence.

"A differential equation is an equation"

An equation is a relation that equates the items left of the equal sign to the items right of the equal sign. For example, for a right triangle with side lengths a, b and hypotenuse length c, the Pythagorean equation is:

$$c^2 = a^2 + b^2$$

This equation has thre evariables that are related by one equation. Hence, if I fix a and b, then I know what c has to be for the triangle to be a right triangle.

"that relates one or more functions and their derivatives."

A derivative is an operation that tells us how a variable changes. For example, if c is a variable measuring the side length of the triangle, then $\mathrm{d}c$ is a variable measuring the *change* in that side length. Typically, these change variables come as ratios to measure how quickly one variable changes with respect to another variable. For example, $\frac{\mathrm{d}c}{\mathrm{d}a}$ is a ratio between a change in c with respect to a change in a.

5.3.3 Dynamical Systems & Differential Equations

Recall the two statements that we have made thus far:

- Dynamical system: a system whose states evolve forward in time in geometric space according to a **function**
- Differential equation: an equation that relates one or more functions and their **derivatives**.

Hence the relationship between these two is that the **function** is a differential equation of **derivatives**. In particular, the derivative of the system states with respect to time.

In the differential equations of a dynamical system, the left-hand side contains a derivative of the state with respect to time, $\frac{\mathrm{d}x}{\mathrm{d}t}$. The right-hand side contains a function of the states, $f(x)$. Hence, generally speaking, a dynamical equation looks like

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x).$$

As a specific example, let's look at the dynamical equation

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x,$$

and let's see what happens at some specific states.

- If $x = 1$, then $\frac{\mathrm{d}x}{\mathrm{d}t} = -1$. In other words, if the system state is at 1, then the change in state with respect to time is negative, such that the state moves towards 0.
- If $x = -1$, then $\frac{\mathrm{d}x}{\mathrm{d}t} = 1$. In other words, if the system state is at -1, then the change in state with respect to time is positive, such that the state moves towards 0.
- If $x = 0$, then $\frac{\mathrm{d}x}{\mathrm{d}t} = 0$. The system does not change, and the state remains at 0.

If we plot the trajectories $x(t)$ over time (left), we see that, as predicted, the trajectories all move towards 0 (left), and that the change in the state, $\frac{\mathrm{d}x}{\mathrm{d}t}$, also points towards 0 (right).

Hence, the dynamical equation for a system describes the evolution of the state at every point in state space. To visualize this description in 2-dimensions, let us revisit the equations for the FitzHugh-Nagumo model,

$$\frac{\mathrm{d}v}{\mathrm{d}t} = v - \frac{v^3}{3} - w + 0.5$$
$$\frac{\mathrm{d}w}{\mathrm{d}t} = \frac{1}{12.5}(v + 0.8 - 0.7w),$$

and at every point (v, w), we will draw an arrow pointing towards $(\mathrm{d}v/\mathrm{d}t, \mathrm{d}w/\mathrm{d}t)$.

We observe that at every point in the state space, we can draw an arrow defined by the dynamical equations. Additionally, we observe that the evolution of the system states, $v(t)$ and $w(t)$, follow these arrows. Hence, the differential equations define the flow of the system states over time.

For convenience, we will name all of our state varibles $x_1, x_2, \cdots, x_N$, and collect them into an N-dimensional vector $\mathbf{x}$. For an additional convenience, instead of always writing the fraction $\frac{\mathrm{d}x}{\mathrm{d}t}$, we will use $\dot{x}$ to represent the time derivative of x.

5.3.4 Linear State-Space Systems

Now that we have a better idea of what a dynamical system is, we would like to move on to control. However, there is a fundamental limitation when attempting to control a system, which is that we do not know how the system will naturally evolve. At any given state, $\mathbf{x}(t)$, we can use the dynamical equations to know where the state will *immediately* go, $\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t}$. However, we generally cannot know where the state will end up after a *finite* amount of time, at $\mathbf{x}(t+T)$. This problem extends to any perturbation we perform on the system, where we cannot know how the perturbation will affect the state after a finite amount of time.

However, there is a class of dynamical systems where we can know both where the states will end up, and how a perturbation will change the states after a finite amount of time. These systems are called linear time-invariant systems, or LTI systems.

scalar LTI system

We have already looked at an example of an LTI system, namely,

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x.$$

We can make this system a bit more general, and look at

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ax,$$

where a is a constant real number. Using some basic calculus, we can actually solve for the trajectory $x(t)$. First, we divide both sides by x and multiply both sides by a to match terms,

$$\frac{1}{x}\mathrm{d}x = a\mathrm{d}t.$$

Then, we integrate both sides,

$$\int \frac{1}{x}\mathrm{d}x = \int a\mathrm{d}t + c$$
$$\ln|x| = at + c.$$

Finally, we exponentiate both sides to pull out $x(t)$:

$$x(t) = Ce^{at},$$

where the constant C is the initial condition, $C = x(0)$. This is because when we plug in $t = 0$, the exponential becomes $e^{a0} = 1$. Hence, we can write our final trajectory as

$$x(t) = x(0)e^{at},$$

which tells us exactly what the state of our system will be at every point in time. This knowledge of the state at every point in time is generally very difficult to obtain for nonlinear systems. To verify that this trajectory really is a solution to our dynamical equation, we can substitute it back into the differential equation, and check if the left-hand side equals the right-hand side. To evaluate the left-hand side, we must take the time derivative of e^{at}, which we can do by writing the exponential as a Taylor series, such that $e^{at} = \sum_{k=0}^{\infty} \frac{(at)^k}{k!}$. Then taking the derivative of each term with respect to time, we get

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}e^{at} &= \frac{\mathrm{d}}{\mathrm{d}t}\left(1 + \frac{at}{1!} + \frac{a^2t^2}{2!} + \frac{a^3t^3}{3!} + \cdots + \frac{a^kt^k}{k!} + \cdots\right) \\
&= 0 + \frac{a}{1!} + 2\frac{a^2t}{2!} + 3\frac{a^3t^2}{3!} + \cdots + k\frac{a^lt^{k-1}}{k!} + \cdots \\
&= a\left(1 + \frac{at}{1!} + \frac{a^2t^2}{2!} + \cdots + \frac{a^kt^k}{k!}\right) \\
&= ae^{at}.
\end{aligned}$$

Hence, the derivative of e^{at} is equal to ae^{at}, such that the left-hand side of the dynamical equation equals the right-hand side.

vector LTI system

Of course, systems like the brain typically have many states, and writing down the equations for all of those states would be quite tedious. Fortunately, we can obtain all of the results in scalar LTI systems for vector LTI systems using matrix notation. In matrix form, the state-space LTI dynamics are written as

$$\underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ \vdots\\ \dot{x}_N\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}a_{11} & a_{12} & \cdots & a_{1N}\\ a_{21} & a_{22} & \cdots & a_{2N}\\ \vdots & \vdots & \ddots & \vdots\\ a_{N1} & a_{N2} & \cdots & a_{NN}\end{bmatrix}}_{A} \underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ \vdots\\ \dot{x}_N\end{bmatrix}}_{\mathbf{x}},$$

or, more compactly, as

$$\dot{\mathbf{x}} = A\mathbf{x}.$$

Here, a_{ij} is the element in the i-th row and j-th column of matrix A, and represents the coupling from state j to state i.

Now, it might be too much to hope that the solution to the vector LTI system is simply a matrix version of the scalar form, perhaps something like $\mathbf{x}(t) = e^{At}\mathbf{x}(0)$. However, this form is precisely the solution to the vector dynamical

equation! Exactly as in the scalar version, we can write the *matrix exponential*, e^{At}, as a Taylor series such that $e^{At} = \sum_{k=0}^{\infty} \frac{(At)^k}{k!}$, and again take the time derivative of each term to get

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t} e^{At} &= \frac{\mathrm{d}}{\mathrm{d}t}\left(1 + \frac{At}{1!} + \frac{A^2t^2}{2!} + \frac{A^3t^3}{3!} + \cdots + \frac{A^kt^k}{k!} + \cdots\right) \\
&= 0 + \frac{A}{1!} + 2\frac{A^2t}{2!} + 3\frac{A^3t^2}{3!} + \cdots + k\frac{A^lt^{k-1}}{k!} + \cdots \\
&= A\left(1 + \frac{At}{1!} + \frac{A^2t^2}{2!} + \cdots + \frac{A^kt^k}{k!}\right) \\
&= Ae^{at}.
\end{aligned}$$

Hence, the trajectory of a vector LTI system is given simply by

$$\mathbf{x}(t) = e^{At}\mathbf{x}(0).$$

In general, e^{At} is called the *impulse response* of the system, because for any impulse $\mathbf{x}(0)$, the impulse response tells us precisely how the system will evolve.

5.3.5 The Potential of Linear Response

Until now, we have written down several examples of systems that we have called linear. Drawing on our prior coursework in linear algebra, we recall that the adjective *linear* is used to describe a particular property of some operator $f(\cdot)$ acting on some objects x_1, x_2. That is, if

$$y_1 = f(x_1) \text{ and } y_2 = f(x_2),$$

then $f(\cdot)$ is linear if

$$ay_1 + by_2 = f(ax_1 + bx_2).$$

Colloquially, if an operator is linear, then it *adds distributively*. Scaling the input by a constant scales the output by the same constant, and the sum of two inputs yields the sum of the outputs.

