Open Access to what, exactly?

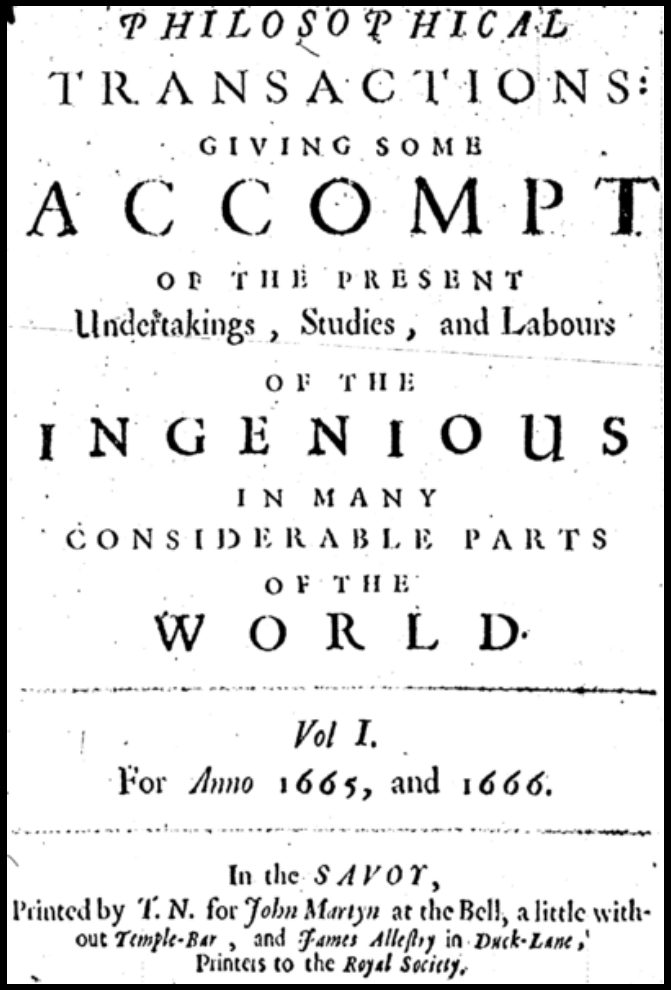
digital versions of traditional journal articles

(maybe updated with hyperlinks and DOIs and a few other elements)

Open Access to traditional journal articles would be a tremendous achievement

fall far short of what is possible

opportunity isn't just to digitize paper



PHILOSOPHICAL
TRANSACTIONS:
GIVING SOME
ACCOMPT
OF THE PRESENT
Undertakings, Studies, and Labours
OF THE
INGENIOUS
IN MANY
CONSIDERABLE PARTS
OF THE
WORLD.

Vol I.
For *Anno* 1665, and 1666.

In the *SAVOY*,
Printed by *T. N.* for *John Martyn* at the Bell, a little without *Temple-Bar*, and *James Allestry* in *Duck-Lane*,
Printers to the *Royal Society*.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

new media forms

radically different to

going far beyond

traditional journal articles

amplify our individual and collective intelligence

What would such
media forms look like?

ELSEVIER

SEARCH

Innovation in Publishing

The Article of the Future

The Article of the Future project is an ongoing initiative to revolutionize the traditional format of the academic paper in regard to three key elements: presentation, content and context.

By IJsbrand Jan Aalbersberg Posted on 21 September 2012

Print PDF

Share story:

Feedback

look further afield

ELSEVIER

SEARCH

Innovation in Publishing

The Article of the Future

The Article of the Future project is an ongoing initiative to revolutionize the traditional format of the academic paper in regard to three key elements: presentation, content and context.

By IJsbrand Jan Aalbersberg Posted on 21 September 2012

Print PDF

Share story:

Feedback

experiments that scientists are doing

ELSEVIER

SEARCH

Innovation in Publishing

The Article of the Future

The Article of the Future project is an ongoing initiative to revolutionize the traditional format of the academic paper in regard to three key elements: presentation, content and context.

By IJsbrand Jan Aalbersberg Posted on 21 September 2012

Print PDF

Share story:

Feedback

most creative designers in other fields

Some of the examples I show will likely be familiar

the point is to look at even familiar examples through an unusual lens, that of media theorist and designer

to identify powerful design patterns that lets us do things traditional media can't do

What I won't do:
look at a lot of the usual suspects: open data, online collaboration, reproducibility, open peer review, etc

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

example from Peter Norvig (Google)

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

interactive essay about economics

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

explore how the distribution of wealth changes over time

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

challenge preconceptions about economics

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

iPython notebook

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

essay format which mixes exposition with (live, editable) Python code

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

you don't need to understand details of Python or economics!

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

point is the media form

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

probably many of you have seen iPython notebooks before

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

often touted for reproducibility or open code

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

focus on something different:

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

iPython notebooks as a powerful new media form

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

enables the reader to easily perform formerly difficult cognitive tasks

Economics Simulation

This is a simulation of an economic marketplace in which there is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (according to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the results shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt
```

taking a close up look at the essay

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
 Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)
```

code to execute transactions between two members of the population

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
 Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)
```

Norvig considers several possibilities

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

deciding which members of the population have transactions

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sveri Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

exposition

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

code

```
In [299]: import random
          import matplotlib
          import matplotlib.pyplot as plt

          N  = 5000 # Default size of population
          mu = 100. # Default mean of population's wealth

          def sample(distribution, N=N, mu=mu):
              "Sample from the distribution N times, then normalize results to hav
          e mean mu."
              return normalize([distribution() for _ in range(N)], mu * N)

          def constant(mu=mu):           return mu
          def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
          /2)
          def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
          def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
          def pareto(alpha=4):          return random.paretovariate(alpha)

          def normalize(numbers, total):
              "Scale the numbers so that they add up to total."
              factor = total / float(sum(numbers))
              return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y . For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

exposition

code

```
In [360]: def random_split(X, Y):
              "Take all the money in the pot and divide it randomly between X and
          Y."
              pot = X + Y
              m = random.uniform(0, pot)
              return m, pot - m

          def winner_take_most(X, Y, most=3/4.):
              "Give most of the money in the pot to one of the parties."
              pot = X + Y
              m = random.choice((most * pot, (1 - most) * pot))
              return m, pot - m

          def winner_take_all(X, Y):
              "Give all the money in the pot to one of the actors."
              return winner_take_most(X, Y, 1.0)

          def redistribute(X, Y):
              "Give 55% of the pot to the winner; 45% to the loser."
              return winner_take_most(X, Y, 0.55)

          def split_half_min(X, Y):
              """The poorer actor only wants to risk half his wealth;
              the other actor matches this; then we randomly split the pot."""
              pot = min(X, Y)
              m = random.uniform(0, pot)
              return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently , but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

exposition

code

```
In [356]: def anyone(pop): return random.sample(range(len(pop)), 2)

          def nearby(pop, k=5):
              i = random.randrange(len(pop))
              j = i + random.choice((1, -1)) * random.randint(1, k)
              return i, (j % len(pop))

          def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the r esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that occupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10% , 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

exposition

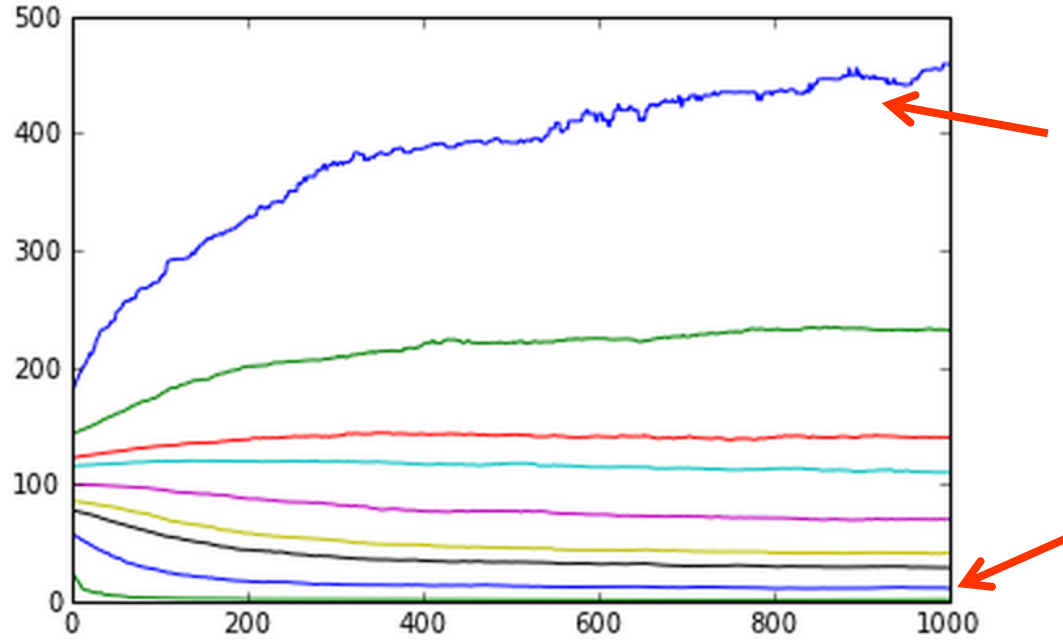
```
In [368]: def simulate(population, transaction_fn, interaction_fn, T, percentiles,
           record_every):
              "Run simulation for T steps; collect percentiles every 'record_every
          ' time steps."
              results = []
              for t in range(T):
```

once Norvig gets the code running, he performs simulations, running transactions over and over again, looking at the long-term distribution of wealth

In [373]:
```
report(gauss, random_split)
report(gauss, winner_take_most)
report(gauss, winner_take_all)
```

```
Simulation: 5000 * gauss(mu=100.0) for T=25000 steps with anyone doing
random_split:

           top 1%    top 10%    top 25%  top 33.3%     median  bot 33.3%
    bot 25%    bot 10%     bot 1%
start      179.15      142.41      122.16      115.14       99.60       85.68
       77.36       57.15       22.66
mid        391.59      220.50      142.05      117.68       76.85       45.21
       33.11       12.83        1.10
final      458.68      231.19      139.85      109.79       69.74       41.12
       28.40       10.73        1.15
```



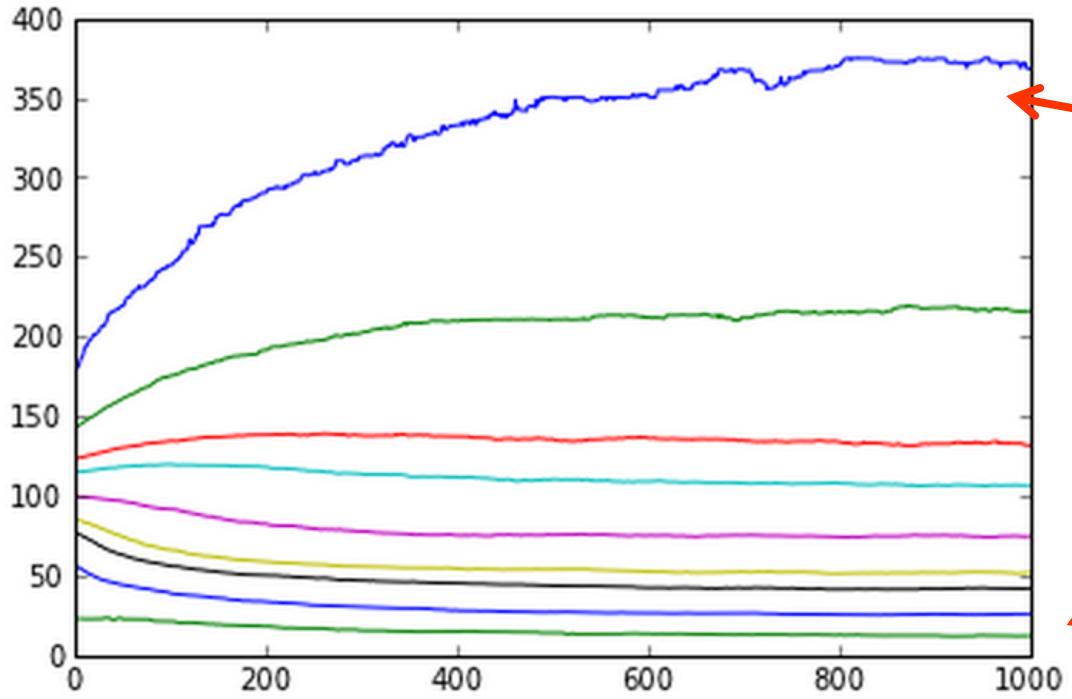
Richest 1%

Poorest 1%

(-> 1 unit)

Simulation: 5000 * gauss(mu=100.0) for T=25000 steps with anyone doing winner_take_most:

	top 1%	top 10%	top 25%	top 33.3%	median	bot 33.3%	bot 25%	bot 10%	bot 1%
start	178.63	142.45	123.04	114.70	99.82	85.52	77.20	56.57	22.87
mid	350.29	210.58	135.68	110.43	75.48	53.93	43.55	26.96	14.01
final	368.07	216.06	131.68	106.61	74.65	51.83	41.93	25.97	12.12



370 units

(down from 460)

12 units

(up from 1)

not going to get into
why this happens

instead: what makes iPython notebooks interesting as a media form?

imagine we used a traditional journal format for this set of results

it'd contain an abstract mathematical model, and describe the results

a reader would need a sophisticated knowledge of mathematics to figure out the consequences of even tiny changes to the model

Norvig's essay doesn't just contain an abstract version of the model

It contains an executable, live version of the model

Anyone who knows a tiny bit of programming can edit the code, and actually change the model

You could, for example, change the split and see how this affects the final distribution of wealth

