Computational Neuroscience 2025

Lecture 5

2025-10-01

Linear-nonlinear model application; Event-based representation; Inhomogeneous Poisson process

- Online slides: lec05-event_driven.html
- Code: code05.ipynb
- Homework: hw05.pdf

Username: `cns`, Password: `nycu2025`

Laboratory of Computing Biological Networks

Estimating the firing rate from the stimulus

Linear dependence

$$\mathrm{r}_{\mathrm{est}}(t) = \mathrm{r}_0 + \int_0^\infty d\tau D(\tau) s(t-\tau) \tag{2.1}$$

Weighted contribution from each past moment is summed up.

Generalization to Volterra (Wiener) expansion

$$\mathrm{r}_{\mathrm{est}}(t) = \mathrm{r}_0 + \int d\tau D(\tau) s(t-\tau) + \int d\tau_1 d\tau_2 D_2(\tau_1,\tau_2) s(t-\tau_1) s(t-\tau_2) + \\ \int d\tau_1 d\tau_2 d\tau_3 D_3(\tau_1,\tau_2,\tau_3) s(t-\tau_1) s(t-\tau_2) s(t-\tau_3) + \dots \tag{2.2}$$

where D is the Wiener kernel (or simply "*kernel*").

Optimization for the best kernel

Minimizing the cost function

$$E = \frac{1}{T}\int_0^T dt[\mathrm{r}_{\mathrm{est}}(t) - r(t)]^2 \tag{2.3}$$

⇒ The resulting D satisfies

$$\int_0^\infty d\tau' Q_{ss}(\tau - \tau')D(\tau') = Q_{\mathrm{rs}}(-\tau) \tag{2.4}$$

White noise stimulus $Q_{ss}(\tau) = \sigma_s^2\delta(\tau)$ results in

$$D(\tau) = \frac{Q_{\mathrm{rs}}(-\tau)}{\sigma_s^2} = \frac{\langle r\rangle C(\tau)}{\sigma_s^2} \tag{2.6}$$

Using STA as the optimal kernel

The spike-triggered average can serve as an estimate of the optimal kernel in linear–nonlinear model that simulates the firing of the neuron under a stimulus.