So then, what do we mean when we say that our dynamical system is linear? In this case, we mean that the *impulse response* is linear. That is, for two initial conditions, $\mathbf{x}_1(0), \mathbf{x}_2(0)$, if

$$\mathbf{x}_1(t) = e^{At}\mathbf{x}_1(0) \text{ and } \mathbf{x}_2(t) = e^{At}\mathbf{x}_2(0),$$

then

$$a\mathbf{x}_1(t) + b\mathbf{x}_2(t) = e^{At}(a\mathbf{x}_1(0) + b\mathbf{x}_2(0)),$$

which is true by the distributive property.

a 2-state example

While this property might not seem so impressive at first glance, the implications are actually quite powerful. Specifically, this linearity allows us to write all possible trajectories of our system as a simple weighted sum of initial conditions. Hence, rather than having to simulate all initial states to see if we reach a particular final state, we can reconstruct the initial state that yields a desired final state. To demonstrate, consider the following simple 2-dimensional system

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}-1 & -2\\ 1 & 0\end{bmatrix},$$

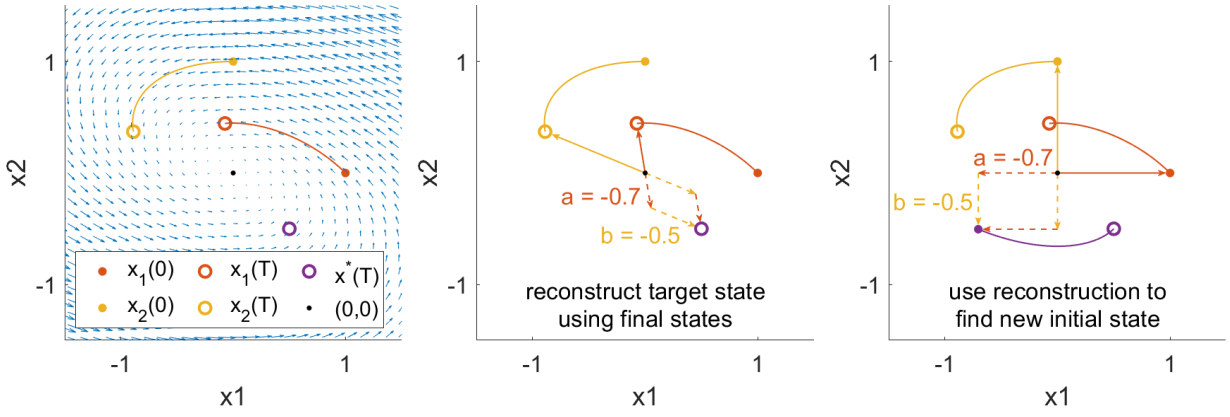
and two initial conditions

$$\mathbf{x}_1(0) = \begin{bmatrix}1\\ 0\end{bmatrix}, \qquad \mathbf{x}_2(0) = \begin{bmatrix}0\\ 1\end{bmatrix}.$$

Evolving these two states until $T = 1$ yields final states

$$\mathbf{x}_1(T) = \begin{bmatrix}-0.0734\\ 0.4445\end{bmatrix}, \qquad \mathbf{x}_2(T) = \begin{bmatrix}-0.8890\\ 0.3711\end{bmatrix},$$

and we can plot the trajectories towards those final states below (left).



Now, suppose we wanted the system to actually reach a different final state, say

$$\mathbf{x}^*(T) = \begin{bmatrix}0.5\\ -0.5\end{bmatrix}.$$

Because of the linearity of the system, we know that weighted sums of the initial states map to the same weighted sums of the trajectories. We can reverse this idea and write the desired final state as a weighted sum of trajectories,

$$\mathbf{x}^*(T) = a\mathbf{x}_1(T) + b\mathbf{x}_2(T) = \begin{bmatrix}\mathbf{x}_1(T) & \mathbf{x}_2(T)\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix},$$

and solve for the weights through simple matrix inversion

$$\begin{bmatrix}a\\ b\end{bmatrix} = \begin{bmatrix}\mathbf{x}_1(T) & \mathbf{x}_2(T)\end{bmatrix}^{-1}\mathbf{x}^*(T) = \begin{bmatrix}-0.7\\ -0.5\end{bmatrix}.$$

Then, if we use the same weighted sums of the initial states, then the new initial state is guaranteed to reach the desired target state,

$$\mathbf{x}^*(0) = a\mathbf{x}_1(0) + b\mathbf{x}_2(0),$$

due to the properties of linearity such that

$$\begin{aligned} e^{AT}\mathbf{x}^*(0) &= e^{AT}(a\mathbf{x}_1(0) + b\mathbf{x}_2(0)) \\ &= ae^{AT}\mathbf{x}_1(0) + be^{AT}\mathbf{x}_2(0) \\ &= a\mathbf{x}_1(T) + b\mathbf{x}_2(T) \\ &= \mathbf{x}^*(T). \end{aligned}$$

As we can see, we did not have to do any guesswork in solving for the initial state that yielded the desired final state. Instead, we reconstructed the final state from a basis of final states, and took advantage of the linear property of the impulse response to apply that reconstruction to the initial states. This reconstruction using basis vectors and linearity is the core principle behind network control theory.

an easier approach

While the previous reconstruction example was useful, the linearity of the impulse response actually allows us to solve the problem *much* faster, because at the end of the day, the impulse response is simply a linear system of equations,

$$\mathbf{x}(T) = e^{AT}\mathbf{x}(0).$$

So, we know A, and we know the desired target state, $\mathbf{x}(T)$, so we just multiply both sides of the equation by the inverse of e^{AT} to yield the correct initial state

$$\mathbf{x}(0) = e^{-AT}\mathbf{x}(T) = \begin{bmatrix} -0.7 \\ -0.5 \end{bmatrix}.$$

And… that's kind of it. And fundamentally, the control of these systems uses the exact same idea. That is, we find some *linear* operation that takes us from the control input to the final state, then solve for the input using some fancy versions of matrix inverses.

5.3.6 Controlled Dynamics

Until now, we have worked with LTI dynamics, which we write as

$$\dot{\mathbf{x}} = A\mathbf{x}.$$

When we say *control*, we intend to perturb the system using some external inputs, $u_1(t), u_2(t), \cdots, u_k(t)$, that we will collect into a vector $\mathbf{u}(t)$. These inputs might be electromagnetic stimulation from transcranial magnetic stimulation (TMS), some modulation of neurotransmitters through medication, or sensory inputs. And of course, these inputs don't randomly affect all brain states separately, but have a specific pattern of effect based on the site of stimulation, neurotransmitter distribution, or sensory neural pathways. We represent this mapping from stimuli to brain regions through vectors $\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_k$, which we collect into an $N \times k$ matrix B. Then our new controlled dynamics become

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

So now we have a bit of a problem. We would like to write $\mathbf{x}(t)$ as some nice linear function as before, but how do we do this? The derivation requires a bit of algebra, so feel free to skip it!

derivation of the controlled response

So the first thing we will try to do is, as before, move all of the same variables to one side. So first, we will subtract both sides by $A\mathbf{x}$

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

Then, as before, we want to integrate the time derivative. However, simply integrating both sides will yield a $\int A\mathbf{x}$ term, which we do not want. To combine the $\dot{\mathbf{x}}$ and $A\mathbf{x}$ terms, we will first mutiply the equation by :math:e^{-At},

$$e^{At}\dot{\mathbf{x}} - e^{At}A\mathbf{x} = e^{At}B\mathbf{u},$$

and notice that we can actually perform the reverse of the product rule on the left-hand side. Specifically, $\frac{\mathrm{d}}{\mathrm{d}t}e^{At}\mathbf{x} = e^{At}\dot{\mathbf{x}} - e^{At}A\mathbf{x}$ (small note, $e^{-At}A = Ae^{-At}$ because a matrix and functions of that matrix commute). Substituting this expression into the left-hand side, we get

$$\frac{\mathrm{d}}{\mathrm{d}t}e^{At}\mathbf{x} = e^{At}B\mathbf{u}$$

Now we are almost done, as we integrate both sides from $t = 0$ to $t = T$ to yield

$$e^{-AT}\mathbf{x}(T) - \mathbf{x}(0) = \int_0^T e^{-At}B\mathbf{u}(t)\mathrm{d}t$$

Finally, we isolate the term $\mathbf{x}(T)$ by adding both sides of the equation by $\mathbf{x}(0)$, and multiplying through by e^{AT} to yield

$$\underbrace{\mathbf{x}(T)}_{\text{target}} = \underbrace{e^{AT}\mathbf{x}(0)}_{\text{natural}} + \underbrace{\int_0^T e^{A(T-t)}B\mathbf{u}(t)\mathrm{d}t}_{\text{controlled}}$$

We notice that the first term, the "natural" term, is actually our original, uncontrolled impulse response. We also notice that the second term, the "controlled" term, is just a convolution of our input, $\mathbf{u}(t)$, with the impulse response. For conciseness, we will write the convolution using a fancy letter $\mathcal{L}(\mathbf{u}) = \int_0^T e^{A(T-t)}B\mathbf{u}(t)\mathrm{d}t$, and rewrite our controlled response as

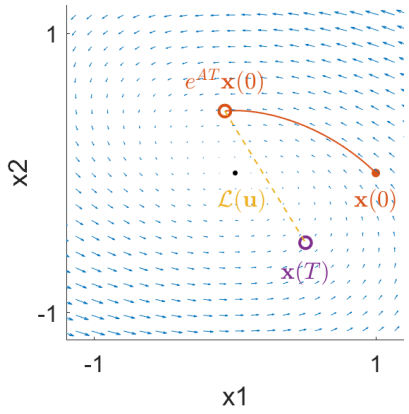
$$\underbrace{\mathbf{x}(T)}_{\text{target}} = \underbrace{e^{AT}\mathbf{x}(0)}_{\text{natural}} + \underbrace{\mathcal{L}(\mathbf{u}(t))}_{\text{controlled}}$$

some intuition for the controlled response

We can gain some simple intuition by rearranging the controlled response a little

$$\underbrace{\mathbf{x}(T)}_{\text{target}} - \underbrace{e^{AT}\mathbf{x}(0)}_{\text{natural}} = \underbrace{\mathcal{L}(\mathbf{u}(t))}_{\text{controlled}}$$

If we look closely, we notice that the controlled response simply makes up the *difference* between the natural evolution of the system from its initial state, and the desired target state. To visualize this equation in our previous 2-dimensional example, we mark the initial state and natural evolution of the initial state in orange, and the desired target state in purple. The controlled response is algebraically responsible for making up the gap between the initial and target state.