This can be done live, simply by editing the notebook

And you can also change the assumptions in other ways

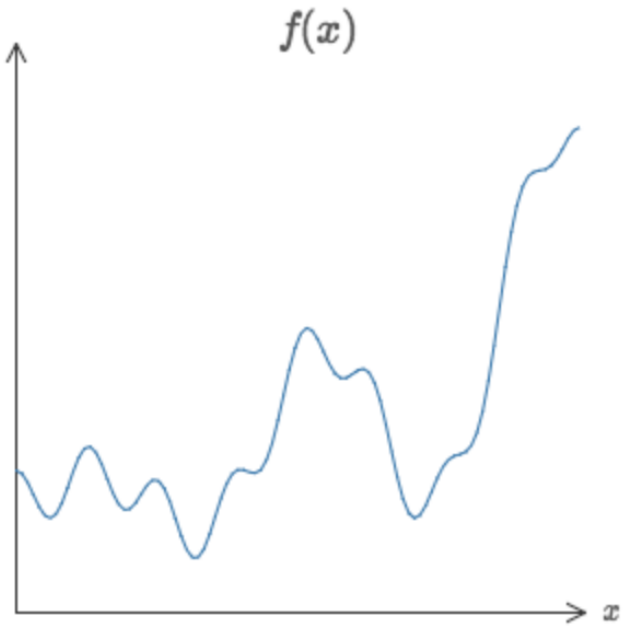
notebook format dramatically
expands the range of people
who can understand and compare
these different models

makes this knowledge both more explorable and more extensible

CHAPTER 4

A visual proof that neural nets can compute any function

One of the most striking facts about neural networks is that they can compute any function at all. That is, suppose someone hands you some complicated, wiggly function, $f(x)$:

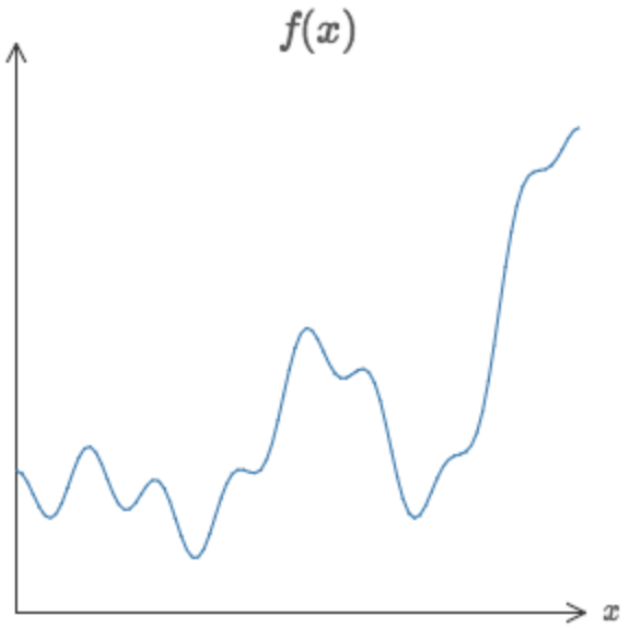


interactive visual essay

CHAPTER 4

A visual proof that neural nets can compute any function

One of the most striking facts about neural networks is that they can compute any function at all. That is, suppose someone hands you some complicated, wiggly function, $f(x)$:



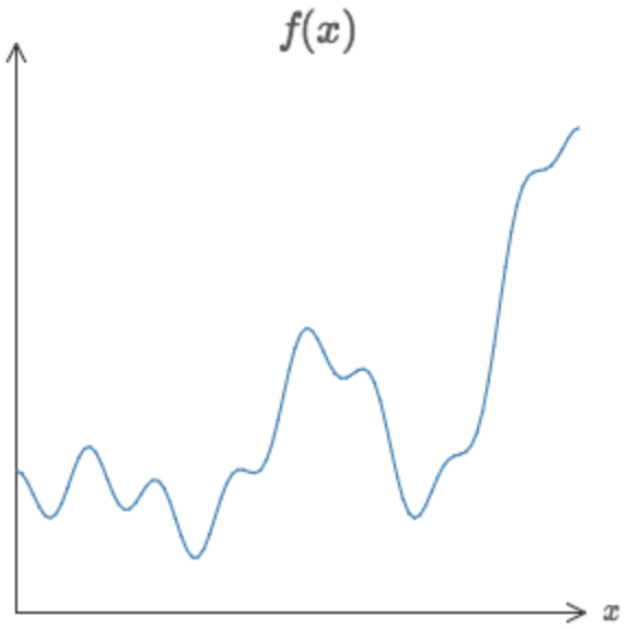
Sponsors

part of a book on neural networks & deep learning

CHAPTER 4

A visual proof that neural nets can compute any function

One of the most striking facts about neural networks is that they can compute any function at all. That is, suppose someone hands you some complicated, wiggly function, $f(x)$:



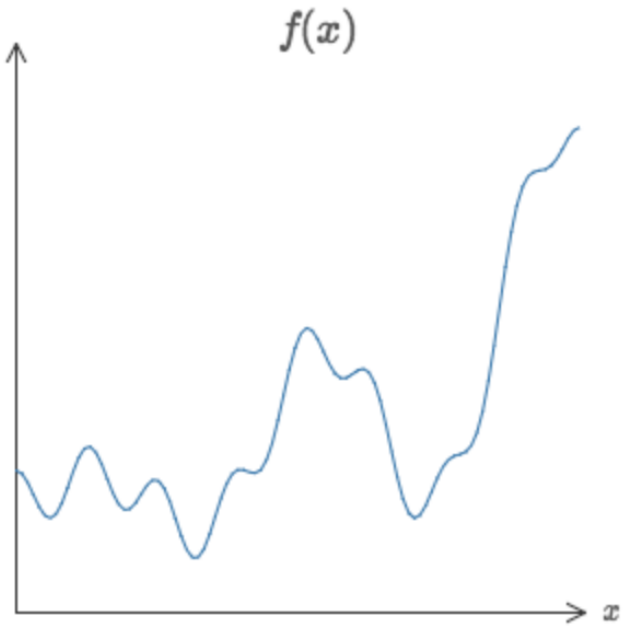
Sponsors

don't worry if you don't know anything about neural networks

CHAPTER 4

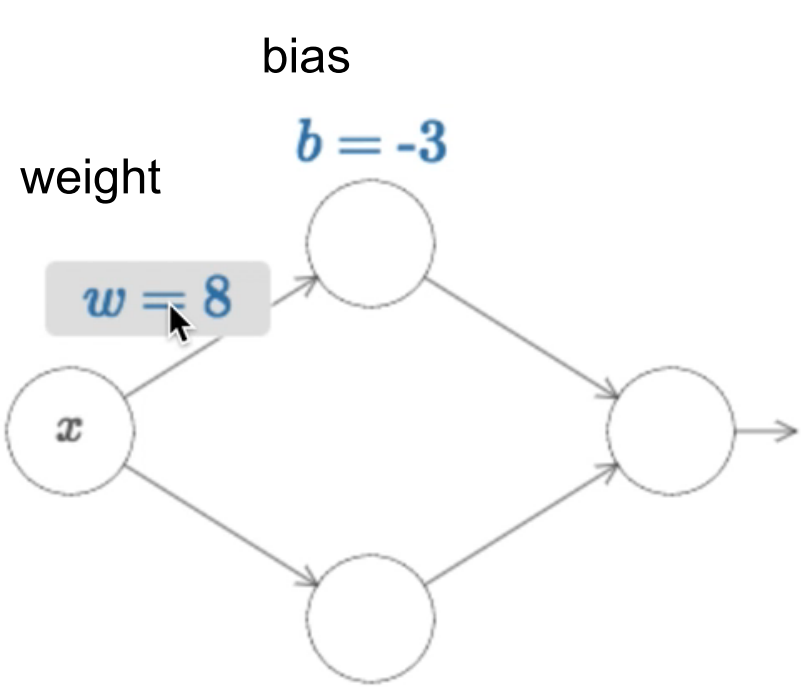
A visual proof that neural nets can compute any function

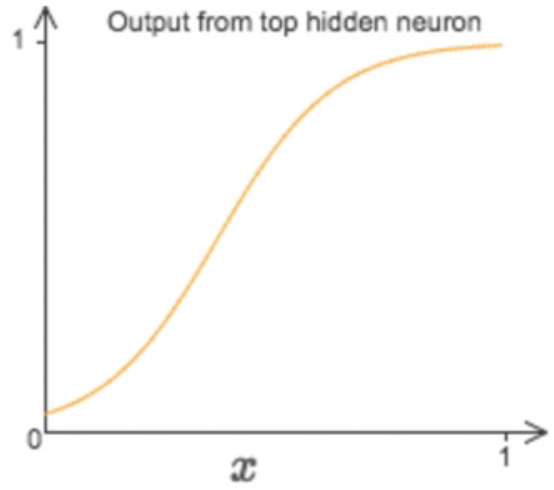
One of the most striking facts about neural networks is that they can compute any function at all. That is, suppose someone hands you some complicated, wiggly function, $f(x)$:



examine this as a medium

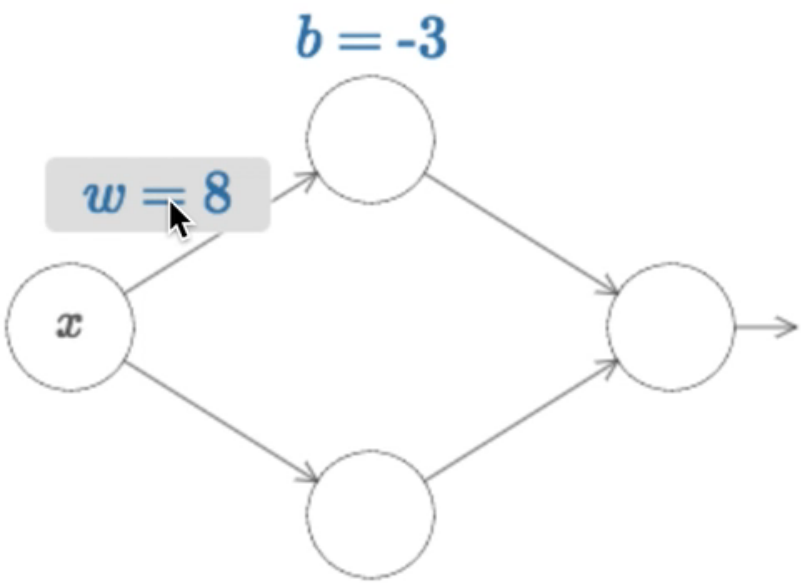
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

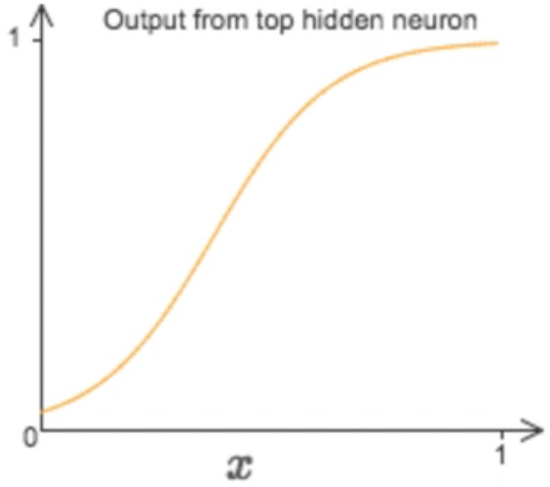




simple artificial neural network (on left)

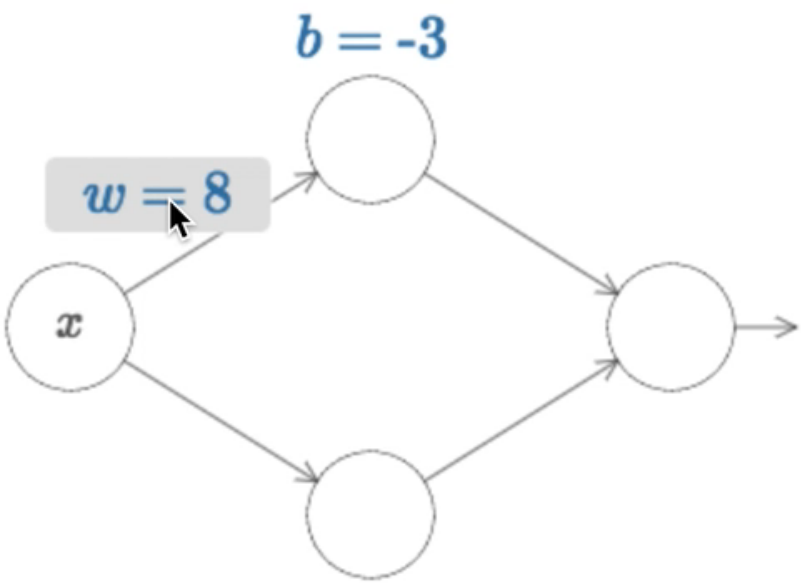
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