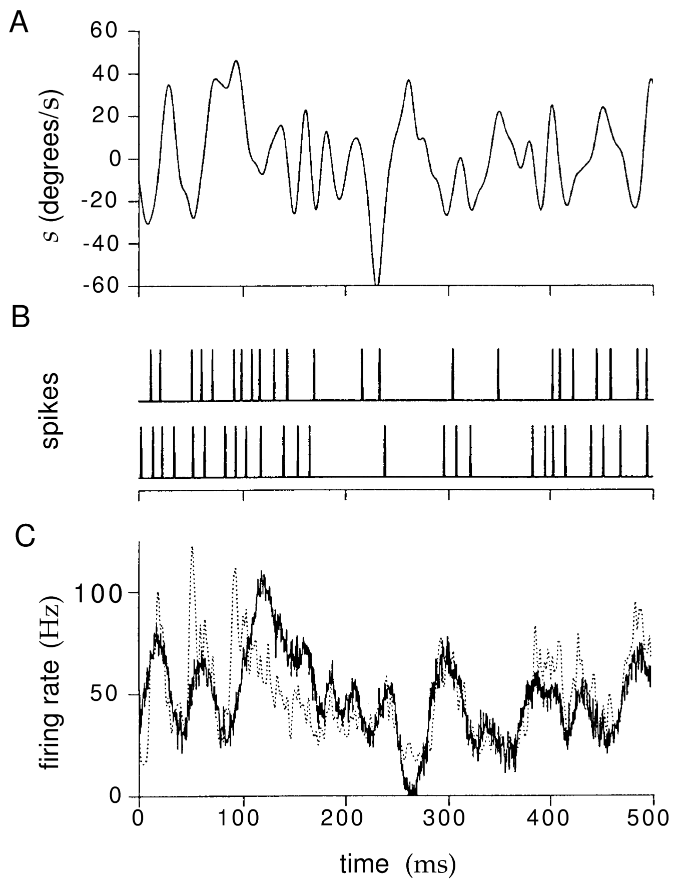


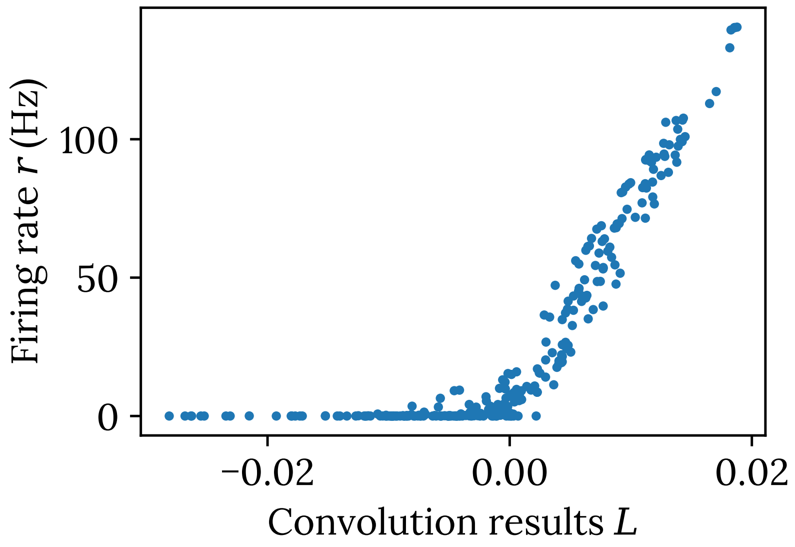
Figure 2.1 Prediction of the firing rate for an H1 neuron responding to a moving visual image. (A) The velocity of the image used to stimulate the neuron. (B) Two of the 100 spike sequences used in this experiment. (C) Comparison of the measured and computed firing rates. The dashed line is the firing rate extracted directly from the spike trains. The solid line is an estimate of the firing rate constructed by linearly filtering the stimulus with an optimal kernel. (Adapted from Rieke et al., 1997.)

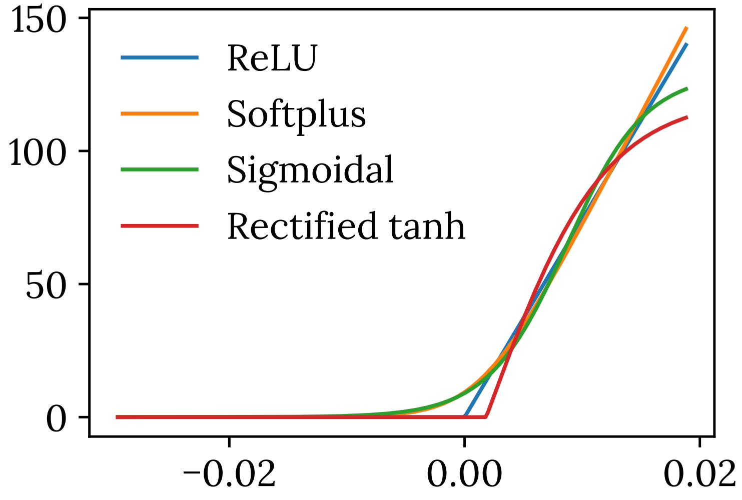
Dayan & Abbott (2007)

Static nonlinearity

Linear contribution from stimulus: $L(t) = \int_0^\infty d\tau D(\tau) s(t-\tau)$.
Nonlinear prediction:

$$\mathrm{r}_{\mathrm{est}}(t) = \mathrm{r}_0 + F(L(t)) \tag{2.8}$$





Different choices of rectification

- ReLU

$$F(L) = G[L - L_0]_+ \tag{2.9}$$

- Softplus

$$F(L) = G \ln\{1 + \exp[g_0(L - L_0)]\}$$

- Sigmoidal

$$F(L) = \frac{\mathrm{r}_{\mathrm{max}}}{1 + \exp\left(g_1(L_{1/2} - L)\right)} \tag{2.10}$$