5.3.7 The Potential of Linear Controlled Response

So now we reach the final question: **how do we design the controlled response, $\mathbf{u}(t)$, that brings our system from an initial state $\mathbf{x}(0)$ to a desired target state $\mathbf{x}(T)$?** And the great thing about this question is that we already know how to do it because the controlled response is *linear*. By linear, we again mean that for some input $\mathbf{u}_1(t)$ that yields an output $\mathbf{y}_1 = \mathcal{L}(\mathbf{u}_1(t))$, and another input $\mathbf{u}_2(t)$ that yields an output $\mathbf{y}_2 = \mathcal{L}(\mathbf{u}_2(t))$, we have that

$$a\mathbf{y}_1 + b\mathbf{y}_2 = \mathcal{L}(a\mathbf{u}_1 + b\mathbf{u}_2)$$

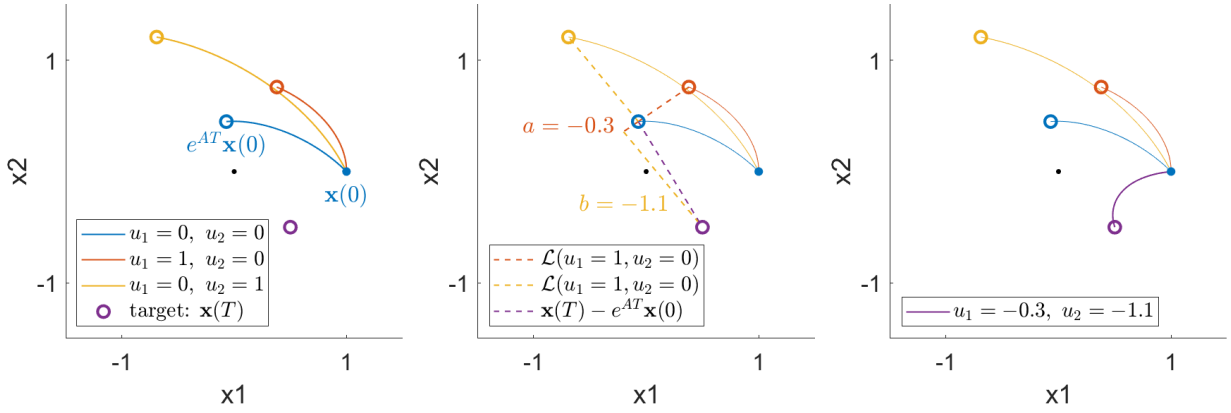
This fact comes from the fact that the convolution operator is *linear*.

a simple 2-state example

So let's try to derive some intuition with the same 2-state example as before, but now our system will have a controlled input such that

$$\underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}-1 & -2\\ 1 & 0\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}x_1\\ x_2\end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}}_{B}\underbrace{\begin{bmatrix}u_1\\ u_2\end{bmatrix}}_{\mathbf{u}}.$$

The natural trajectory of the system is shown as the blue curve, while the first controlled trajectory when $u_1 = 1$ is shown in the red curve, and the second controlled trajectory when $u_2 = 1$ is shown in the yellow curve (left).



Now, we have to be careful about exactly *what* is linear. And the thing that is linear is the convolution operator, $\mathcal{L}(\mathbf{u}) = \int_0^T e^{A(T-t)}B\mathbf{u}(t)\mathrm{d}t$. This operator takes the control input, $\mathbf{u}(t)$, as its input, and outputs the *difference* between the final state, $\mathbf{x}(T)$, and the natural, uncontrolled evolution, $e^{AT}\mathbf{x}(0)$. Hence, we have to speak about the states *relative* to the natural, uncontrolled evolution.

So when we look at the effect of the first controlling input $u_1 = 1$, we are looking at the difference between the controlled final state (open orange circle) from the natural final state (open blue circle). Similarly, when we look at the effect of the second controlling input $u_2 = 1$, we are looking at the difference between the controlled final state (open yellow circle) from the natural final state (open blue circle). And if we want to reach a new target state (open purple circle), we use these differences in controlled trajectories (dashed red and yellow lines) as the basis vectors, and find the weighted sums that yield the difference between the target state and the natural final state (dashed purple line), which yields $u_1 = -0.3, u_2 = -1.1$. And when we control our system using this linear combination of inputs, we see that the trajectory indeed reaches the desired target state (right).

5.3.8 Minimum Energy Control

Of course, this process is all a bit tedious, because we first have to simulate controlled trajectories, then take combinations of those trajectories. Is there a faster and easier way to solve for control inputs that perform a state transition without having to run simulations? The answer is yes, because the controlled response operator $\mathcal{L}(\mathbf{u})$ is linear, but requires a bit of care.

So first, let's think about a typical linear regression problem, $M\mathbf{v} = \mathbf{b}$, where M is an $k \times n$ matrix, $\mathbf{v}$ is an n dimensional vector, and $\mathbf{b}$ is an k dimensional vector,

$$\underbrace{\begin{bmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1n} \\ m_{21} & m_{22} & m_{23} & \cdots & m_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & m_{k3} & \cdots & m_{kn} \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}}_{\mathbf{v}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}}_{\mathbf{b}}.$$

One solution to this regression problem is $\mathbf{v}^* = A^\top(AA^\top)^{-1}\mathbf{b}$, where $A^+ = A^\top(AA^\top)$ is called the pseudoinverse. In fact, this pseudoinverse is quite special, because when a solution to the system of equations exists, $\mathbf{v}^*$ is the *smallest*, or *least squares* solution, where the magnitude is measured simply by the inner product, which in the case of n-dimensional vectors is

$$< \mathbf{a}, \mathbf{b} >_{\mathbb{R}^n} = \mathbf{a}^\top\mathbf{b} = a_1b_1 + a_2b_2 + \cdots + a_nb_n,$$

where the subscript $\mathbb{R}^n$ indicates that the inner product is on the space of n-dimensional vectors. We can extend the exact same equations to our control problem. Explicitly, instead of a matrix M, we will use our control response operator $\mathcal{L}$. Instead of a vector of numbers $\mathbf{v}$, we will use a vector of functions $\mathbf{u}(t)$. And instead of dependent variable $\mathbf{b}$, we will use the state transition $\mathbf{x}(T) - e^{AT}\mathbf{x}_0$. Then the solution to our least squares solution will be

$$\mathbf{u}^*(t) = \mathcal{L}^*(\mathcal{L}\mathcal{L}^*)^{-1}(\mathbf{x}(T) - e^{AT}\mathbf{x}(0)).$$

Now, you may have noticed a slight problem, which has to do with the fact that our inputs are no longer vectors of *numbers*, but rather vectors of *functions*. This problem shows up in the transpose, or adjoint $M^\top$. In our linear regression example, because the operator M is a matrix, it makes sense to take it's tranpose. And this transpose

satisfies an important property, which is that it preserves the *inner product* of input and output vectors. So if $M\mathbf{v}$ is an n-dimensional vector, and $M^\top \mathbf{b}$ is an k-dimensional vector, then $M^\top$ is defined such that

$$\begin{aligned} < M\mathbf{v}, \mathbf{b} >_{\mathbb{R}^k} =& < \mathbf{v}, M^\top \mathbf{b} >_{\mathbb{R}^n} \\ (M\mathbf{v})^\top \mathbf{b} &= \mathbf{v}^\top (M^\top \mathbf{b}) \\ \mathbf{v}^\top M^\top \mathbf{b} &= \mathbf{v}^\top M^\top \mathbf{b} \end{aligned}$$

And when we are thinking about our control response operator $\mathcal{L}$, we can actually do the same thing! First, we see that $\mathcal{L}$ is not mapping vectors to vectors as M, but rather maps *functions* $\mathbf{u}(t)$ to vectors. So we first need to define the inner product of functions, which is simply

$$< \mathbf{a}(t), \mathbf{b}(t) >_{\not\leqslant^k} = \int_0^T \mathbf{a}(t)^\top \mathbf{b}(t) \mathrm{d}t = \int_0^T a_1(t)b_1(t) + a_2(t)b_2(t) + \cdots + a_k(t)b_k(t) \mathrm{d}t,$$

where the subscript $\not\leqslant^k$ indicates that the inner product is on the space of k-dimensional functions. Now, to find the adjoint of operator $\mathcal{L}$, we have to satisfy the same inner product relationship (where we call $\mathbf{b} = \mathbf{x}(T) - e^{AT}\mathbf{x}(0)$ for brevity)

$$\begin{aligned} < \mathcal{L}(\mathbf{u}(t)), \mathbf{b} >_{\mathbb{R}^n} =& < \mathbf{u}(t), \mathcal{L}^*(\mathbf{b}) >_{\not\leqslant^k} \\ \mathcal{L}(\mathbf{u}(t))^\top \mathbf{b} &= \int_0^T \mathbf{u}(t)^\top \mathcal{L}^*(\mathbf{b}) \mathrm{d}t \\ \left(\int_0^T e^{A(T-t)} B\mathbf{u}(t) \right)^\top \mathrm{d}t \mathbf{b} &= \int_0^T \mathbf{u}(t)^\top \mathcal{L}^*(\mathbf{b}) \mathrm{d}t \\ \int_0^T \mathbf{u}(t)^\top B^\top e^{A^\top (T-t)} \mathbf{b} \mathrm{d}t &= \int_0^T \mathbf{u}(t)^\top \mathcal{L}^*(\mathbf{b}) \mathrm{d}t, \end{aligned}$$