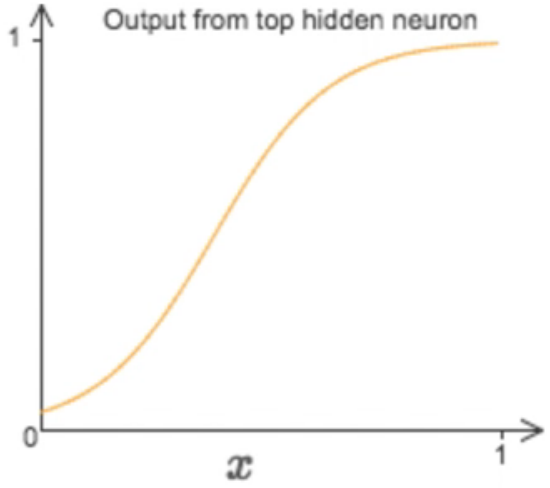




ask reader to experiment with changing weights and biases

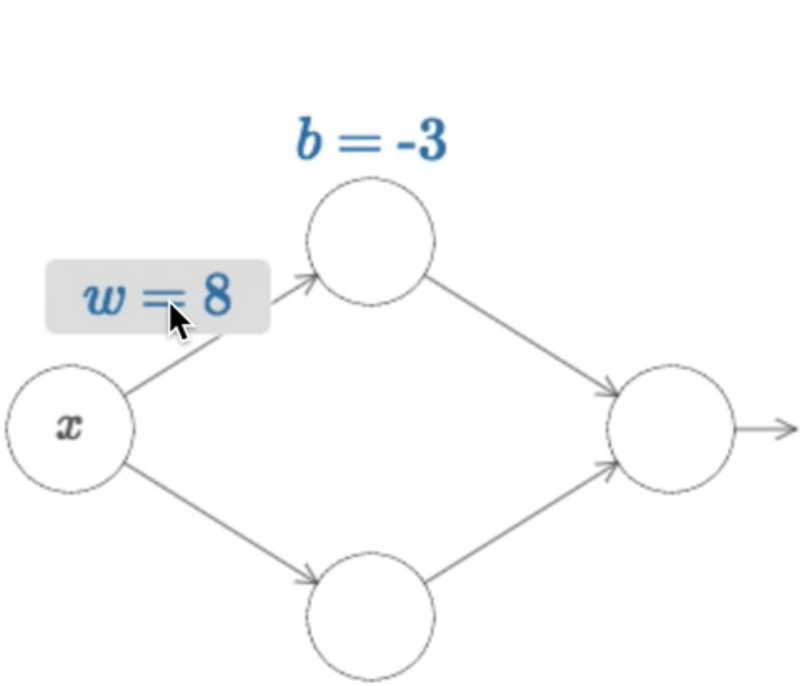
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

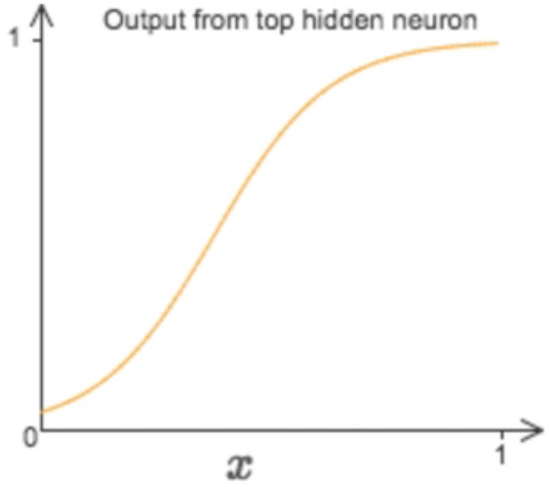




take the reader through several exercises

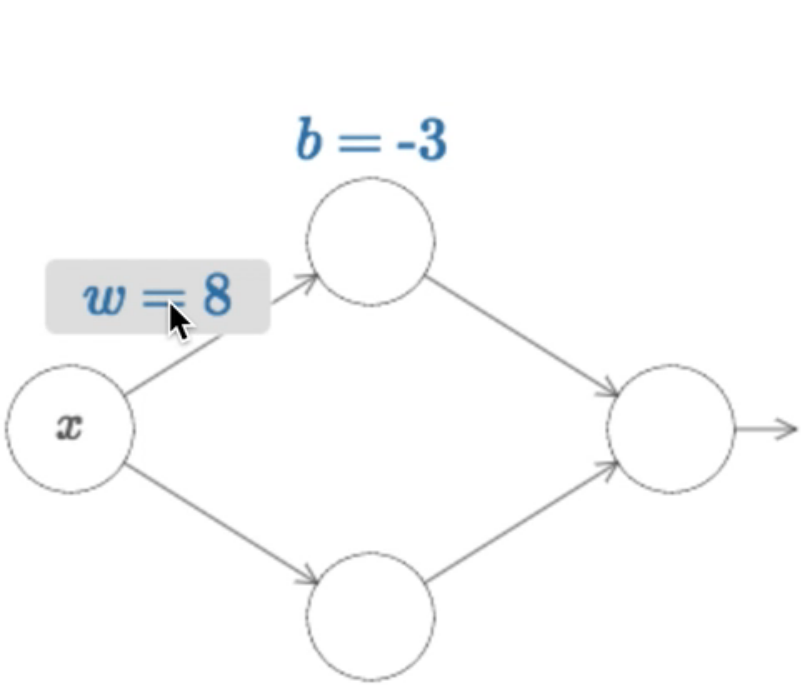
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

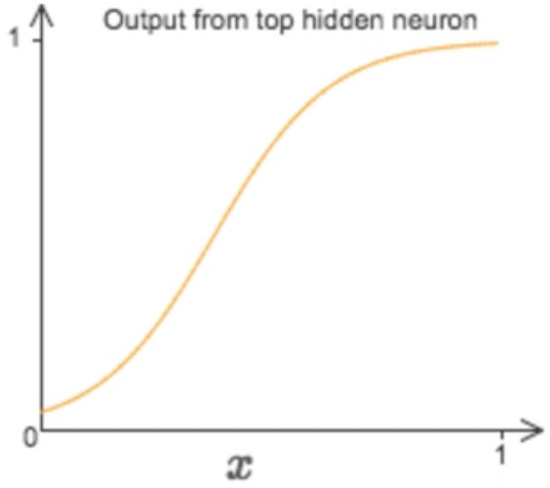




asking them to vary weights and biases
to get desired output signal

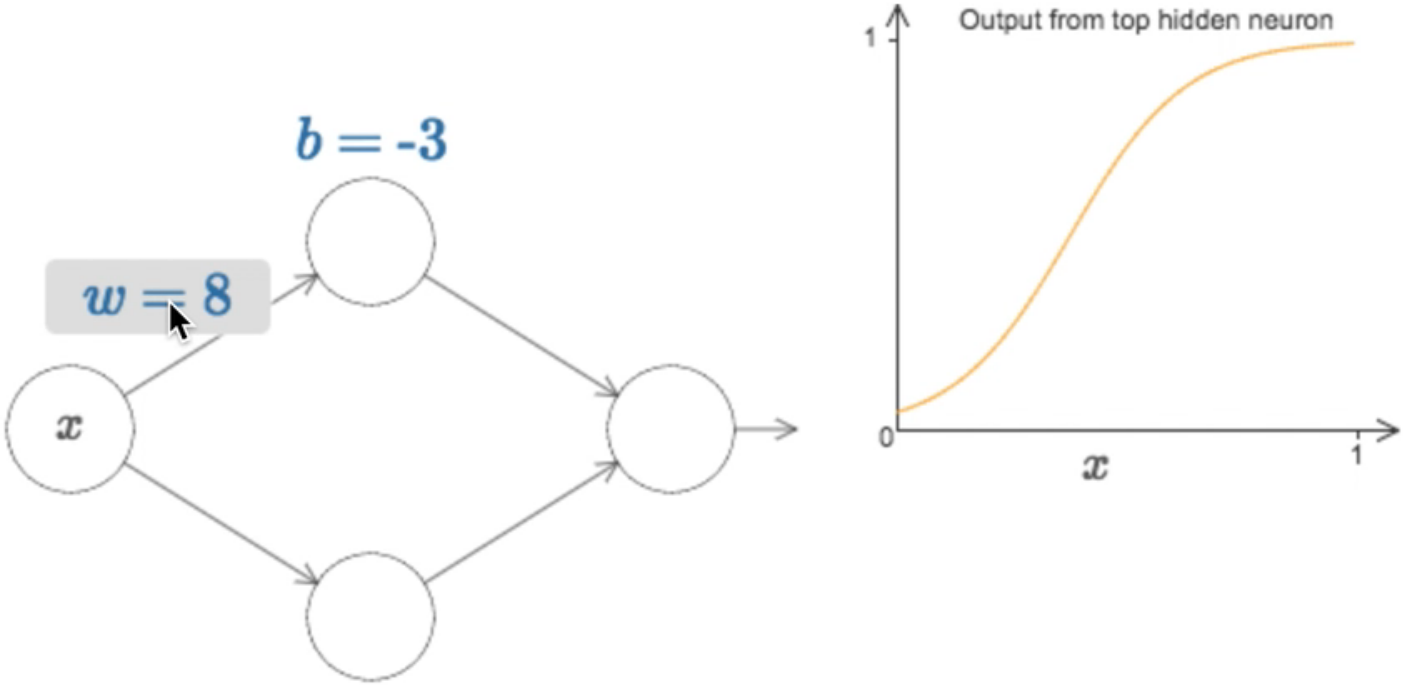
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:





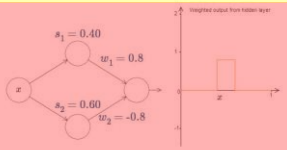
won't go through details

To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

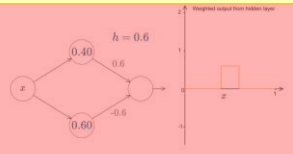


by going through this, readers can build up intuition for what's going on in the network

Finally, try setting w_1 to be 0.8 and w_2 to be −0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height. To reduce clutter I'll also remove the "$s_1 = \ldots$" and "$w_1 = \ldots$" notations.



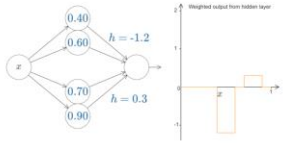
Try changing the value of h up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

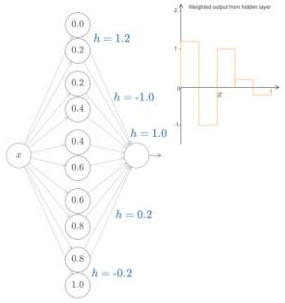
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the h values for each pair of hidden neurons. Try increasing and decreasing both h values, and observe how it changes the graph. Move the bumps around by changing the step points.

More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval [0, 1] up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