- Rectified hyper-tangent

$$F(L) = \mathrm{r}_{\mathrm{max}}[\tanh(g_2(L - L0))]_+ \tag{2.11}$$

Representations of point processes

Frame-based representation

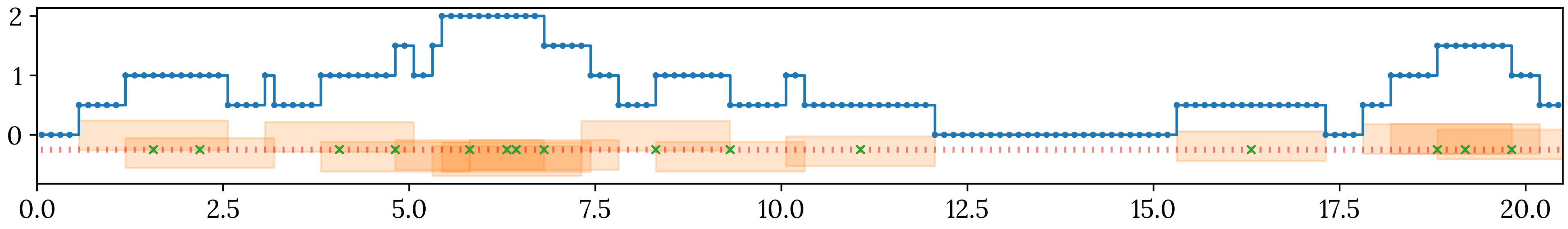
High time precision requires small time-step, leads to large fluctuations in estimated rates that need averages over repeatable trials to alleviate.

Event-based representation

Time precision is limited by machine hardware. Need to handle variable data size for fixed time duration. May need complex considerations to preserve the time-precision.

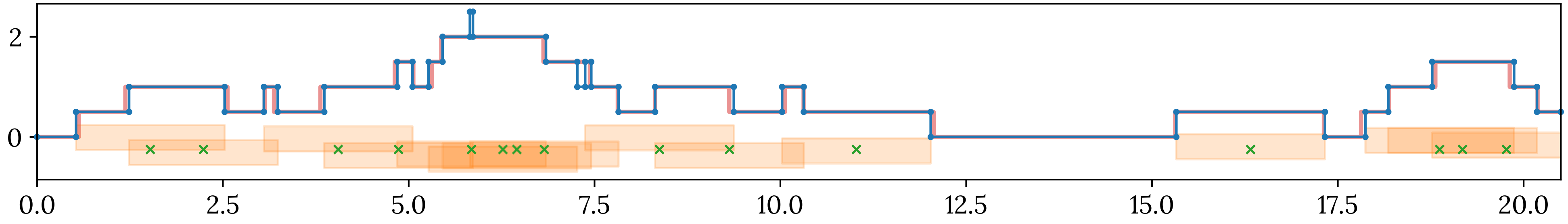
Sliding window in two representations

Frame-based



Event times, window size and positions are discretized to step size. One point for each frame even when things are not changing.

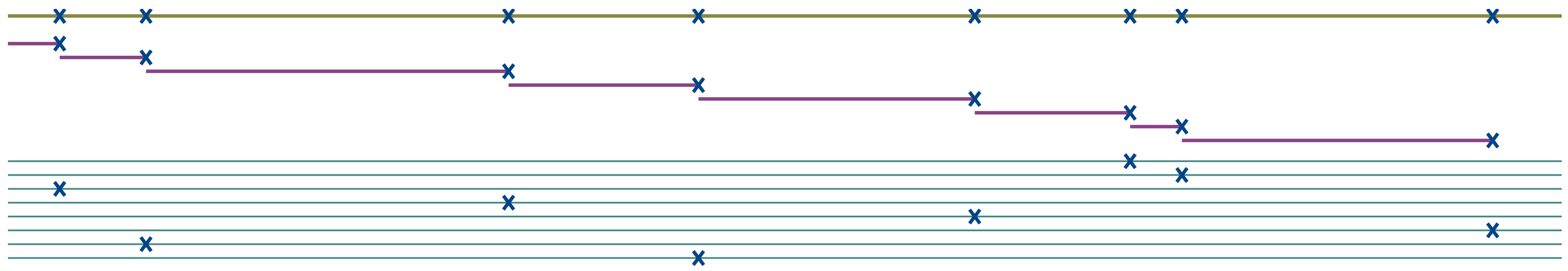
Event-based



Numerically exact precision for time. Data stored when and only when something happens. An implementation:

https://networks.tir.tw/wiki/technical:sliding_window

Independent intervals and independent events



In a renewal process, you only need to keep track of the last event time. The time to the next event is drawn anew from a random distribution.

For Poisson process, the event times are drawn independently.