and we see that for the left and right sides to be equal, the adjoint must be equal to $\mathcal{L}^* = B^\top e^{A^\top (T-t)}$. Intuitively, this makes sense because if the original operator $\mathcal{L}$ took functions of time as inputs and output a vector of numbers, then the adjoint should take vectors of numbers as inputs and output functions of time. Finally, plugging this adjoint back into our solution, we get

$$\begin{aligned} \mathbf{u}^*(t) &= \mathcal{L}^*(\mathcal{L}\mathcal{L}^*)^{-1}\mathbf{b} \\ &= B^\top e^{A^\top (T-t)} \left(\underbrace{\int_0^T e^{A(T-t)} B B^\top e^{A^\top (T-t)} \mathrm{d}t}_{W_c} \right)^{-1} \mathbf{b}, \end{aligned}$$

where for convenience, we will refer to the bracketed quantity as the *controllability Gramian*. To compute the magnitude of this control input, we simply take the norm of this solution to get

$$\begin{aligned} E^* =< \mathbf{u}^*(t), \mathbf{u}^*(t) > &= \int_0^T e^{A(T-t)} B B^\top e^{A^\top (T-t)} W_c^{-1} \mathbf{b} \, \mathrm{d}t \\ &= \mathbf{b}^\top W_c^{-1} \int_0^T e^{A(T-t)} B B^\top e^{A^\top (T-t)} \mathrm{d}t \, W_c^{-1} \mathbf{b} \\ &= \mathbf{b}^\top W_c^{-1} W_c W_c^{-1} \mathbf{b} \\ &= \mathbf{b}^\top W_c^{-1} \mathbf{b}. \end{aligned}$$

final equations

So here we are! After some derivations, we can compute the control input $\mathbf{u}^*(t)$ that brings our system $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$ from an initial state $\mathbf{x}(0)$ to a final state $\mathbf{x}(T)$ with minimal norm input as

$$\mathbf{u}^*(t) = B^\top e^{A^\top (T-t)} W_c^{-1}(\mathbf{x}(T) - e^{AT}\mathbf{x}(0)),$$

which costs the minimum energy

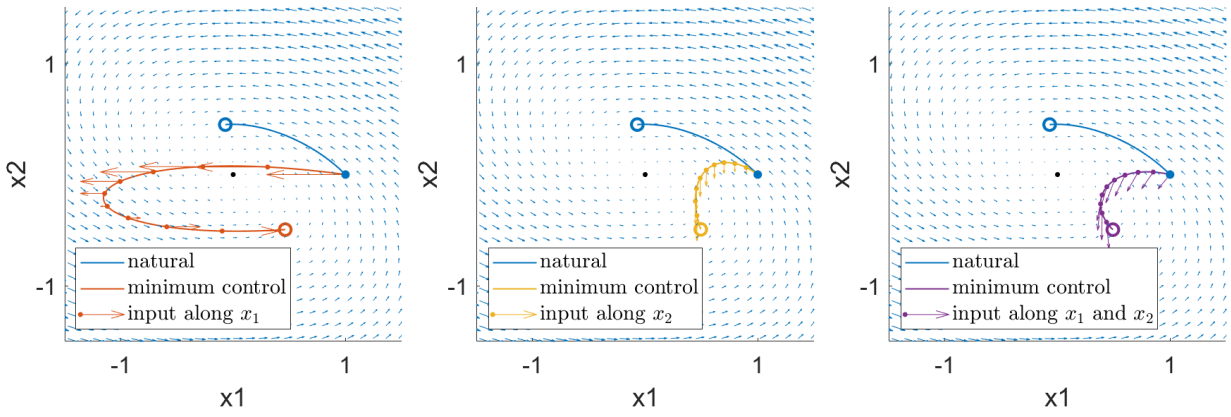
$$E^* = (\mathbf{x}(T) - e^{AT}\mathbf{x}(0))^\top W_c^{-1}(\mathbf{x}(T) - e^{AT}\mathbf{x}(0))$$

a simple 2-dimensional example

To provide a bit of intuition for the control process, we look again at our simple 2-dimensional linear example, but with only one control input, $u_1(t)$, along the x_1 direction

$$\underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}-1 & -2\\ 1 & 0\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}x_1\\ x_2\end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix}1\\ 0\end{bmatrix}}_{B}\underbrace{\begin{bmatrix}u_1\end{bmatrix}}_{\mathbf{u}}.$$

This means that we can only push our system along the x_1 direction, and have to rely on the internal dynamics to change the x_2 state. The natural trajectory (blue), controlled trajectory (orange), and control input (orange arrows) are shown in the left subplot below.



We observe that the state takes a rather roundabout trajectory to reach the target state, because the only way for the system state x_2 to move downward is to push the state x_1 to a regime where the natural dynamics allow x_2 to decrease. Now, if we define a different control system where we can only influence the dynamics along the x_2 state such that

$$\underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}-1 & -2\\ 1 & 0\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}x_1\\ x_2\end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix}0\\ 1\end{bmatrix}}_{B}\underbrace{\begin{bmatrix}u_1\end{bmatrix}}_{\mathbf{u}},$$

then we get the trajectory in the center subplot. Notice that the dynamics don't push the system straight downard, but rather follows the natura dynamics upwards for a while before moving downard. This is because it costs less energy (input) to fight the weaker natural upward dynamics near the center of the vector field, as opposed to fighting the stronger natural upward dynamics near the right of the vector field.

Finally, if we are able to independently influence both of the system states,

$$\underbrace{\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix}-1 & -2\\ 1 & 0\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}x_1\\ x_2\end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}}_{B}\underbrace{\begin{bmatrix}u_1\\ u_2\end{bmatrix}}_{\mathbf{u}},$$

then we get the controlled trajectory and inputs in the right subplot.

5.4 Numerics

When computing control energy, we mean that for a linear dynamical system of the form

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

we are computing the amount of energy it costs to bring the system from an initial state $\mathbf{x}(0)$ to a final state $\mathbf{x}(T)$ in T time through a set of input functions $\mathbf{u}(t)$. This cost is called the *control energy*, and is defined by

$$E(\mathbf{u}(t)) = \int_0^T \mathbf{u}^\top(t)\mathbf{u}(t)\mathrm{d}(t) = \int_0^T u_1^2(t) + u_2^2(t) + \cdots + u_k^2(t)\mathrm{d}t.$$

From the theory, we found that the *minimum control energy* is given by

$$E^* = (\mathbf{x}(T) - e^{AT}\mathbf{x}(0))^\top W_c^{-1}(\mathbf{x}(T) - e^{AT}\mathbf{x}(0)),$$

where W_c is the *controllability Gramian*. Here, we will discuss how to numerically evaluate the theoretical relationships. Because the specific states $\mathbf{x}(0)$ and $\mathbf{x}(T)$ are just vectors, we will focus on the terms e^{AT} and W_c.

5.4.1 The Matrix Exponential

The matrix exponential has a long history in both analytical and numerical applications. But what *is* the exponential of a matrix? Fortunately, there is a relatively simple way to conceptualize it through a Taylor series expansion:

$$e^A = I + A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \cdots + \frac{1}{n!}A^n + \cdots = \sum_{n=0}^{\infty} \frac{1}{n!}A^n.$$

So in one sense, the matrix exponential is just a weighted sum of powers of A, so nothing too fancy. Most numerical packages will have built in functions for the evaluation of matrix exponentials. Please note that the matrix exponential is *not* simply an element-wise exponential of A. As we can see in the series expansion, the matrix exponential contains matrix powers of A, which are very different from element-wise operations.

5.4.2 The Controllability Gramian

Of great importance to linear network control is the controllability Gramian, defined in the theory as

$$W_c = \int_0^T e^{A(T-t)}BB^\top e^{A^\top(T-t)}\mathrm{d}t.$$

We can simplify this expression a bit through a simple change of variables $\tau = T - t$ (classic exercise left to reader) to write

$$W_c = \int_0^T e^{A\tau}BB^\top e^{A^\top\tau}\mathrm{d}\tau = \int_0^T f(\tau)\mathrm{d}\tau.$$

So… how do we turn this equation into a matrix on our computer? Well, it is no different than any other form of numerical integration! It just looks a bit scary because of the matrices, but never fear. Let us start with a very simple approach.

built-in numerical integrators

Perhaps the most simple approach would be to use built-in numerical integrators from various sources, whether it be Python, MATLAB, Mathematica, etc. In MATLAB, the command would look something like

```
T = 1;
Wc = f(t) = integral(@(t) (expm(A*t)*B)*(expm(A*t)*B)',0,T,'ArrayValued',1);
```

right-hand Riemann sum

Provided our particular numerical package lacks a matrix-valued numerical integrator, we can use the most basic integration scheme: the right-hand Riemann sum. The basic idea is that rather than compute the exact area under the curve, we can just evaluate the curve at different points in time, and pretend the function is constant in between points. So if we break up our integration into time steps of ΔT, then our Gramian approximately evaluates to

$$W_c \approx \sum_{n=0}^{T/\Delta T-1} e^{An\Delta T} BB^\top e^{A^\top n\Delta T} \Delta T.$$

Because we assume our numerical package can evaluate matrix integrals and perform matrix multiplications, we can perform this summation without much issue. We can speed up this process by recognizing that the product of matrix exponentials yields the sum of their exponents (when the matrices in the exponentials commute), such that

$$e^{An\Delta T} e^{A\Delta T} = e^{A(n+1)\Delta T}.$$

Hence, we only ever actually have to evaluate *one* matrix exponential, $e^{A\Delta T}$, and can obtain all subsequent matrix exponentials by accumulating products of matrix exponentials.