exposition

neural network

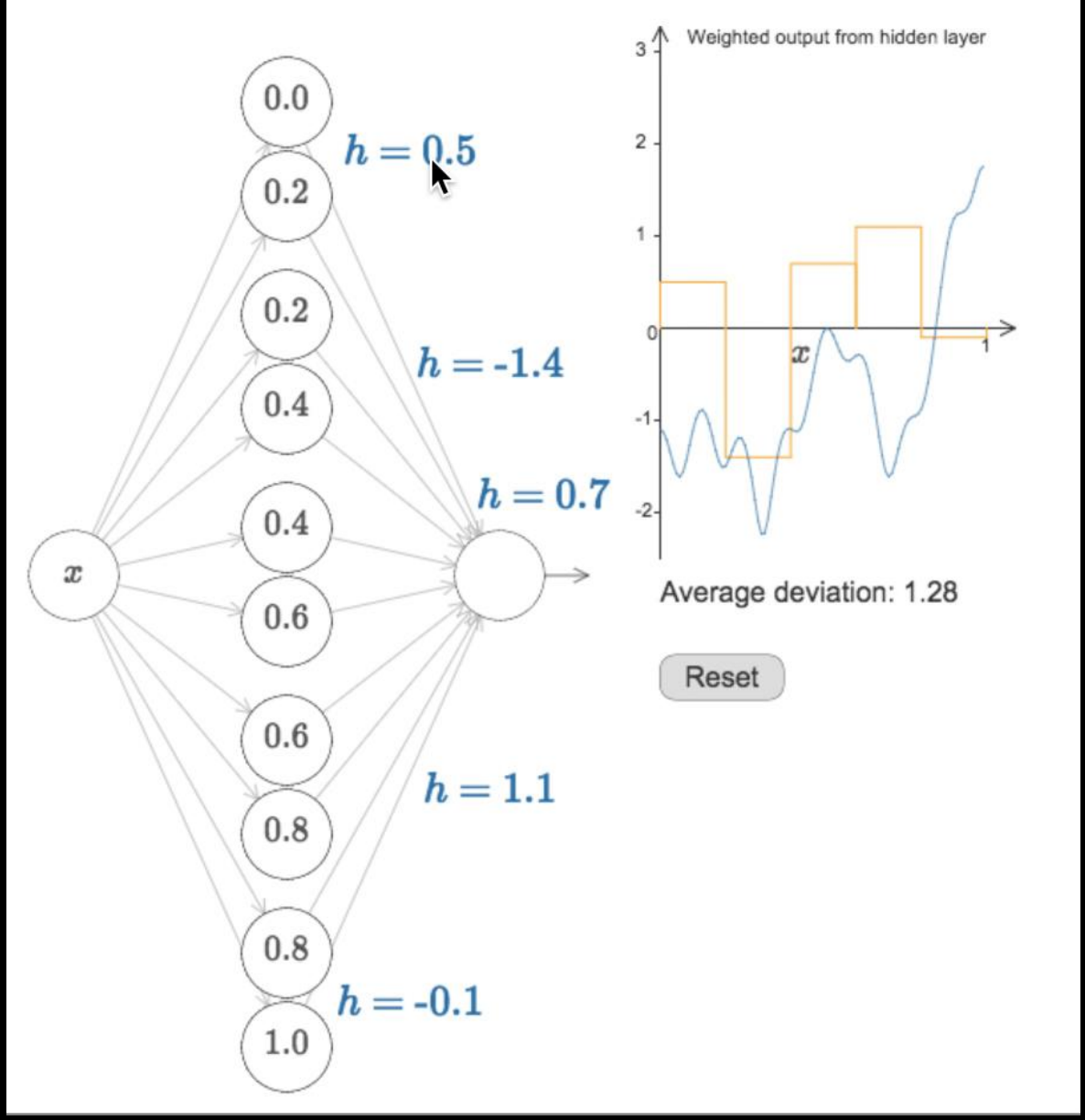
exposition

slightly more complex neural network

and so on

gradually learn more and more complex things we can do with a neural network

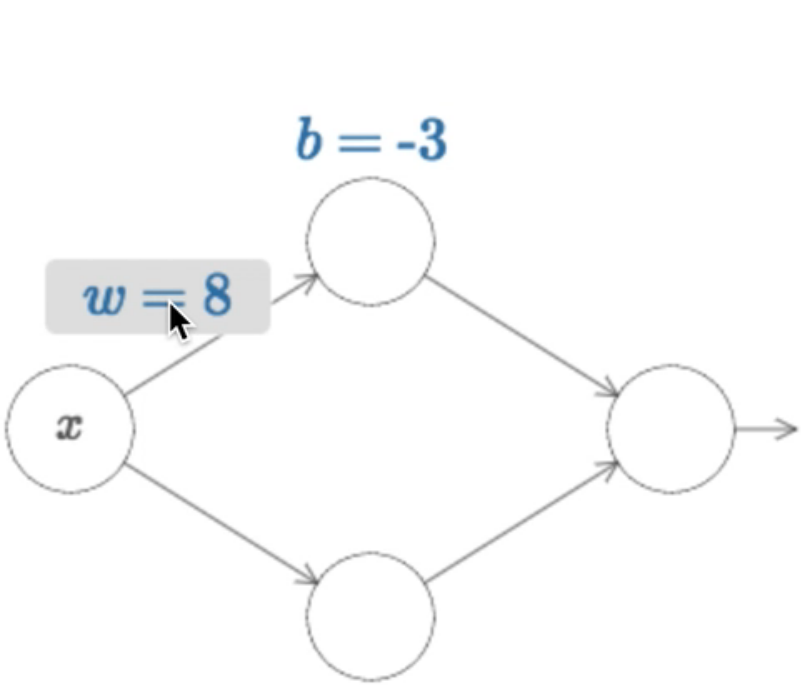
essay proceeds back and forth

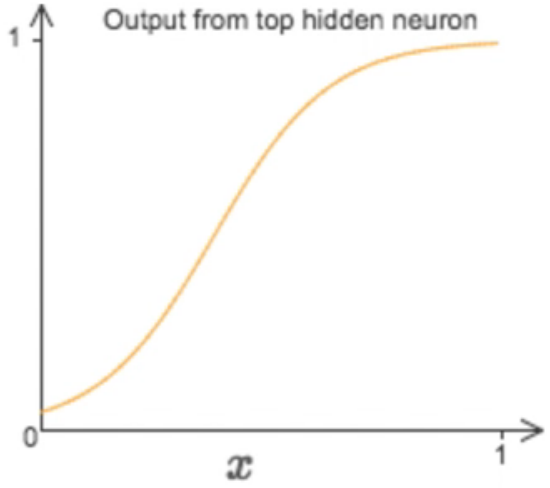


eventually, I ask readers to change parameters of network to compute some desired function

abstracting away, there's a dialogue back and forth between discussions of an abstract mathematical model, and concrete, explorable versions of the model, which the reader can play with to achieve some goal

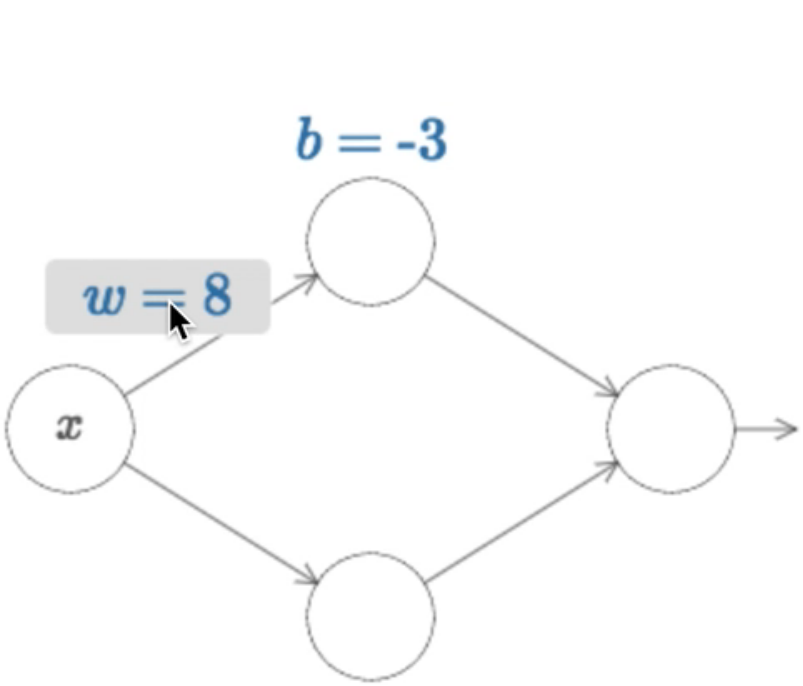
To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

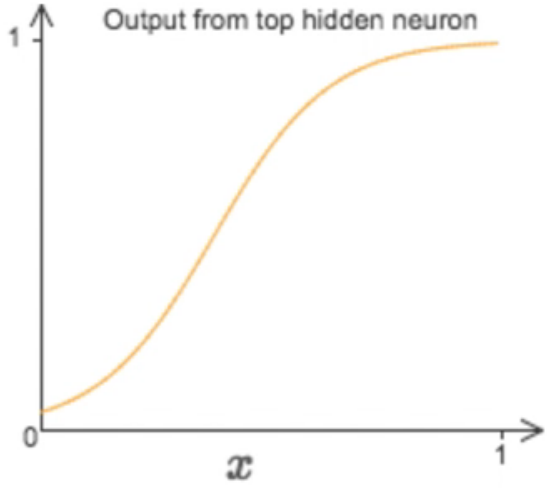




early models are very simple

To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, w, and drag the mouse a little ways to the right to increase w. You can immediately see how the function computed by the top hidden neuron changes:

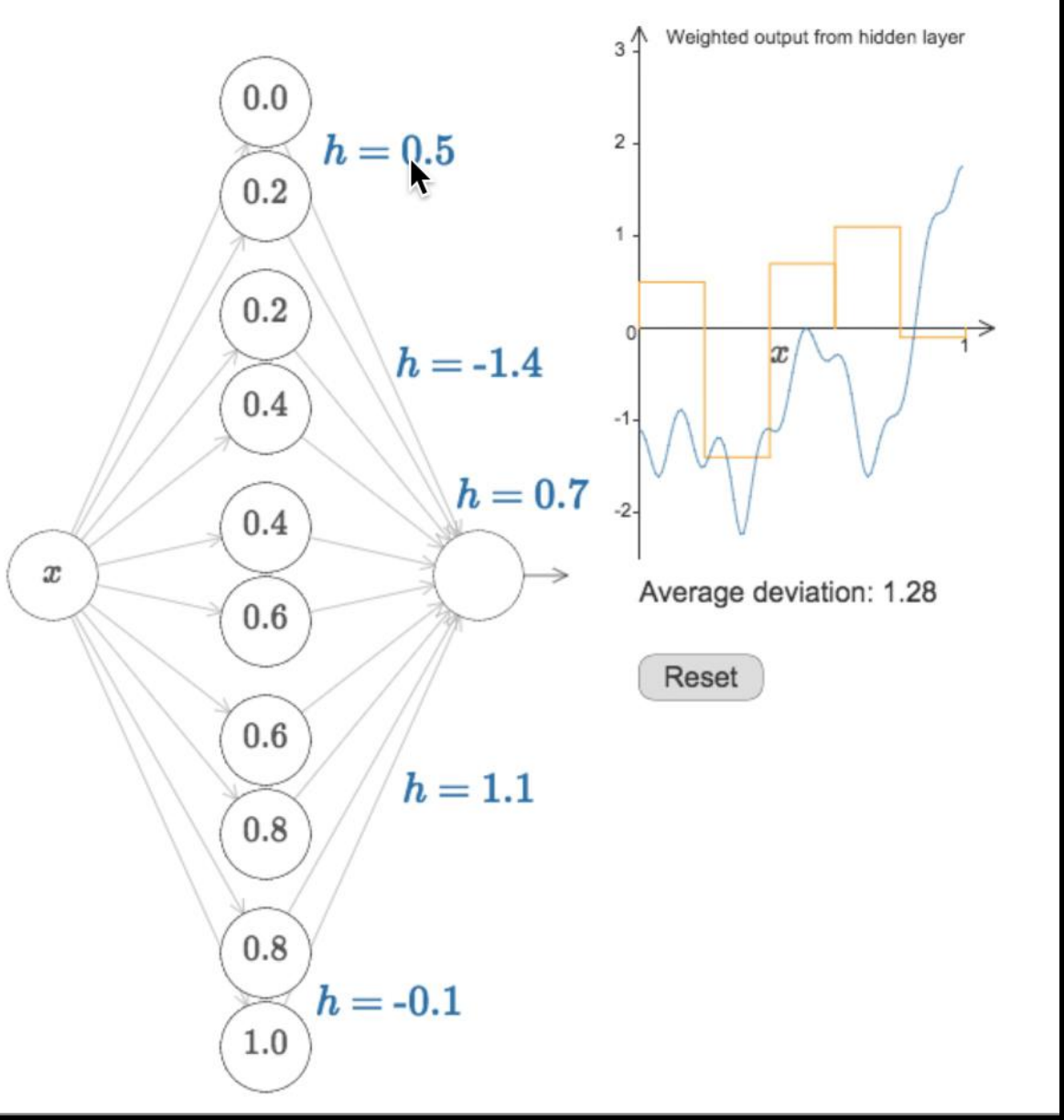




little the reader can do

as the reader builds up knowledge,
the models get more complex,
and the readers get more options
to explore

those options add complexity and make the reader's life more difficult


0.0
0.2
0.2
0.4
0.4
0.6
0.6
0.8
0.8
1.0
x
h = 0.5
h = -1.4
h = 0.7
h = 1.1
h = -0.1
Weighted output from hidden layer
3
2
1
0
-1
-2
x
1
Average deviation: 1.28
Reset

but the reader's understanding has grown, and so they are able to deal with the added complexity

media form:
building this dialogue between abstract discussion and concrete, explorable models

this media form looks
superficially similar to a
journal article,
but is actually essentially
different

also looks similar to the media form of Norvig's article: a dialogue between abstract discussion and code

actually quite different

nbviewer FAQ IPython Jupyter

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the *r* esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that occupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot where the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

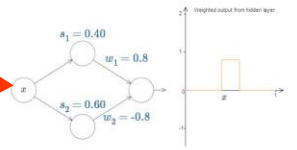
each model is complete in itself

each code snippet is merely part of an overall model

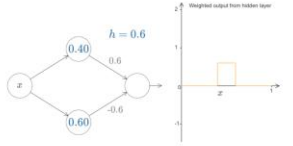
build up to complexity through a series of models, each complete in itself

build up to complexity by looking closely at all the constituent parts

Finally, try setting w_1 to be 0.8 and w_2 to be –0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height. To reduce clutter I'll also remove the "$s_1 = \ldots$" and "$w_1 = \ldots$" notations.



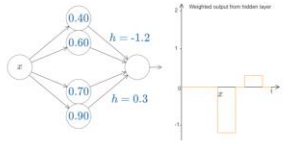
Try changing the value of h up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

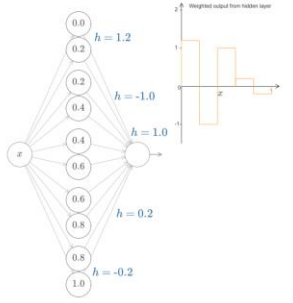
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the h values for each pair of hidden neurons. Try increasing and decreasing both h values, and observe how it changes the graph. Move the bumps around by changing the step points.

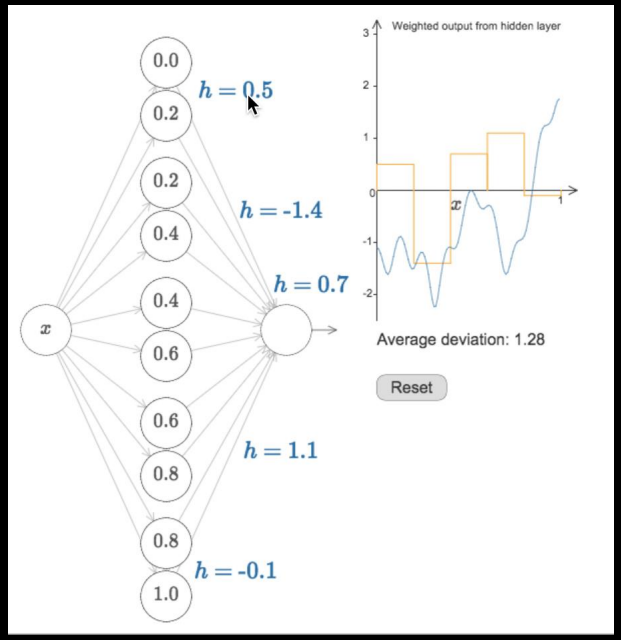
More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval $[0, 1]$ up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



the media forms do different things

both are valuable

neither can be done in a conventional journal format



Maybe the lesson is that we should be adding interactivity?

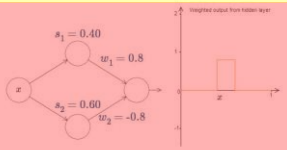
That analysis is too shallow

Interactivity by itself is irrelevant

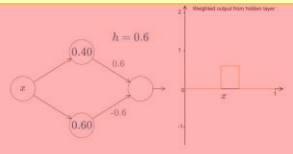
easy to create terrible
interactive explanations

what matters is the details of
the media form

Finally, try setting w_1 to be 0.8 and w_2 to be –0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height. To reduce clutter I'll also remove the "$s_1 =$ …" and "$w_1 =$ …" notations.