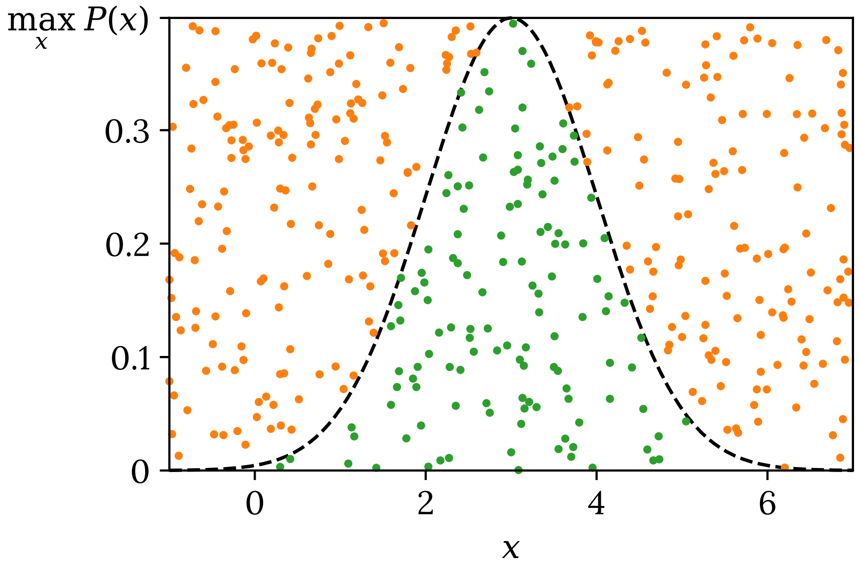
- For homogeneous Poisson process (constant rate μ), the event times can be generated by accumulating independent intervals from an exponential distribution.

```
np.cumsum(rng.exponential(size=N)/mu)
```

Random variable for any distribution

Monte Carlo methods: stochastic algorithms which will stop and produce results that are very likely to be correct.

Gaussian (normal) distribution



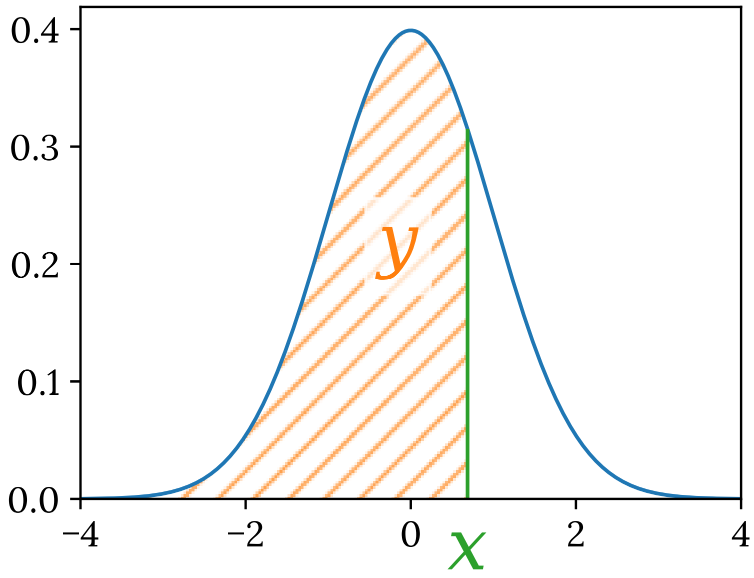
1. Generate a uniform random number v in the range of the distribution and decide whether to keep it based on the probability $P(v)/P_{\max}$.
2. Repeat 1. until we have enough random numbers.

- Any distribution function with finite range of interest can be generated.
- Precision is limited by the frame step size for discretization.
- May drop many random points if $P_{\max} \gg \bar{P}$.

Probability integral transformation

Probability distribution of random variable x is $P(x)$.

Define: $y \equiv \int_{-\infty}^{x} P(x')dx'$



Distribution for y:

$$P(y)dy = P(x)dx$$

$$\Rightarrow P(y) = P(x)\frac{dx}{dy} = P(x)\left(\frac{dy}{dx}\right)^{-1} = P(x)P^{-1}(x) = 1 \qquad (2)$$

Thus, y is uniformly distributed between 0 and 1. If $y = F(x)$, then $x = F^{-1}(y)$ transform a uniform RV to that follows $P(x)$.