Simpson's rule

While the Riemann sum is simple, it is also prone to errors if the function being integrated changes too quickly with respect to the time step, and might require too small of a time step ΔT. Now, taking $T/\Delta T$ products of matrices shouldn't be too computationally expensive given the system is not too large. However, another issue arises when $A\Delta T$ becomes too small to evaluate to numerical precision. For example, if we require 10,000 time steps to accurately capture the curvature of the matrix exponential, then we need to accurately compute $e^{A/10{,}000}$, and then accurately multiply those very small matrices 10,000 times. This approach can lead to the exonential growth of numerical precision errors.

Instead, we can use Simpson's rule, which essentially uses higher-order polynomials to fit the curvature of functions. So, rather than assuming the function stays constant at each sampled point as in the Riemann sum, we instead fit polynomials, which ultimately evaluates to

$$W_c \approx \frac{\Delta T}{3}\left(f(0) + 2\sum_{n=1}^{\frac{T}{2\Delta T}-1} f(2n\Delta T) + 4\sum_{j=1}^{\frac{T}{2\Delta T}} f((2n-1)\Delta T) + f(T)\right).$$

More advanced versions of this polynomial integration scheme can be found in the Newton-Cotes formulas.

5.4.3 Evaluating Minimum Control Energy

Now, once we have our controllability Gramian and state transitions, we evaluate the minimum control energy using

$$E^* = (\mathbf{x}(T) - e^{AT}\mathbf{x}(0))^\top W_c^{-1}(\mathbf{x}(T) - e^{AT}\mathbf{x}(0)).$$

But let's pause for a moment here. Notice that the controllability Gramian is *only* a function of the connectivity matrix A, the input matrix B, and the time horizon T as we reproduce below

$$W_c = \int_0^T e^{A\tau} B B^\top e^{A^\top \tau} \mathrm{d}\tau.$$

What this means is that for any analysis that involves assessing *many* state transitions for one set of system parameters A, B, T, we only have to compute the Gramian *once*, and invert the Gramian *once*. After obtaining W_c^{-1}, we can evaluate all of the energies for all state transitions through simple matrix multiplications, which are computationally way more efficient.

We can take this idea one step further and notice that the matrix exponential, e^{AT}, also only has to be evaluated *once*. Hence, as a tip to our readers, it is likely going to be much more computationally efficient to evaluate and store the Gramians and matrix exponentials, then batch together the state transitions.

5.5 Examples

5.5.1 Different approaches to computing minimum control energy

As discussed in *Theory*, there is more than one way to estimate the minimum control energy associated with a set of state transitions. Here, we compare our standard approach for calculating minimum control energy to an approach that leverages a shortcut based on vectorization. **"why would I want to calculate control energy this way?"** I hear you ask. The answer is simple: speed! As you'll see below, the vectorization approximation provides highly convergent estimates of energy ~300 times faster than the standard approach.

The data used here are structural connectomes taken from the Philadelphia Neurodevelopmental Cohort.

Here, our Python workspace contains a single structural connectome stored in `A`, a `numpy.array` with 200 nodes along dimensions 0 and 1.

```
print(A.shape)
```

```
Out:
(200, 200)
```

Before we calculate minimum control energy, we first need to define some brain states. Here, we'll use binary brain states, where each state comprises a set of brain regions that are designated as "on" (activity = 1) while the rest of the brain is "off" (activity = 0). Just for illustrative purposes, we'll define these brain states arbitrarily by grouping the rows/columns of `A` into equally-sized non-overlapping subsets of regions.

```
# setup states
n_nodes = A.shape[0]
n_states = int(n_nodes/10)
state_size = int(n_nodes/n_states)

states = np.array([])
for i in np.arange(n_states):
    states = np.append(states, np.ones(state_size) * i)
states = states.astype(int)
```

The above code simply generates a vector of integers, stored in `states`, that designates which of 20 states each brain region belongs to. Owing to the fact that `n_nodes` equals 200 here, each state comprises 10 nodes. Note, no nodes are assigned to multiple states.

```
print(states)
```

```
Out:
[ 0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2
  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4
  4  4  5  5  5  5  5  5  5  5  5  5  6  6  6  6  6  6  6  6  6  6  7  7
  7  7  7  7  7  7  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9
  9  9  9  9 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 14 14 14 14
 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16
 16 16 17 17 17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 19 19
 19 19 19 19 19 19 19 19]
```

So the first 10 nodes of `A` belong to state 0, the next 10 to state 1, and so on and so forth. Using these states, we'll first compute the minimum control energy required to transition between all possible pairs using our standard function, `nctpy.energies.get_control_inputs()`.

```
from nctpy.utils import matrix_normalization
from nctpy.energies import get_control_inputs, integrate_u

# settings
# time horizon
T = 1
# set all nodes as control nodes
B = np.eye(n_nodes)
# normalize A matrix for a continuous-time system
system = 'continuous'
A_norm = matrix_normalization(A, system=system)

import time
from tqdm import tqdm
start_time = time.time()  # start timer

# settings for minimum control energy
S = np.zeros((n_nodes, n_nodes))  # x is not constrained
xr = 'zero'  # x and u constrained toward zero activity

e = np.zeros((n_states, n_states))
for i in tqdm(np.arange(n_states)):
    x0 = states == i  # get ith initial state
    for j in np.arange(n_states):
        xf = states == j  # get jth target state
        x, u, n_err = get_control_inputs(A_norm=A_norm, T=T, B=B, x0=x0, xf=xf,␣
↪system=system, xr=xr, S=S,
                                         expm_version='eig')  # get control inputs using␣
↪minimum control
        e[i, j, :] = np.sum(integrate_u(u))

e = e / 1000  # divide e by 1000 to account for dt=0.001 in get_control_inputs
```

(continues on next page)

(continued from previous page)

```
end_time = time.time()  # stop timer
elapsed_time = end_time - start_time
print('time elapsed in seconds: {:.2f}'.format(elapsed_time))  # print elapsed time
```

```
Out:
100%|| 20/20 [01:35<00:00,  4.78s/it]
time elapsed in seconds: 95.68
```

The standard approach took ~95 seconds to calculate the control energy associated with completing 400 (20 x 20) state transitions. Now we'll compare that to our alternative approach, which is implemented in `nctpy.energies.minimum_energy_fast()`.

In order to use this variant of minimum control energy, we first have to use our `nctpy.utils.expand_states()` function to convert `states` into a pair of boolean matrices, `x0_mat` and `xf_mat`, that together encode all possible pairwise state transitions.

```
from nctpy.utils import expand_states
x0_mat, xf_mat = expand_states(states)
print(x0_mat.shape, xf_mat.shape)
```

```
Out:
(200, 400) (200, 400)
```

The rows of `x0_mat` and `xf_mat` correspond to the nodes of our system and the columns correspond to the states we defined above. Critically, `x0_mat` and `xf_mat` are paired; if you take the same column across both matrices you will end up with the initial state (`x0_mat[:, 0]`) and the target state (`xf_mat[:, 0]`) that comprise a specific **state transition**. Note, `nctpy.utils.expand_states()` only works for binary brain states. If you have non-binary brain states you'll have to create `x0_mat` and `xf_mat` on your own. Equipped with these state transition matrices, let's compute energy again!

```
from nctpy.energies import minimum_energy_fast

start_time = time.time()  # start timer

e_fast = minimum_energy_fast(A_norm=A_norm, T=T, B=B, x0=x0_mat, xf=xf_mat)
e_fast = e_fast.transpose().reshape(n_states, n_states, n_nodes)
e_fast = np.sum(e_fast, axis=2)  # sum over nodes

end_time = time.time()  # stop timer
elapsed_time = end_time - start_time
print('time elapsed in seconds: {:.2f}'.format(elapsed_time))  # print elapsed time
```

```
Out:
time elapsed in seconds: 0.29
```

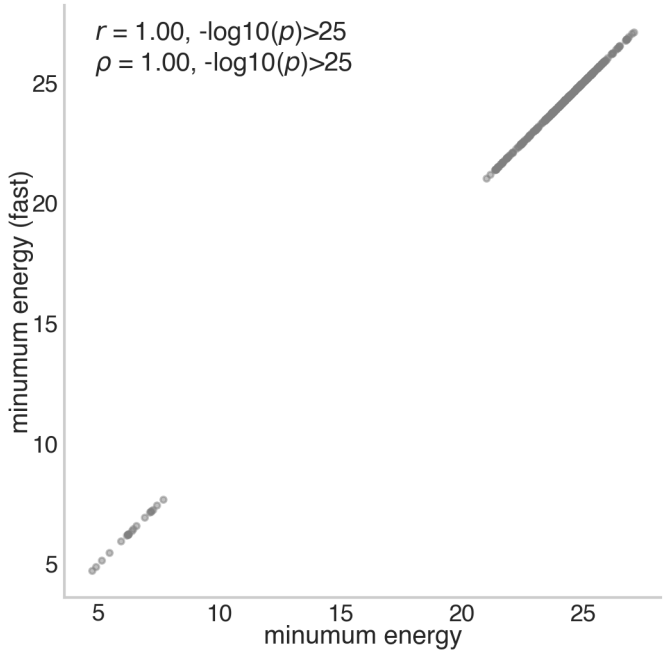
This time we managed to compute all of our transition energies in less than half a second! So our vectorization approach is fast, but is it equivalent?

```
print(np.max(e - e_fast))
```

```
Out:
2.7267077484793845e-11
```

The largest difference between energies is tiny! Great, they're pretty much the same. Let's also visualize the energies using a correlation plot for good measure.

```
import matplotlib.pyplot as plt
from nctpy.plotting import set_plotting_params, reg_plot
set_plotting_params()

# plot
f, ax = plt.subplots(1, 1, figsize=(4, 4))

# correlation between whole-brain energy across state transitions
reg_plot(x=e.flatten(), y=e_fast.flatten(), xlabel='minumum energy', ylabel='minumum␣
↪energy (fast)',
         ax=ax, add_spearman=True, kdeplot=False, regplot=False)
plt.show()
```



Note, there are several caveats to consider when using the above approach. First, `nctpy.energies.minimum_energy_fast()` only works for continuous time systems. Second, the function does not output the control inputs (`u`), state trajectory (`x`), or the numerical errors (`n_err`) provided by `nctpy.energies.get_control_inputs()`. That is, you will *only* get node-level energy. Finally, as mentioned above, `nctpy.utils.expand_states()` only works for binary brain states. This is trivial however; if you have non-binary brain states you'll just have to create `x0_mat` and `xf_mat` on your own.