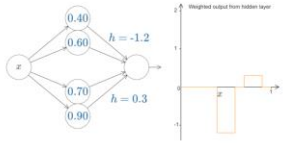
Try changing the value of h up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

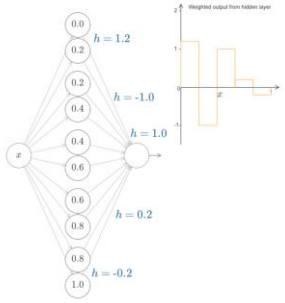
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the h values for each pair of hidden neurons. Try increasing and decreasing both h values, and observe how it changes the graph. Move the bumps around by changing the step points.

More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval [0, 1] up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



exposition

live model

exposition

more complex live model

…

dialogue back and forth

nbviewer FAQ IPython Jupyter

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the r esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that occupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot where the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

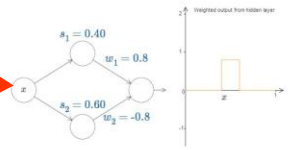
doesn't require that the reader be a programmer

need to know just a little mathematics

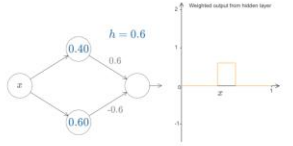
because it uses visual, concrete representations, rather than Python programs

OTOH the representations are fixed

Finally, try setting w_1 to be 0.8 and w_2 to be –0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height. To reduce clutter I'll also remove the "$s_1 = \ldots$" and "$w_1 = \ldots$" notations.



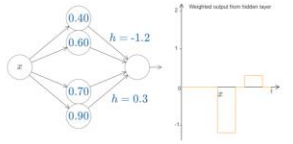
Try changing the value of h up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

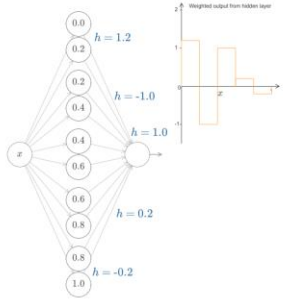
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the h values for each pair of hidden neurons. Try increasing and decreasing both h values, and observe how it changes the graph. Move the bumps around by changing the step points.

More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval $[0, 1]$ up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



nbviewer FAQ IPython Jupyter

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu):  return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3):  return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):     return random.betavariate(alpha, beta)
def pareto(alpha=4):           return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the *r* esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that reccupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

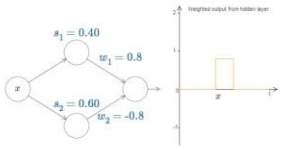
you are not limited by what Norvig has done

you can arbitrarily modify and extend the code

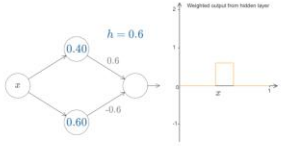
a platform for open-ended exploration

you could, for instance, introduce your own model of a transaction, & play with that, to develop insight

Finally, try setting w_1 to be 0.8 and w_2 to be –0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height. To reduce clutter I'll also remove the "$s_1 = \ldots$" and "$w_1 = \ldots$" notations.



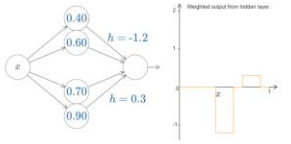
Try changing the value of h up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

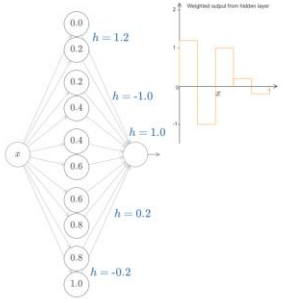
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the h values for each pair of hidden neurons. Try increasing and decreasing both h values, and observe how it changes the graph. Move the bumps around by changing the step points.

More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval $[0, 1]$ up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the r esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that reccupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

Norvig's notebook is something of a toy

imagine you developed it further in the same vein

introduce more complex models, explore the assumptions behind them, ...

eventually, your models would reach the edge of human thought on economics

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the *r* esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and *r* epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that reccupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

someone playing with the notebook could make modifications that would let them explore questions no-one had ever asked previously

and they might make new discoveries

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sveni Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the *r* esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that occupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

such a notebook would be a powerful cognitive medium

a medium for people to think and explore and make new discoveries in

for millenia, paper and pencil has been the most common cognitive medium

a sufficiently good digital notebook could reify in an active form much of the knowledge we have about the world

Economics Simulation

This is a simulation of an economic marketplace in which ther e is a *population* of actors, each of which has a level of wealth (a single number) that changes over time. On each time step two agents (chosen by an *interaction rule*) interact with each other and exchange wealth (accor ding to a *transaction rule*). The idea is to understand the evolution of the population's wealth over time. My hazy memory is that this idea came from a class by Prof. Sven Anderson at Bard (any errors or misconceptions here are due to my (Peter Norvig) misunderstanding of his idea). Why this is interesting: (1) an example of using simulation to model the world. (2) Many students will have preconceptions about how economies work that will be challenged by the r esults shown here.

Population Distributions

First things first: what should our initial population look like? We will provide several distribution functions (constant, uniform, Gaussian, etc.) and a `sample` function, which samples *N* elements form a distribution and then normalizes them to have a given mean. By default we will have *N*=5000 actors and an initial mean wealth of 100 simoleons.

In [299]:
```
import random
import matplotlib
import matplotlib.pyplot as plt

N  = 5000 # Default size of population
mu = 100. # Default mean of population's wealth

def sample(distribution, N=N, mu=mu):
    "Sample from the distribution N times, then normalize results to hav
e mean mu."
    return normalize([distribution() for _ in range(N)], mu * N)

def constant(mu=mu):           return mu
def uniform(mu=mu, width=mu): return random.uniform(mu-width/2, mu+width
/2)
def gauss(mu=mu, sigma=mu/3): return random.gauss(mu, sigma)
def beta(alpha=2, beta=3):    return random.betavariate(alpha, beta)
def pareto(alpha=4):          return random.paretovariate(alpha)

def normalize(numbers, total):
    "Scale the numbers so that they add up to total."
    factor = total / float(sum(numbers))
    return [x * factor for x in numbers]
```

Transactions

In a transaction, two actors come together; they have existing wealth levels X and Y. For now we will only consider transactions that conserve wealth, so our transaction rules will decide how to split up the pot of X+Y total wealth.

In [360]:
```
def random_split(X, Y):
    "Take all the money in the pot and divide it randomly between X and
Y."
    pot = X + Y
    m = random.uniform(0, pot)
    return m, pot - m

def winner_take_most(X, Y, most=3/4.):
    "Give most of the money in the pot to one of the parties."
    pot = X + Y
    m = random.choice((most * pot, (1 - most) * pot))
    return m, pot - m

def winner_take_all(X, Y):
    "Give all the money in the pot to one of the actors."
    return winner_take_most(X, Y, 1.0)

def redistribute(X, Y):
    "Give 55% of the pot to the winner; 45% to the loser."
    return winner_take_most(X, Y, 0.55)

def split_half_min(X, Y):
    """The poorer actor only wants to risk half his wealth;
    the other actor matches this; then we randomly split the pot."""
    pot = min(X, Y)
    m = random.uniform(0, pot)
    return X - pot/2. + m, Y + pot/2. - m
```

Interactions

How do you decide which parties interact with each other? The rule `anyone` samples two members of the population uniformly and independently, but there are other possible rules, like `nearby(pop, k)`, which choses one member uniformly and then chooses a second within `k` index elements away, to simulate interactions within a local neighborhood.

In [356]:
```
def anyone(pop): return random.sample(range(len(pop)), 2)

def nearby(pop, k=5):
    i = random.randrange(len(pop))
    j = i + random.choice((1, -1)) * random.randint(1, k)
    return i, (j % len(pop))

def nearby1(pop): return nearby(pop, 1)
```

Simulation

Now let's describe the code to run the simulation and summarize/plot the r esults. The function `simulate` does the work; it runs the interaction function to find two actors, then calls the transaction function to figure out how to split their wealth, and r epeats this T times. The only other thing it does is record results. Every so-many steps, it records some summary statistics of the population (by default, this will be every 25 steps).

What information do we record to summarize the population? Out of the N=5000 (by default) actors, we will record the wealth of exactly nine of them: the ones, in sorted-by-wealth or der that reccupy the 1% spot (that is, if N=5000, this would be the 50th wealthiest actor), then the 10%, 25% 1/3, and median; and then likewise fr om the bottom the 1%, 10%, 25% and 1/3.

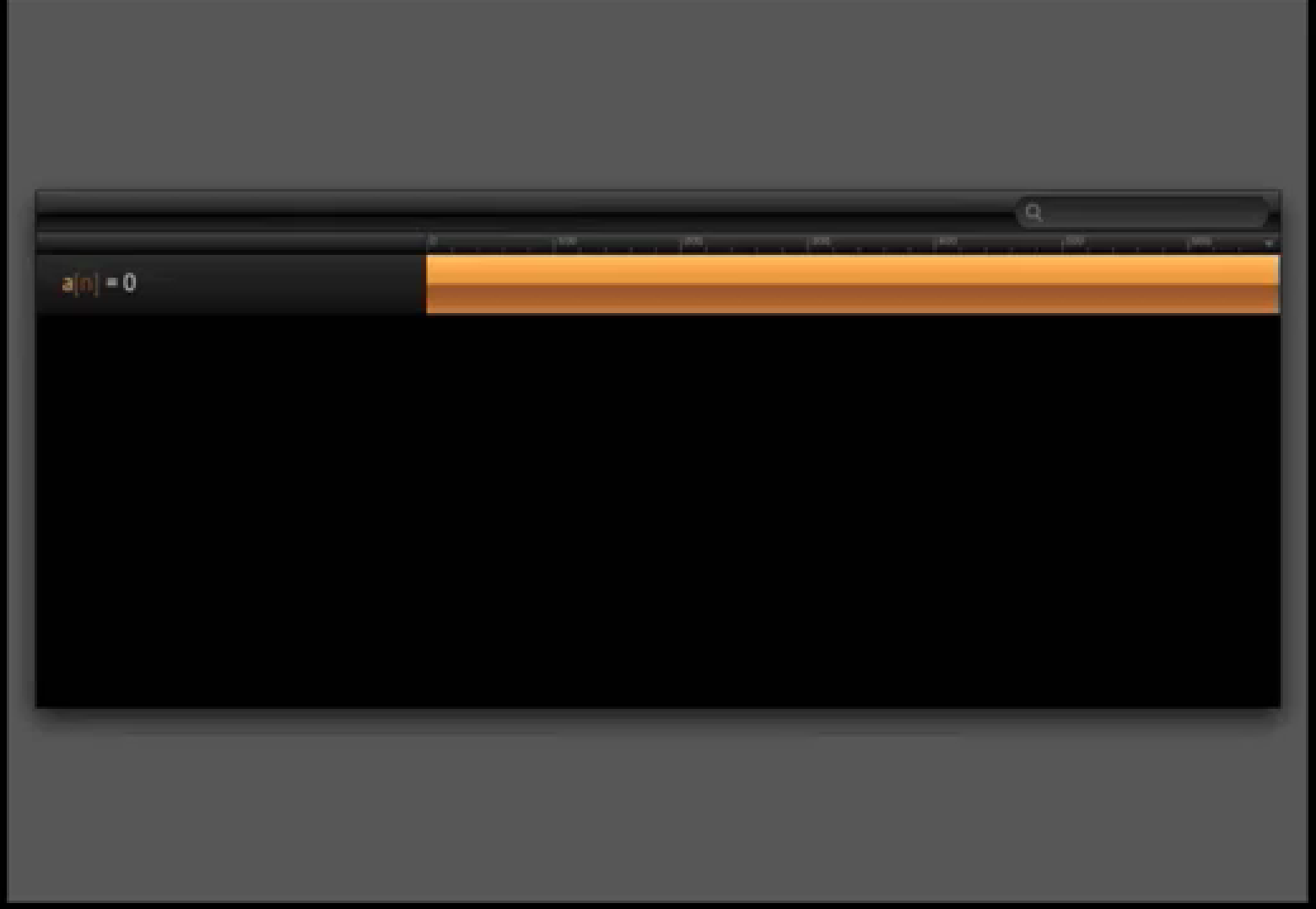
(Note that we record the *median*, which changes over time; the *mean* is defined to be 100 when we start, and since all transactions conserve wealth, the mean will always be 100.)

What do we do with these results, once we have recorded them? First we print them in a table for the first time step, the last, and the middle. Then we plot them as nine lines in a plot wher e the *y*-axis is wealth and the *x*-axis is time (note that when the *x*-axis goes fr om 0 to 1000, and we have `record_every=25`, that means we have actually done 25,000 transactions, not 1000).