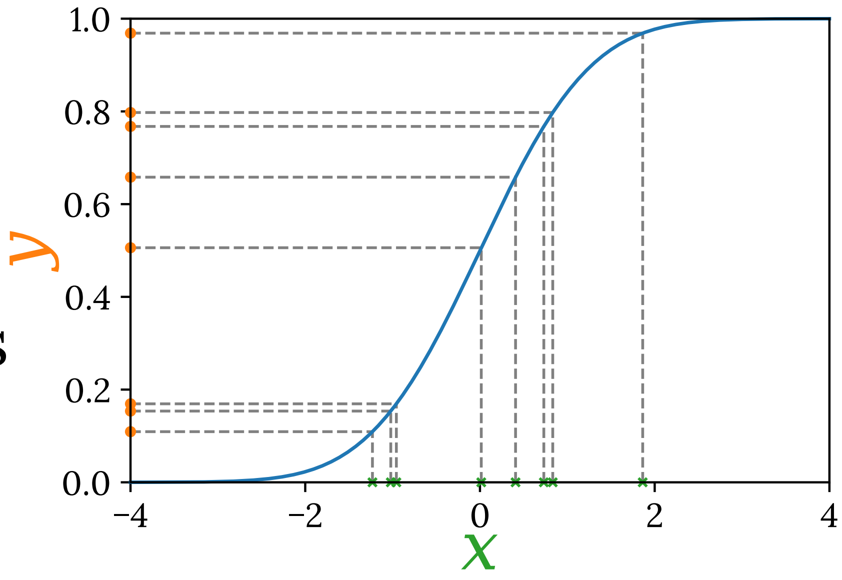
Integration and inversion of Gaussian

Gaussian distribution: $P(x) = \frac{1}{\sqrt{2\pi}} \exp - \frac{x^2}{2}$

$$\Rightarrow y = \int_{-\infty}^{x} dx P(x) = \frac{1}{2}\left(1 + \operatorname{erf}\frac{x}{\sqrt{2}}\right)$$

$$\Rightarrow x = \sqrt{2}\operatorname{erf}^{-1}(2y-1) \qquad (3)$$

where the error function `erf` and its inverse `erfinv` are available in the `scipy.special` library.



```
xs = erfinv(rng.uniform(size=sz)*2-1)*np.sqrt(2)
```

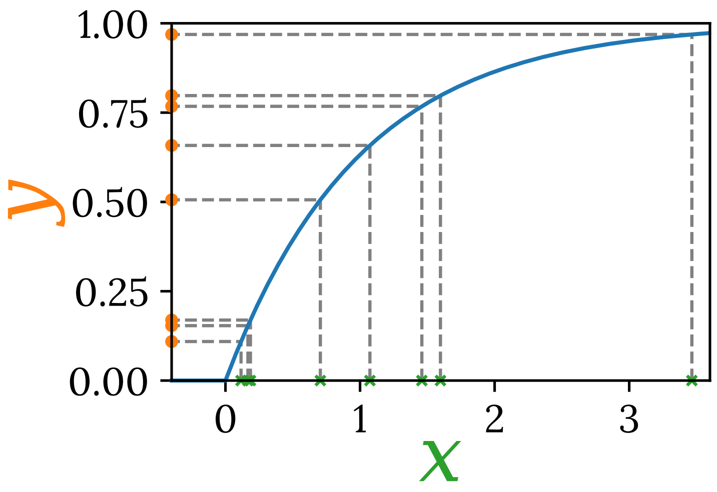
Integration and inversion of exponential