5.5.2 Relationships between regional Network Control Theory metrics

Note: Relevant publication: Gu et al. 2015 Nature Communications

In this example, we illustrate how average and modal controllability correlate to weighted degree (strength) and to each other. The data used here are structural connectomes taken from the Philadelphia Neurodevelopmental Cohort.

Here, our Python workspace contains subject-specific structural connectomes stored in `A`, a `numpy.array` with 200 nodes along dimensions 0 and 1 and 253 subjects along dimension 3:

```
print(A.shape)
```

```
Out:
(200, 200, 253)
```

With these data, we'll start by calculating weighted degree (strength), average controllability, and modal controllability for each subject:

```
# import
from tqdm import tqdm
from nctpy.utils import matrix_normalization
from nctpy.metrics import ave_control, modal_control

n_nodes = A.shape[0] # number of nodes (200)
n_subs = A.shape[2] # number of subjects (253)

# strength function
def node_strength(A):
    str = np.sum(A, axis=0)

    return str

# containers
s = np.zeros((n_subs, n_nodes))
ac = np.zeros((n_subs, n_nodes))
mc = np.zeros((n_subs, n_nodes))

# define time system
system = 'discrete'

for i in tqdm(np.arange(n_subs)):
    a = A[:, :, i]  # get subject i's A matrix
    s[i, :] = node_strength(a)  # get strength

    a_norm = matrix_normalization(a, system=system)  # normalize subject's A matrix
    ac[i, :] = ave_control(a_norm, system=system)  # get average controllability
    mc[i, :] = modal_control(a_norm)  # get modal controllability
```

Then we'll average over subjects to produce a single estimate of weighted degree, average controllability, and modal controllability at each node:

```
# mean over subjects
s_subj_mean = np.mean(s, axis=0)
```

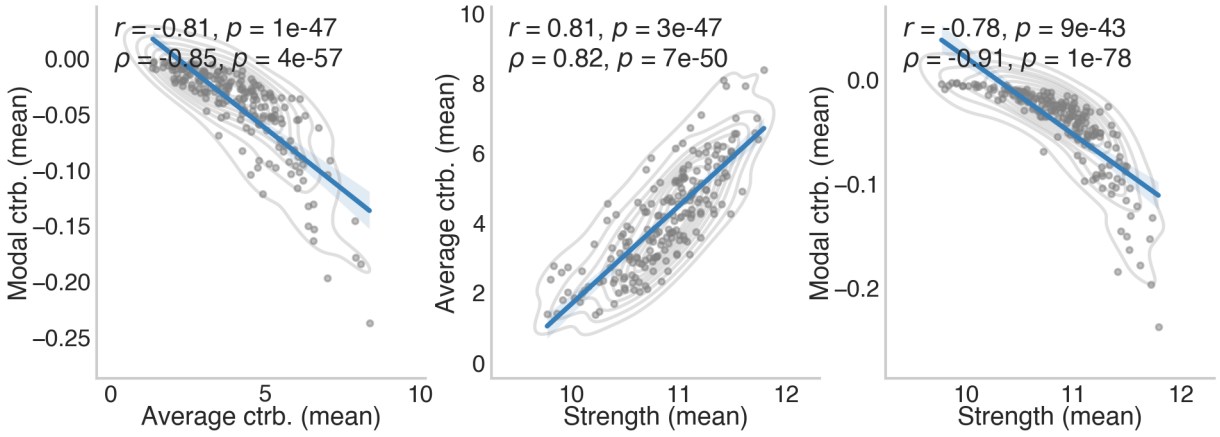
(continues on next page)

(continued from previous page)

```
ac_subj_mean = np.mean(ac, axis=0)
mc_subj_mean = np.mean(mc, axis=0)

# take log transform to normalize
s_subj_mean = np.log(s_subj_mean)
ac_subj_mean = np.log(ac_subj_mean)
mc_subj_mean = np.log(mc_subj_mean)
```

Lastly, we'll plot the relationship between metrics:

```
import matplotlib.pyplot as plt
from nctpy.metrics import ave_control, modal_control
from nctpy.plotting import set_plotting_params, reg_plot
set_plotting_params()

f, ax = plt.subplots(1, 3, figsize=(7.5, 2.5))
reg_plot(x=ac_subj_mean, y=mc_subj_mean,
         xlabel='Average ctrb. (mean)', ylabel='Modal ctrb. (mean)',
         add_spearman=True, ax=ax[0])
reg_plot(x=s_subj_mean, y=ac_subj_mean,
         xlabel='Strength (mean)', ylabel='Average ctrb. (mean)',
         add_spearman=True, ax=ax[1])
reg_plot(x=s_subj_mean, y=mc_subj_mean,
         xlabel='Strength (mean)', ylabel='Modal ctrb. (mean)',
         add_spearman=True, ax=ax[2])
plt.show()
```



The above results are consistent with Gu et al. 2015 (see Figure 2).

5.5.3 Effect of development on average and modal controllability

Note: Relevant publication: Tang et al. 2017 Nature Communications

In this example, we illustrate how average and modal controllability vary as a function of age in a developing sample. The data used here are structural connectomes taken from the Philadelphia Neurodevelopmental Cohort.

Here, our Python workspace contains subject-specific structural connectomes stored in `A`, a `numpy.array` with 200 nodes along dimensions 0 and 1 and 769 subjects along dimension 3:

```
print(A.shape)
```

```
Out:
(200, 200, 769)
```

We also have demographic data stored in `df`, a `pandas.dataframe` with subjects along dimension 0. Let's take a peek at age, which is stored in months:

```
print(df['ageAtScan1'].head())
```

```
Out:
0    240
1    232
2    231
3    249
4    234
Name: ageAtScan1, dtype: int64
```

With these data, we'll start by calculating average and modal controllability for each subject:

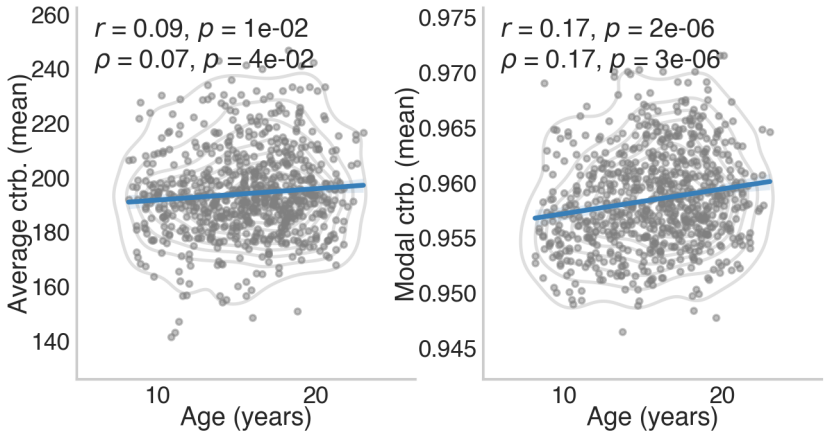
```
# import
from tqdm import tqdm
from nctpy.utils import matrix_normalization
from nctpy.metrics import ave_control, modal_control

n_nodes = A.shape[0] # number of nodes (200)
n_subs = A.shape[2] # number of subjects (769)

# containers
ac = np.zeros((n_subs, n_nodes))
mc = np.zeros((n_subs, n_nodes))

# define time system
system = 'discrete'

for i in tqdm(np.arange(n_subs)):
    a = A[:, :, i]  # get subject i's A matrix

    a_norm = matrix_normalization(a, system=system)  # normalize subject's A matrix
    ac[i, :] = ave_control(a_norm, system=system)  # get average controllability
    mc[i, :] = modal_control(a_norm)  # get modal controllability
```

Then we'll average over nodes to produce estimates of whole-brain average and modal controllability for each subject:

```
# mean over nodes
ac_node_mean = np.mean(ac, axis=1)
mc_node_mean = np.mean(mc, axis=1)
```

Lastly, we'll plot the relationship between age and each metric:

```
f, ax = plt.subplots(1, 2, figsize=(5, 2.5))
reg_plot(x=df['ageAtScan1']/12, y=ac_node_mean,
         xlabel='Age (years)', ylabel='Average ctrb. (mean)',
         add_spearman=True, ax=ax[0])
reg_plot(x=df['ageAtScan1']/12, y=mc_node_mean,
         xlabel='Age (years)', ylabel='Modal ctrb. (mean)',
         add_spearman=True, ax=ax[1])
plt.show()
```



The above figure shows that whole-brain average and modal controllability both increase throughout development. This is consistent Tang et al. 2017 (see Figure 2c for average controllability).