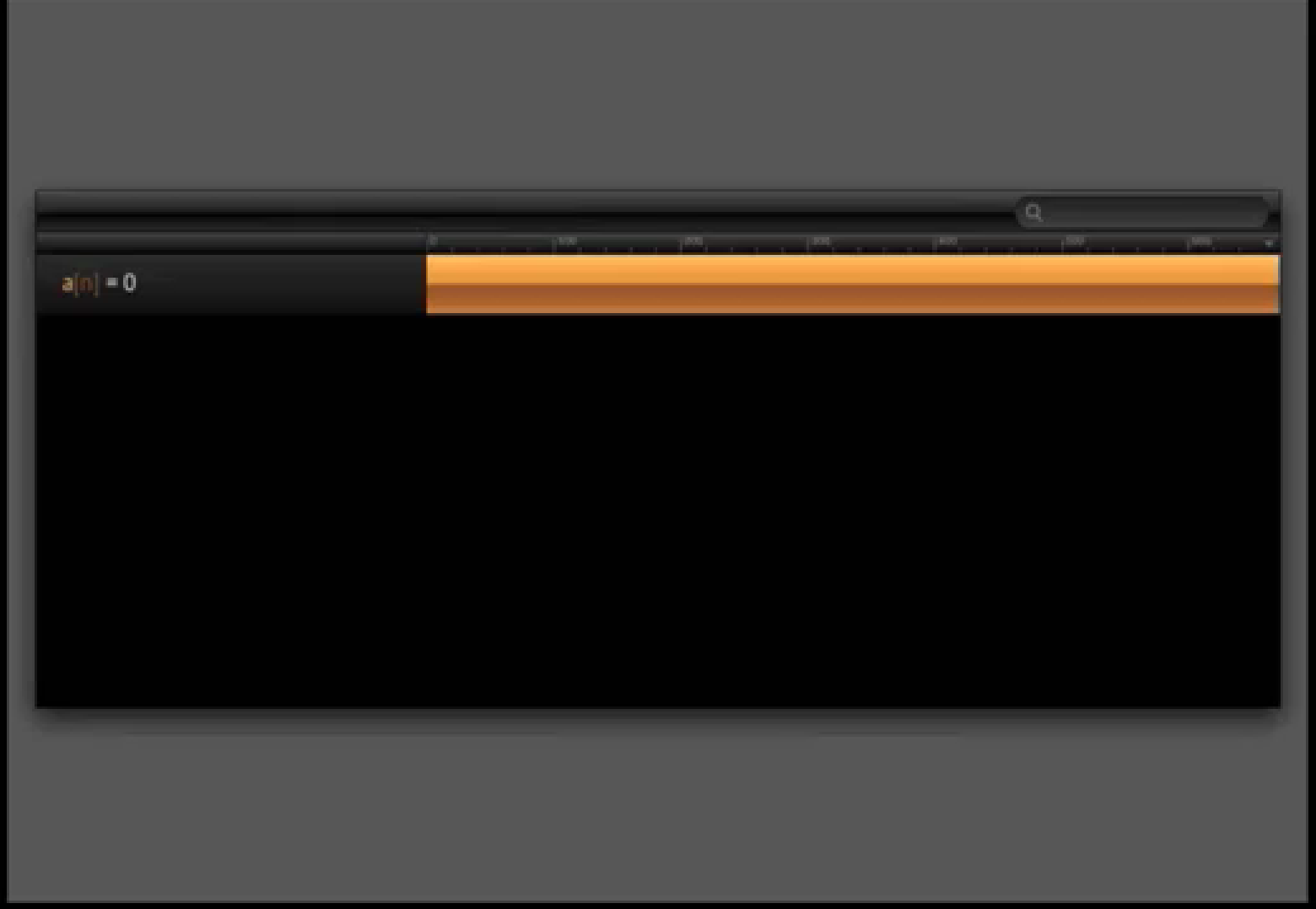
In [368]:
```
def simulate(population, transaction_fn, interaction_fn, T, percentiles,
 record_every):
    "Run simulation for T steps; collect percentiles every 'record_every
' time steps."
    results = []
    for t in range(T):
```

that knowledge would then be composable in a way that paper and pencil is not

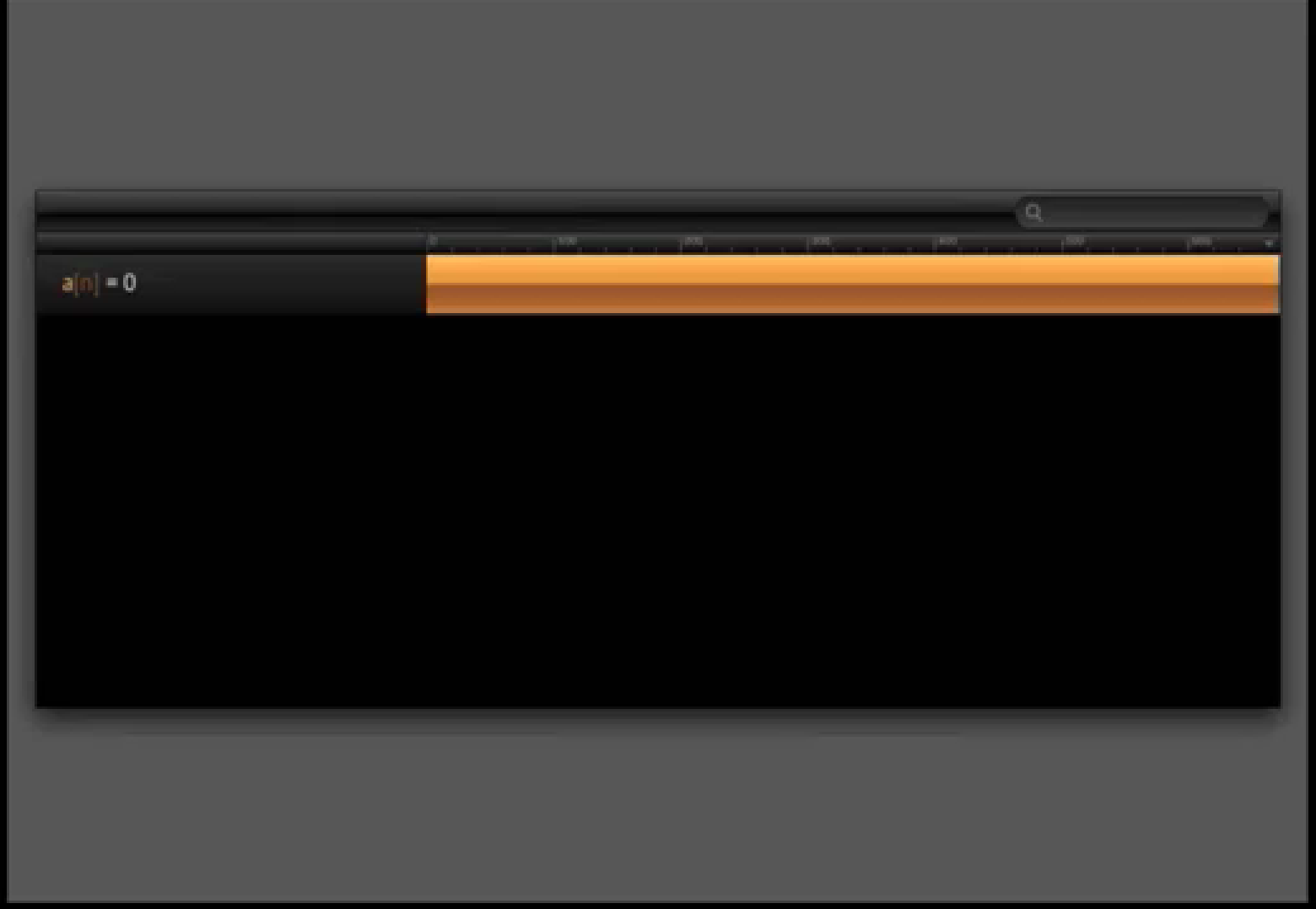
that would make it far, far easier for people to explore than in a conventional, static form



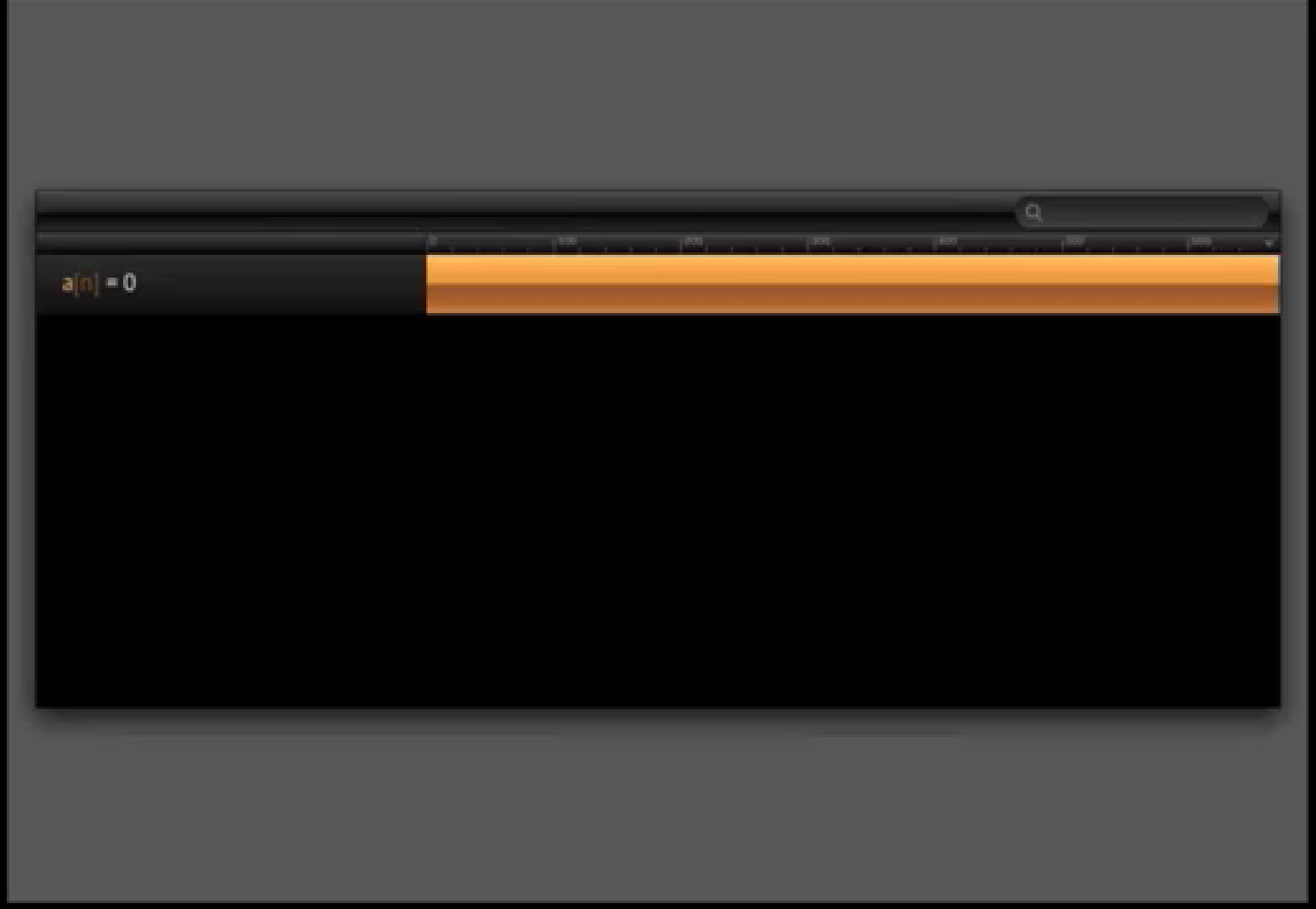
another cognitive medium,
from designer Bret Victor