5.5.4 Inter-metric coupling across the principal gradient of functional connectivity

Note: Relevant publication: Parkes et al. 2021 Biological Psychiatry

In this example, we illustrate how the cross-subject correlations between weighted degree and average controllability, as well as between weighted degree and modal controllability, vary as a function of the principal cortical gradient of functional connectivity. The data used here are structural connectomes taken from the Philadelphia Neurodevelopmental Cohort.

Here, our Python workspace contains subject-specific structural connectomes stored in `A`, a `numpy.array` with 200 nodes along dimensions 0 and 1 and 1068 subjects along dimension 3:

```
print(A.shape)
```

```
Out:
(200, 200, 1068)
```

We also have `gradient`, a `numpy.array` that designates where each of the 200 regions in our parcellation are situated along the cortical gradient:

```
print(gradient.shape)
```

```
Out:
(200,)
```

With these data, we'll start by calculating weighted degree (strength), average controllability, and modal controllability for each subject:

```
# import
from tqdm import tqdm
from nctpy.utils import matrix_normalization
from nctpy.metrics import ave_control, modal_control

n_nodes = A.shape[0] # number of nodes (200)
n_subs = A.shape[2] # number of subjects (1068)

# strength function
def node_strength(A):
    str = np.sum(A, axis=0)

    return str

# containers
s = np.zeros((n_subs, n_nodes))
ac = np.zeros((n_subs, n_nodes))
mc = np.zeros((n_subs, n_nodes))

# define time system
system = 'discrete'

for i in tqdm(np.arange(n_subs)):
    a = A[:, :, i]  # get subject i's A matrix
    s[i, :] = node_strength(a)  # get strength

    a_norm = matrix_normalization(a, system=system)  # normalize subject's A matrix
    ac[i, :] = ave_control(a_norm, system=system)  # get average controllability
    mc[i, :] = modal_control(a_norm)  # get modal controllability
```

Next, for each region, we'll calculate the cross-subject correlation between strength and average/modal controllability:

```
# compute cross subject correlations
corr_s_ac = np.zeros(n_nodes)
corr_s_mc = np.zeros(n_nodes)

for i in tqdm(np.arange(n_nodes)):
    corr_s_ac[i] = sp.stats.spearmanr(s[:, i], ac[:, i])[0]
    corr_s_mc[i] = sp.stats.spearmanr(s[:, i], mc[:, i])[0]
```

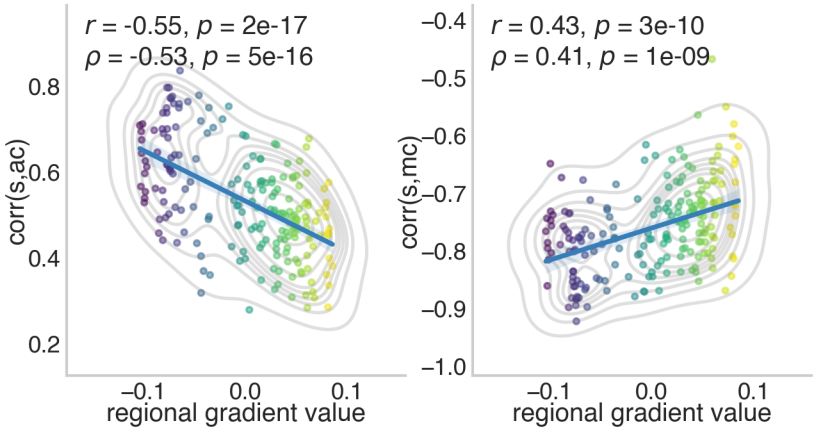
Plotting time! Below we illustrate how the above correlations vary over the cortical gradient spanning unimodal to transmodal cortex:

```
f, ax = plt.subplots(1, 2, figsize=(5, 2.5))
reg_plot(x=gradient, y=corr_s_ac,
```

(continues on next page)

(continued from previous page)

```
        xlabel='regional gradient value', ylabel='corr(s,ac)',
        add_spearman=False, ax=ax[0], c=gradient)
reg_plot(x=gradient, y=corr_s_mc,
        xlabel='regional gradient value', ylabel='corr(s,mc)',
        add_spearman=False, ax=ax[1], c=gradient)
plt.show()
```



The above shows that the cross-subject correlations between strength and both average and modal controllability get weaker as regions traverse up the cortical gradient. The results for average controllability can also be seen in Figure 3a of Parkes et al. 2021.

5.5.5 Replication from White Matter Network Architecture Guides Direct Electrical Stimulation through Optimal State Transitions

Note: Relevant publication: Stiso et al. 2019 Cell Reports

Direct electrical brain stimulation is a useful treatment for some neurological disorders (most famously, Parkinson's). However, we still don't really have a good way of predicting how stimulation at a single region will influence the rest of the brain. Here, we hypothesize that input from stimulation will spread linearly along white matter tracts to impact brain activity at distant regions. We use a dataset of combined iEEG, DWI, and stimulation data to simulate stimulation based on this hypothesis.

In this example, we're specifically going to ask if, given this model, it takes more stimulation energy to travel to more cognitively distant states. We're going to define cognitive distance with the output from a classifier that guesses memory performance from brain states.

Data consist of one DWI structural adjacency matrix per dataset, and a variable number of stimulation trials. Each stimulation trial includes the brain state before and after stimulation.

The data used here are taken from the Restoring Active Memory Project.

Let's take a look at the data:

```
import scipy.io as sio
workdir = '/Users/stiso/Documents/Code/test_data/'
subj = ['R1067P.PS1_set1','R1051J.PS1_set1','R1051J.PS1_set2.1', 'R1030J.PS1_set1',
↪'R1124J_1.PS2_set2',
```

(continues on next page)

(continued from previous page)

```
        'R1124J_1.PS2_set3','R1124J_1.PS2_set1', 'R1150J.PS2_set1','R1150J.PS2_set2',
↪'R1170J_1.PS2_1_set1',
        'R1170J_1.PS2_1_set2','R1067P.PS1_set1','R1068J.PS1_set2.1','R1089P.PS2_set1',
↪'R1062J.PS1_set1',
        'R1173J.PS2_1_set1','R1066P.PS1_set1'];
dat = sio.loadmat(f'{workdir}Trajectory_Data.{subj[0]}.mat')
dat.keys()
```

```
Out:
dict_keys(['__header__', '__version__', '__globals__', 'Electrode_ROI', 'Post_Freq_State
↪', 'Post_Stim_Prob',
'Pre_Freq_State', 'Pre_Stim_Prob', 'Stim_Amp', 'Stim_Duration', 'Stim_Freq', 'Stim_Loc_
↪Idx', 'Struct_Adj'])
```

Here, the first important variable is `Struct_Adj`, which is our structural adjacency matrix ($\mathbf{A}$). Second is `Pre_Freq_State`, which is the band limited power in 8 frequencies, in all iEEG contacts, for each stimulation trial. This is our estimate of an initial brain state ($x0$). Because we have different frequency bands, we're basically going to average across all of them to get one energy estimate (an alternative to this would be to vectorize all the frequency bands into one). The next important variable is `Pre_Stim_Prob`, which is the cognitive state at the start of every trial. High values indicate that someone is in a good cognitive state, or highly likely to encode a memory. The last useful variables are `Stim_Loc_Idx`, which tells us which contacts were used to stimulate on each trial, and `Electrode_ROI`, which helps us map between the iEEG contacts and the structural adjacency atlas.

If we're gong to use *optimal control*, we're missing a target state. We're going use a *good memory* state as the target state. This is defined as an average of the 5% of states with the best classifier predictions for memory performance.

```
prefix = subj[0].split('_set')[0]
target = sio.loadmat(f'{workdir}targets/{prefix}.mat')
target.keys()
```

```
Out:
dict_keys(['__header__', '__version__', '__globals__', 'ROI_target'])
```

Now that we know what our data are, we can set the parameters for our optimal control model. These specific parameters were chosen to minimize the error of the optimal control calculation.

```
# balance between minimizing energy or minimizing distance from target state
rho = .2
# time to go from initial to target state
T = .7
# the number of time points the code spits out: T * 1000 + 1
nTime = 701
gamma = 4
# to try and simulate stimuluation, we're gonna weight the B matrix
B_mu = .0005
B_sigma = .00005
```

You'll notice that there are a few new parameters here too, `B_mu` and `B_sigma`. Using optimal control to model stimulation with iEEG data presents some unique problems that these parameters help address. The first challenge is that we want to have input concentrated at stimulation electrodes, but having a sparse B matrix leads to very high error. The second challenge is that we dont have iEEG contacts in every region of the brain, so we have to guess at what those regions activity levels are. We therefore set the activity at all regions that dont have contacts to 1 (see the supplementary materials of Stiso et al. for evidence that the exact number doesn't matter), but we also don't want to waste a bunch

of energy keeping these state values at 1 when we don't really care about them. To address both these problems, we set the diagonal entries in B without corresponding electrodes to very small values drawn from a normal distribution defined by `B_mu` and `B_sigma`. This way, our B matrix is less sparse, our model will yield lower error, and states that we dont have recordings for will be able to regulate themselves.