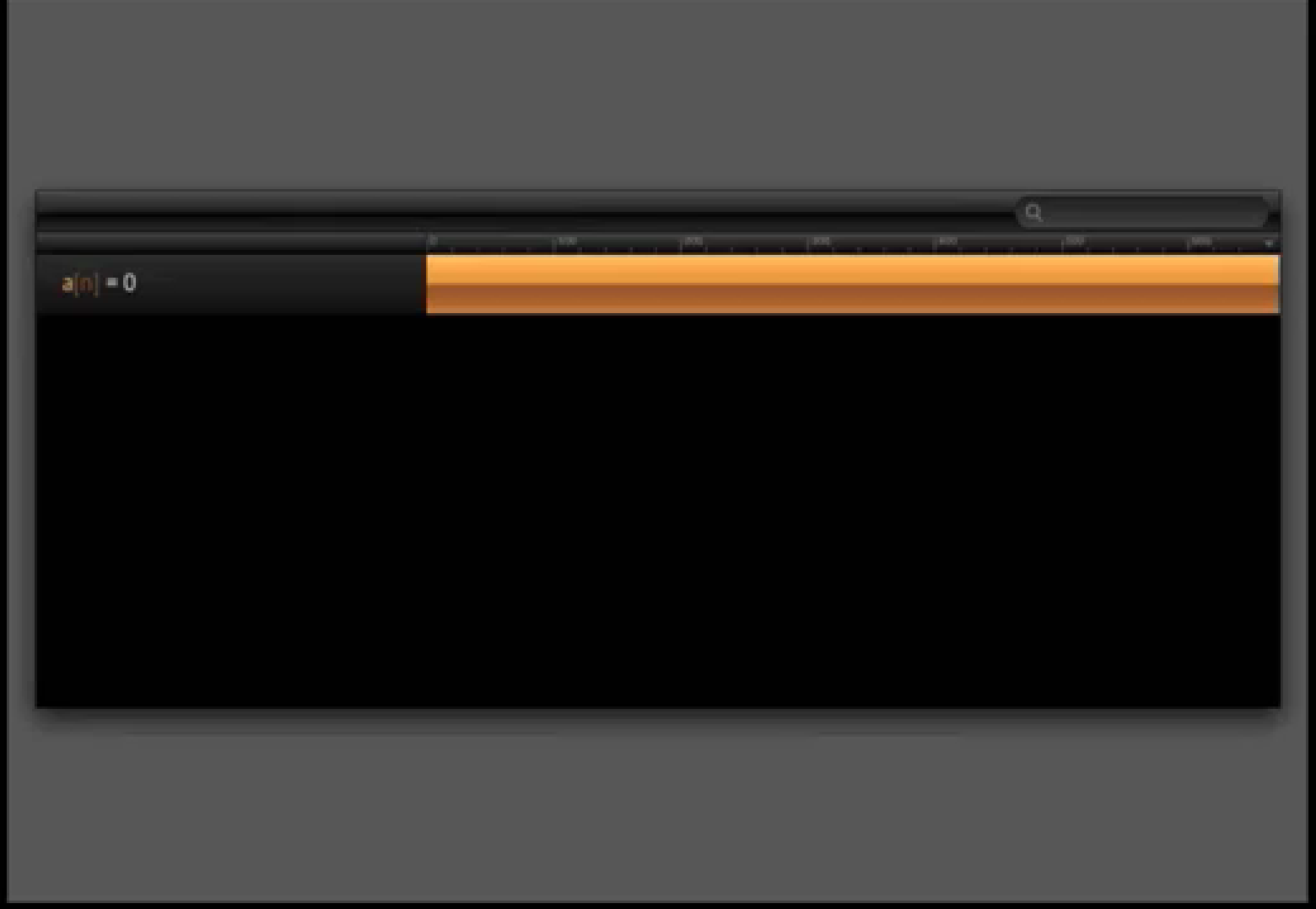
this isn't a journal article



no reason such a medium shouldn't be integrated with iPython notebook

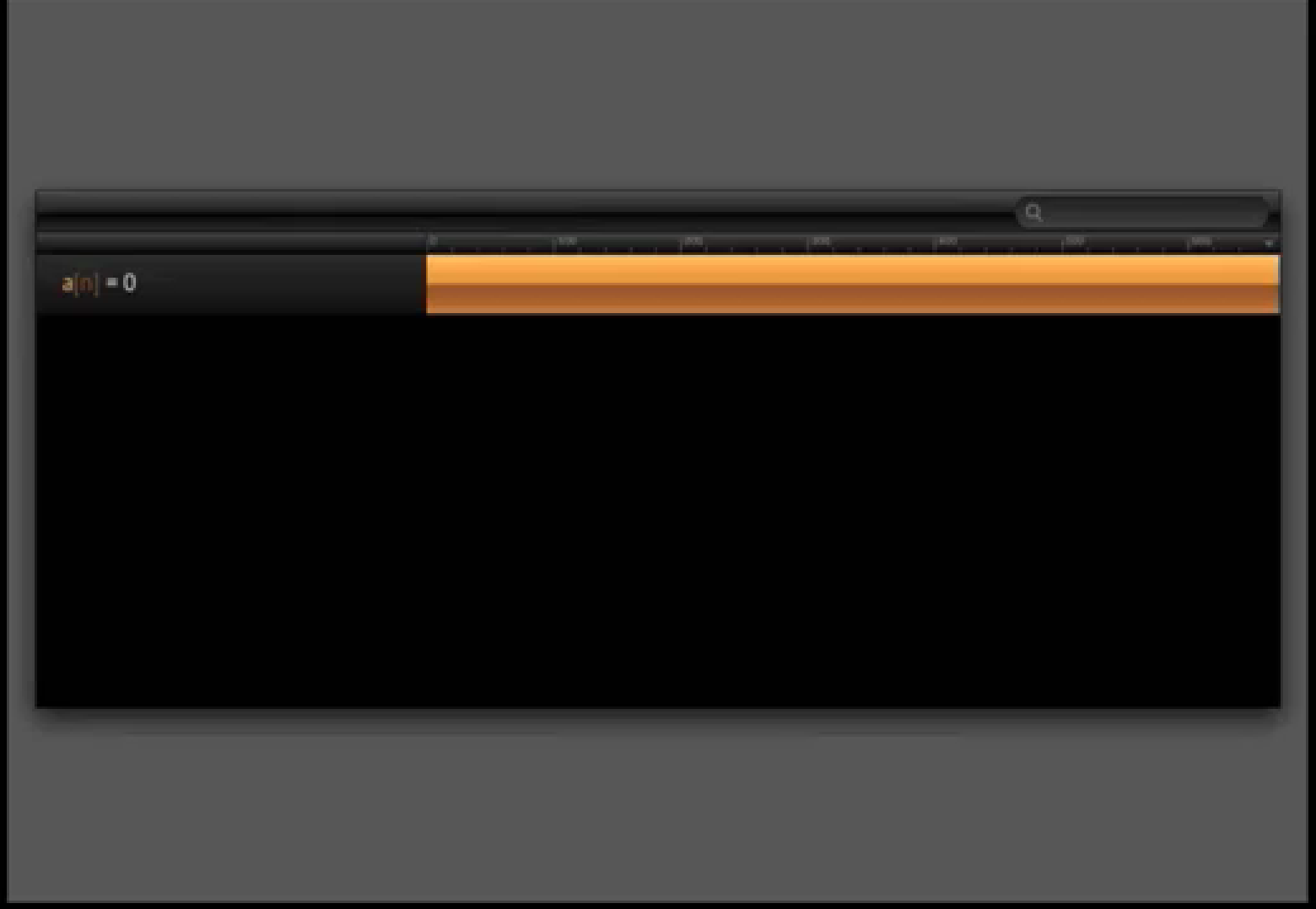


both are media for
exploration and creation

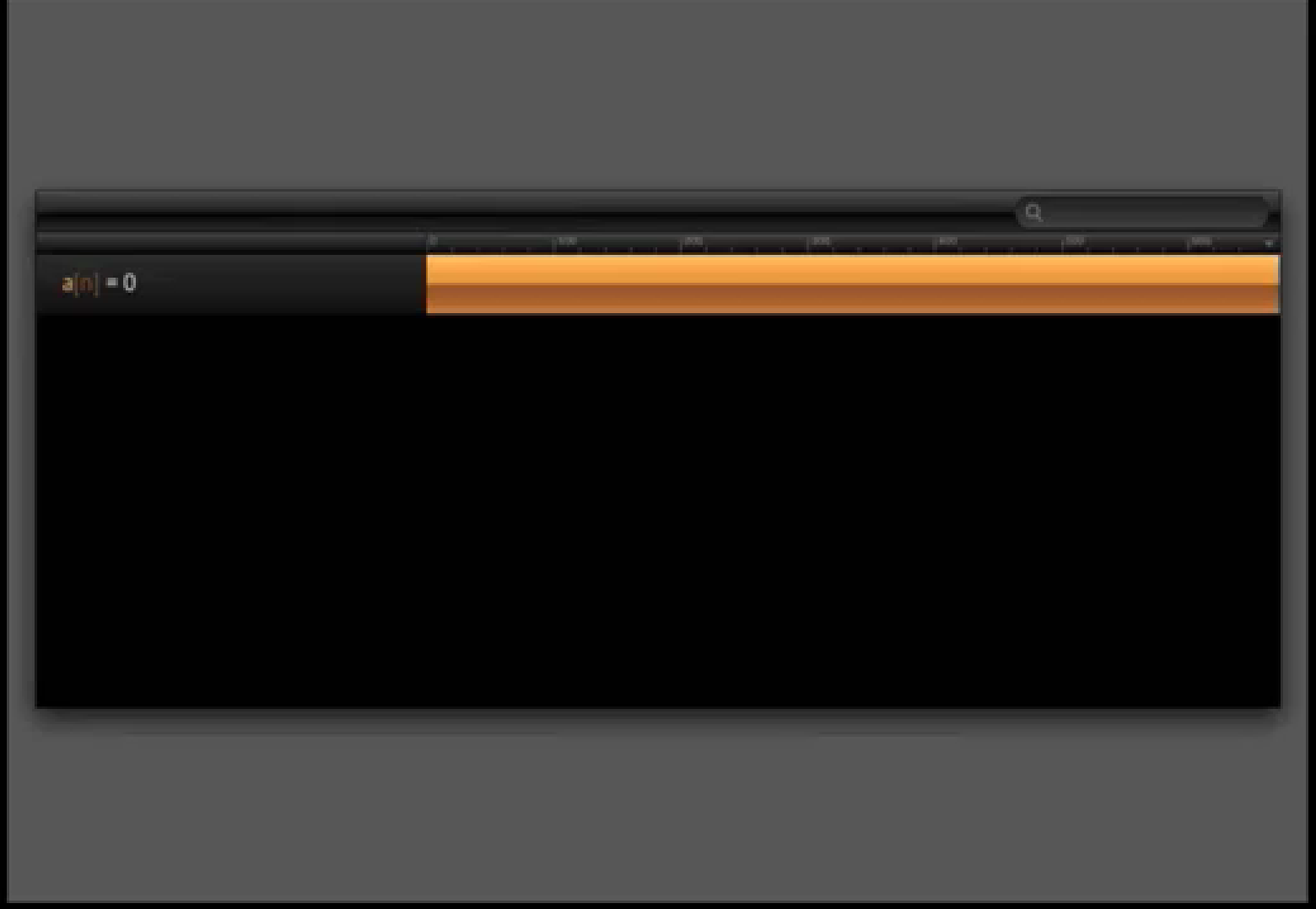


both are media for
exploration and creation

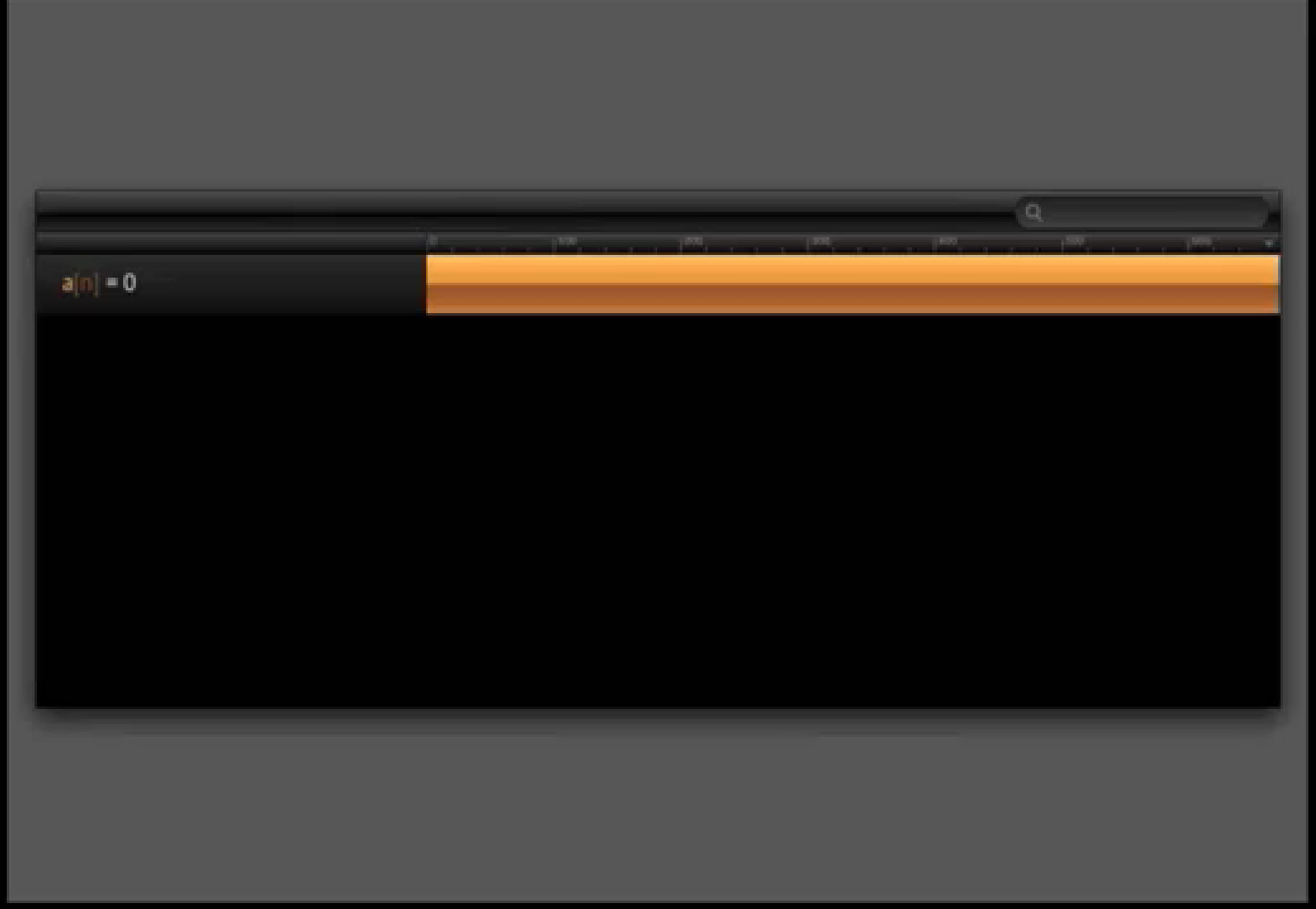
suggest a publishing platform which evolves toward being a cognitive medium, integrating all our best tools in an environment for exploration and discovery