Now we're ready to get the optimal input required to go from any given starting state, to a good memory state. We expect that it is going to take more energy to go from bad to good states than good to good states.

```
from nctpy.utils import matrix_normalization
from nctpy.energies import get_control_inputs, integrate_u
import numpy as np
import pandas as pd
np.random.seed(0)

# intiialize final data strcutre
energies = pd.DataFrame(columns=['energy', 'condition', 'subject', 'trial', 'error'])

for i, s in enumerate(subj):
    prefix = subj[0].split('_set')[0]
    # load in data
    dat = sio.loadmat(f'{workdir}Trajectory_Data.{s}.mat')
    target = sio.loadmat(f'{workdir}targets/{prefix}.mat')

    # subject specific constants
    # number of stim trials for this set
    nTrial = np.size(dat['Post_Freq_State'], 0)
    # number of nodes/regions in the atlas we are using - one of the complications of␣
↪this project is that we
    # dont have iEEG/state data for every regions of the atlas
    nROI = np.size(dat['Post_Freq_State'], 1)
    # number of bands
    nFreq = np.size(dat['Post_Freq_State'], 2)
    # these are the regions with contacts
    elec_idx = np.sum(dat['Post_Freq_State'][:, :, 0], 0) != 0
    ROI_idx = [not x for x in elec_idx]
    # number of contacts
    nElec = sum(elec_idx)
    # stim contacts
    stim_idx = [x[0][0] for x in dat['Stim_Loc_Idx']]

    # which regions we want to constrain the state of
    S = np.eye(nROI)

    # scale A matrix (continuous)
    # this variable will be the same for both datasets
    A = dat['Struct_Adj']
    A = matrix_normalization(A, c=gamma, system='continuous')

    # get optimal input and trajectory for each trial
    # each participant has a "good memory state", as determined by a linear classifier␣
↪trained on memory performance
    xf = target['ROI_target']
    # this will take a while
    for t in range(nTrial):
```

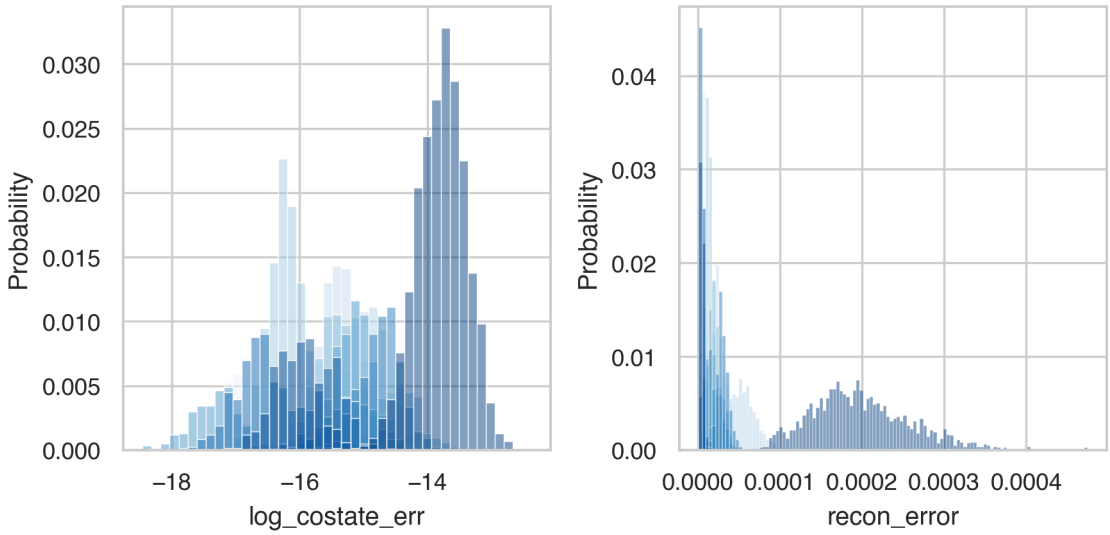
(continues on next page)

(continued from previous page)

```
        # get stim contacts
        e = stim_idx[t]

        # set sparse B matrix - ultimate goal is to have the majority of input be at the␣
↪stim elecs
        # first, we set small input everywhere
        B = np.eye(nROI) * np.random.normal(loc=B_mu, scale=B_sigma, size=(1, nROI))
        # then we add 0s to all the areas whos activity we know
        B[elec_idx, elec_idx] = 0
        # then, we add big numbers to the stim elecs
        for c in e:
            B[c, c] = 1

        # get states
        x0 = np.squeeze(dat['Pre_Freq_State'][t, :, :])

        # add 1s to regions without elecs
        x0[ROI_idx, :] = 1

        # concatenate across frequency bands
        u = np.zeros((nROI, nTime, nFreq))
        err = np.zeros((1, nFreq))
        for f in range(nFreq):
            _, curr_u, curr_err = get_control_inputs(A, T, B, x0[:, f], xf[:,f],
↪'continuous', rho, S)

            curr_u = curr_u.T
            err[:, f] = curr_err
            u[:, :, f] = curr_u

        # get summary of optimal input
        # we incorporated the B matrix into our input summary because of the weighting
        # we use the term energy to be consistent with other literature, but in some␣
↪sense this is a different summary statistic
        u = sum(np.linalg.norm(u.T*np.diag(B), axis=(0, 2)))/nTime

        # average over frequencies
        err = np.mean(err)

        # add to data frame (averaged over freqs)
        curr = pd.DataFrame({'energy':[np.mean(u)],
                             'initial_mem_state': dat['Pre_Stim_Prob'][0][t],
                             'subject':[s],
                             'trial':[t],
                             'error':[err]})
        energies = pd.concat([energies, curr], sort=False)

energies['log_eng'] = np.log(energies['energy'])
energies['log_err'] = np.log(energies['error'])
```

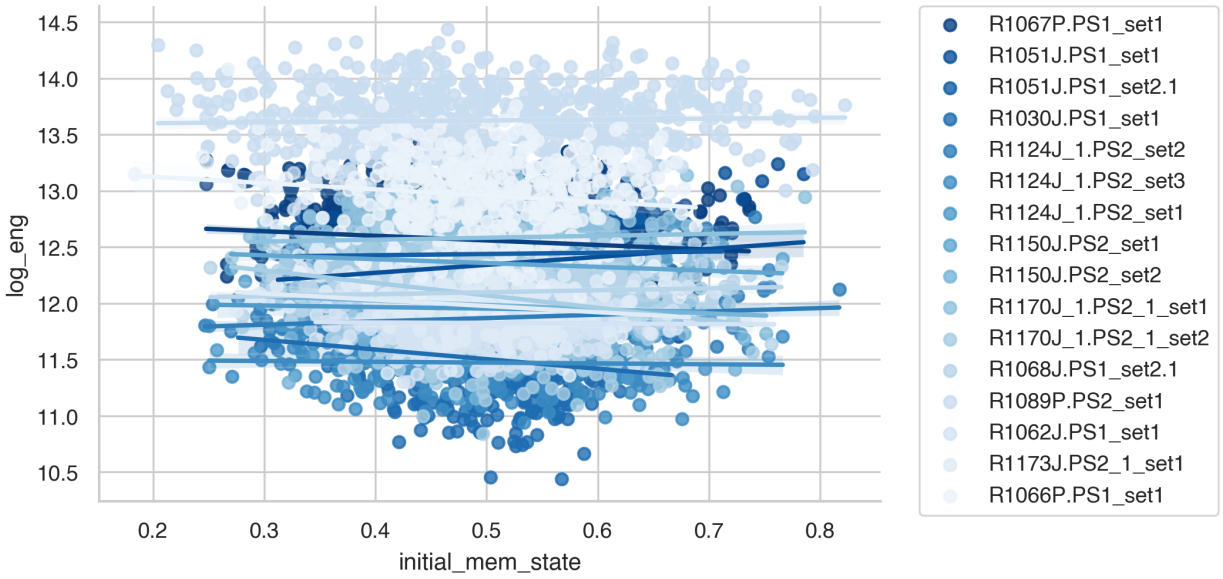
After a long time this will finish. Let's first demonstrate that we have small error, since we went through a lot of trouble to make sure that was the case.

```
import seaborn as sns
import matplotlib.pyplot as plt
from nctpy.plotting import set_plotting_params
set_plotting_params()

fig, ax = plt.subplots(1, 2, figsize=(6, 3))
sns.histplot(energies, x='log_costate_err', hue='subject', stat='probability',
             ax=ax[0], palette='Blues_r', legend=False)
sns.histplot(energies, x='recon_error', hue='subject', stat='probability',
             ax=ax[1], palette='Blues_r', legend=False)
plt.tight_layout()
plt.show()
```



All the different datasets are in different shades of blue. And we can see here that everyone has low error values. Now lets check our actual hypothesis.

```
sns.lmplot(data=energies, y='log_eng', x='initial_mem_state', hue='subject', palette=
↪'Blues_r',
           height=3, aspect=2, legend=False)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



This plot looks a little different from the one in the paper because we don't normalize the output. But as we can see, for most participants, transitions to good memory states require more energy when starting from a poorer memory state. In the paper, we show that the initial memory state explains more variance than the Euclidean distance between states as well.

5.6 References

When using nctpy, please cite the following:

Useful introductions:

User guide targeted at researchers in neuroscience and psychology

- Karrer, T. M., Kim, J. Z., Stiso, J., Kahn, A. E., Pasqualetti, F., Habel, U., & Bassett, D. (2020). A practical guide to methodological considerations in the controllability of structural brain networks. Journal of Neural Engineering.

Introduction to relevant linear systems theory

- Kim, J. Z., & Bassett, D. S. Linear dynamics & control of brain networks. arXiv (2019).

First publication applying control metrics used in this package to human structural brain data

- Gu, S., Pasqualetti, F., Cieslak, M. et al. Controllability of structural brain networks. Nature Communications (2015).

First publication applying optimal control energy used in this package to human structural brain data

- Gu, S., Betzel R. F., Mattar, M. G. et al. Optimal trajectories of brain state transitions. NeuroImage (2017).