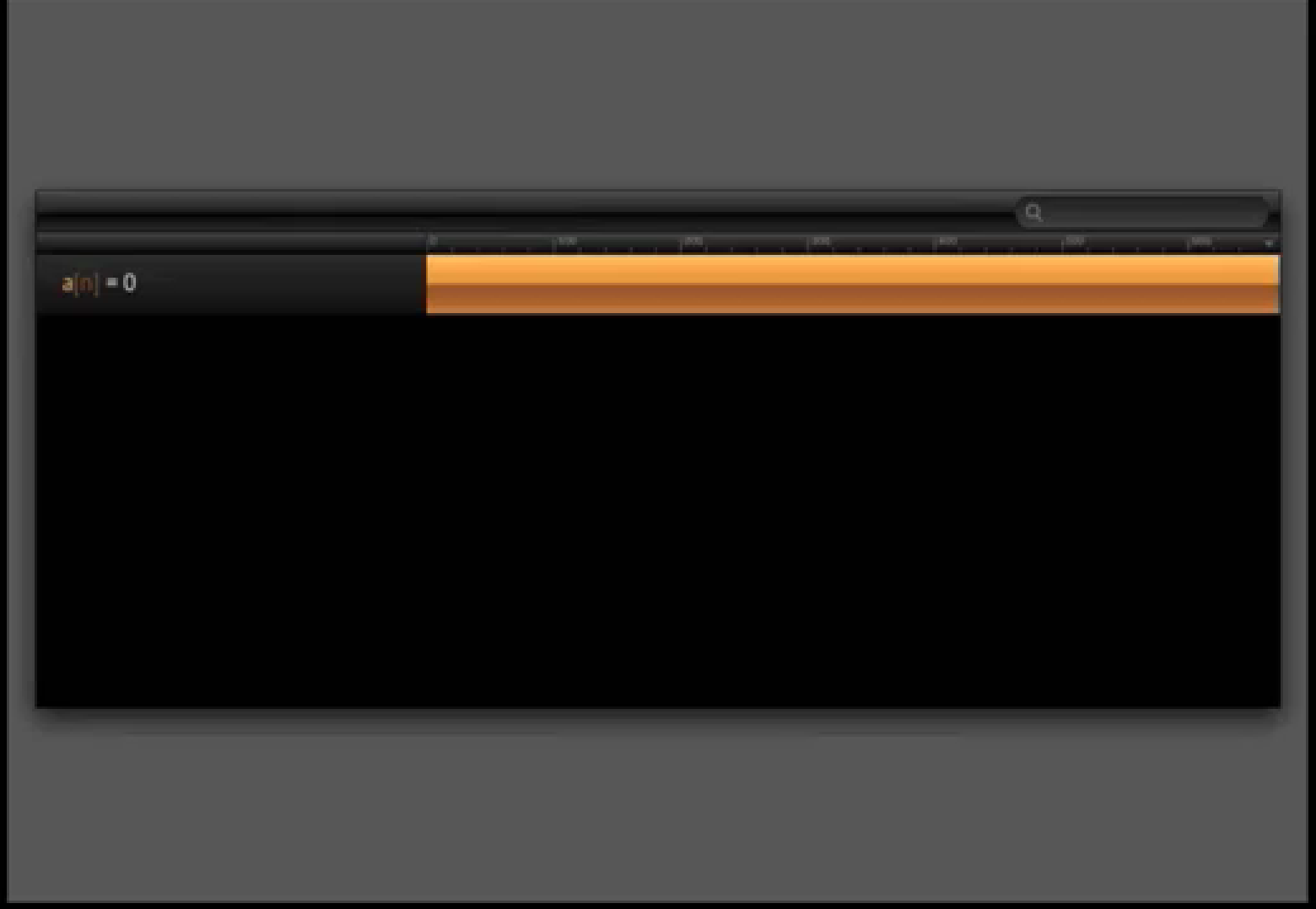
consider more closely some
of the user interface ideas



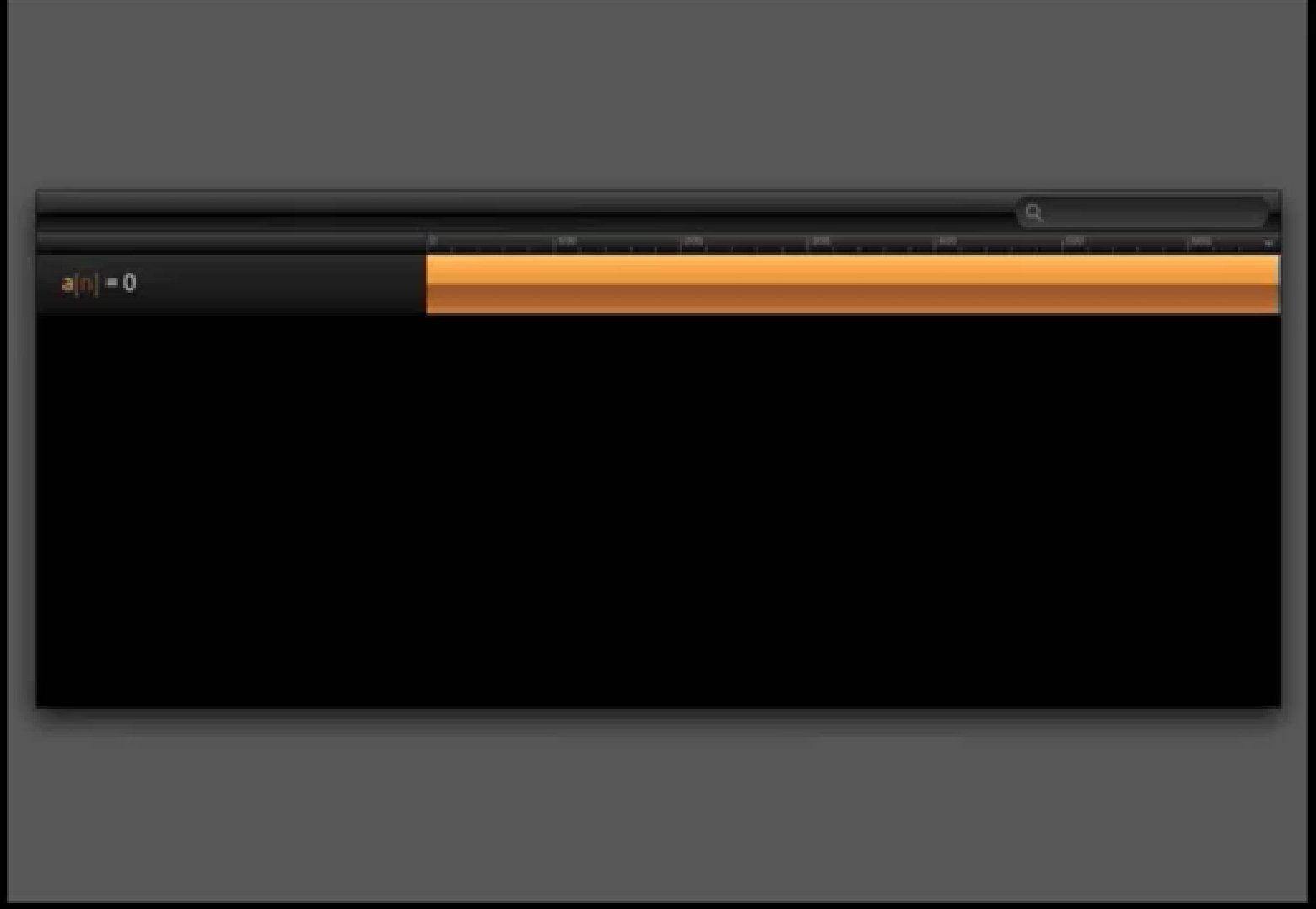
the “joining trick”



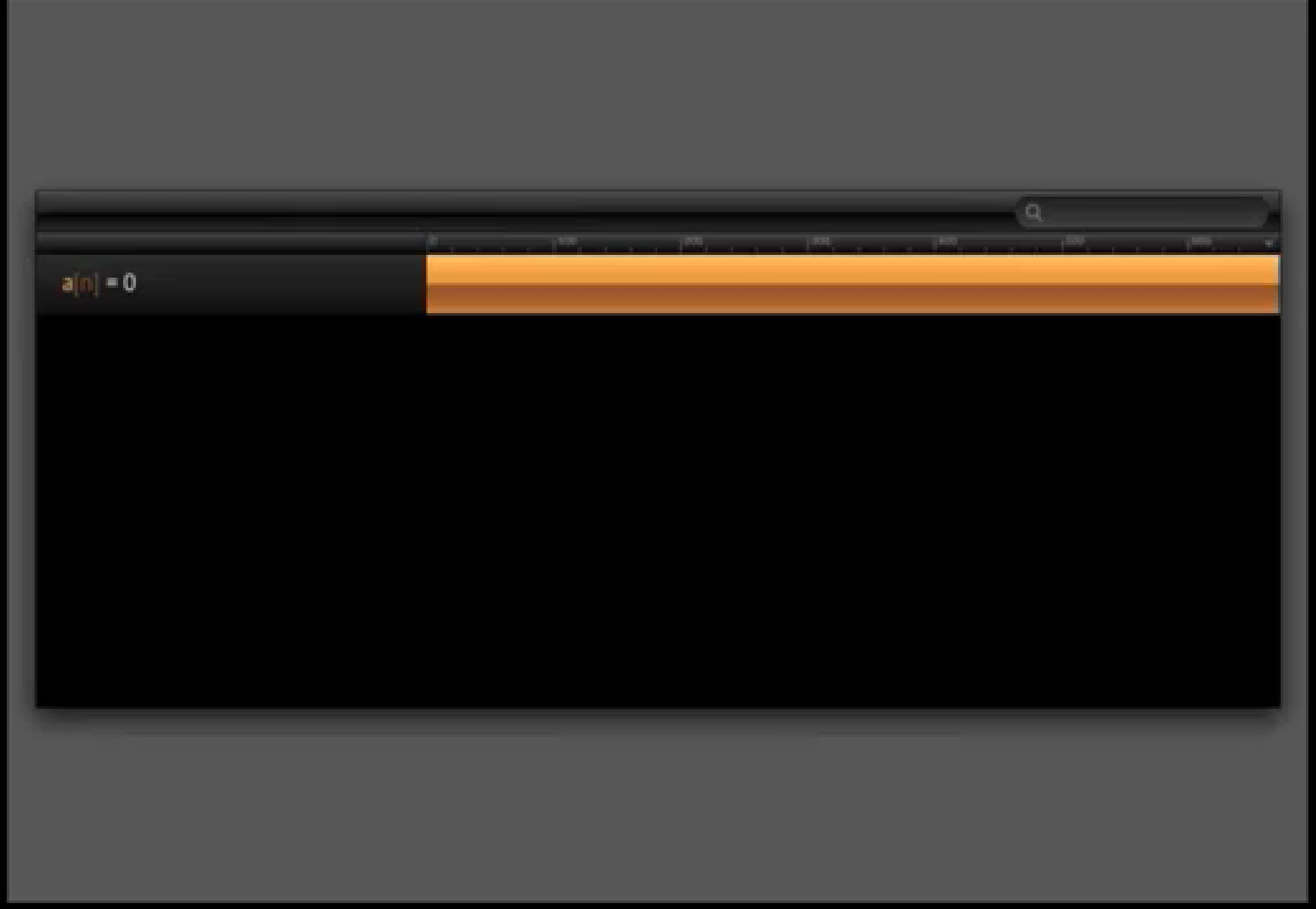
"searching over an expression"



these are fundamental new operations to do with difference equations



they actually expand the range of our thinking



they're new atoms of cognition

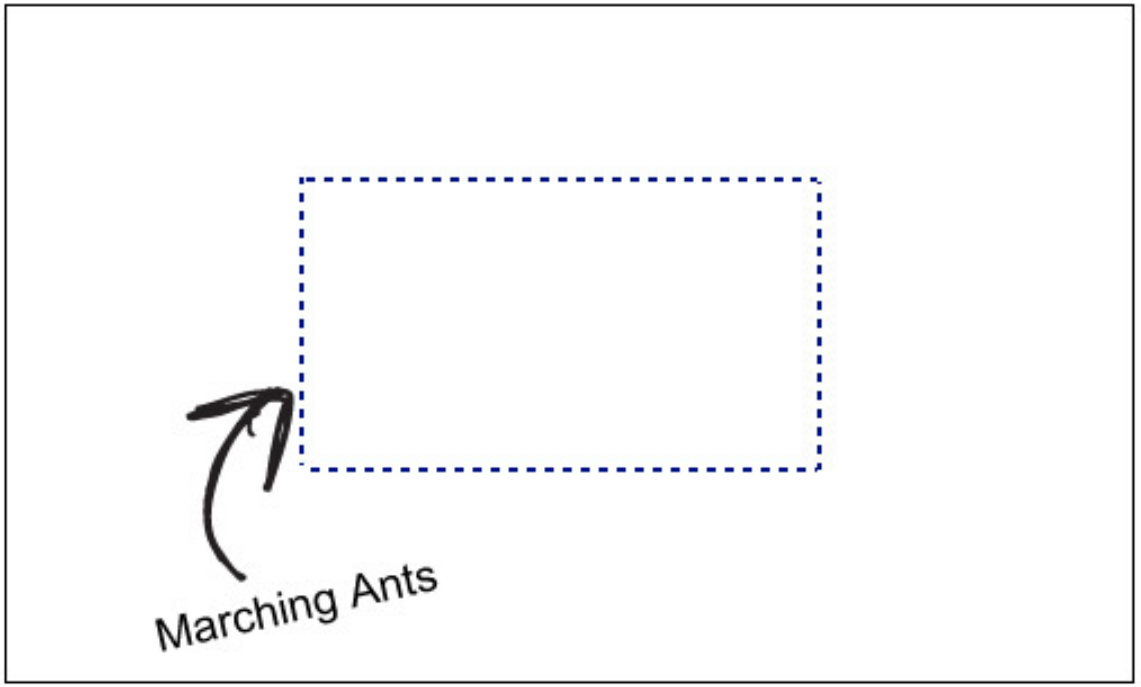
maybe cognitive media will stimulate
the invention of many new
atoms of cognition

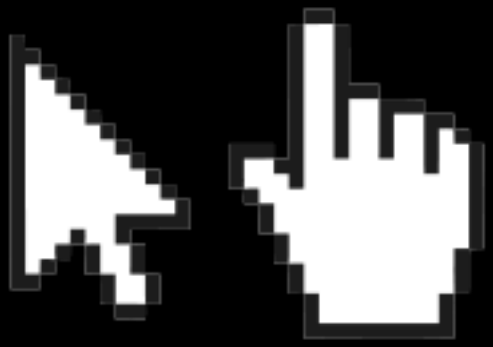
Hyperlink

From Wikipedia, the free encyclopedia

For help creating links in Wikipedia, see Help:Contents/Links.

In computing, a **hyperlink** is a
the reader can directly follow e
hovering.[1] A hyperlink points t
to a specific element within a d
text with hyperlinks. A software
viewing and creating hypertext
and to create a hyperlink is *to l*
link). A user following hyperlink
browse the hypertext.





all these ideas had to be invented

What new primitives
for thought can we invent?

What's all this got to do with open access?

Open access to what?

Do repositories like the arXiv and PubMed provide a platform for experimentation with more powerful media forms?

How should open access policies be crafted to ensure we don't inhibit innovation by constraining experimentation?

How can we ensure wild experimentation with new media forms?

Thankyou

http://michaelnielsen.org/

@michael_nielsen