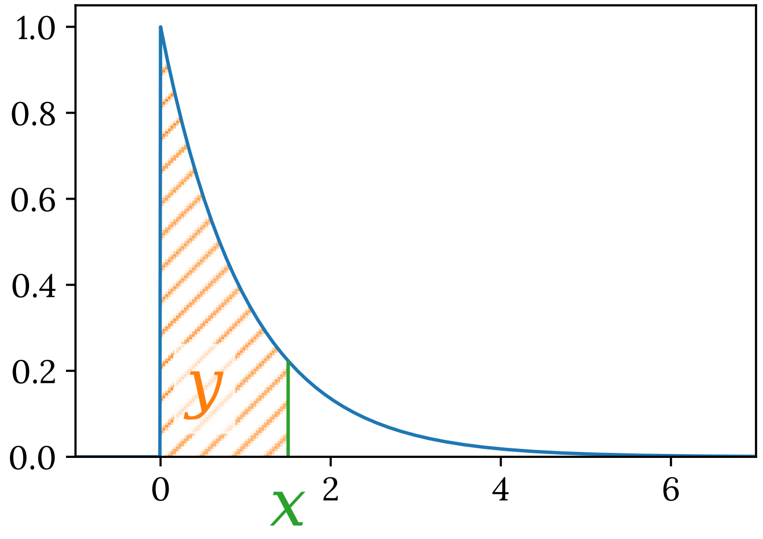
Normalized exponential: $P(x) = \frac{1}{\tau}\exp\left(-\frac{x}{\tau}\right)\theta(x)$

τ: mean value of x; $\theta(\cdot)$: Heaviside step function

$$\Rightarrow y = \int_{-\infty}^{x} dx' \, \frac{1}{\tau} e^{-x'/\tau} dx' = 1 - \exp\left(-\frac{x}{\tau}\right)$$

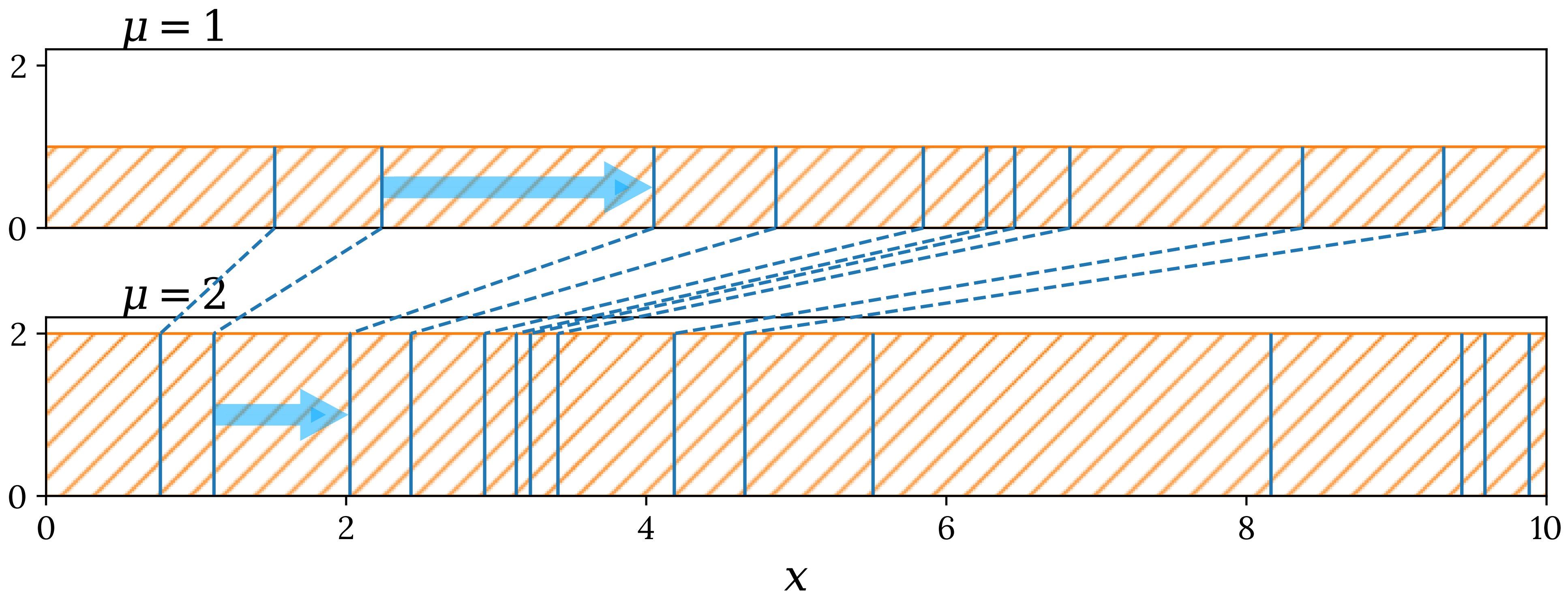
$$\Rightarrow x = -\tau \ln(1 - y)$$





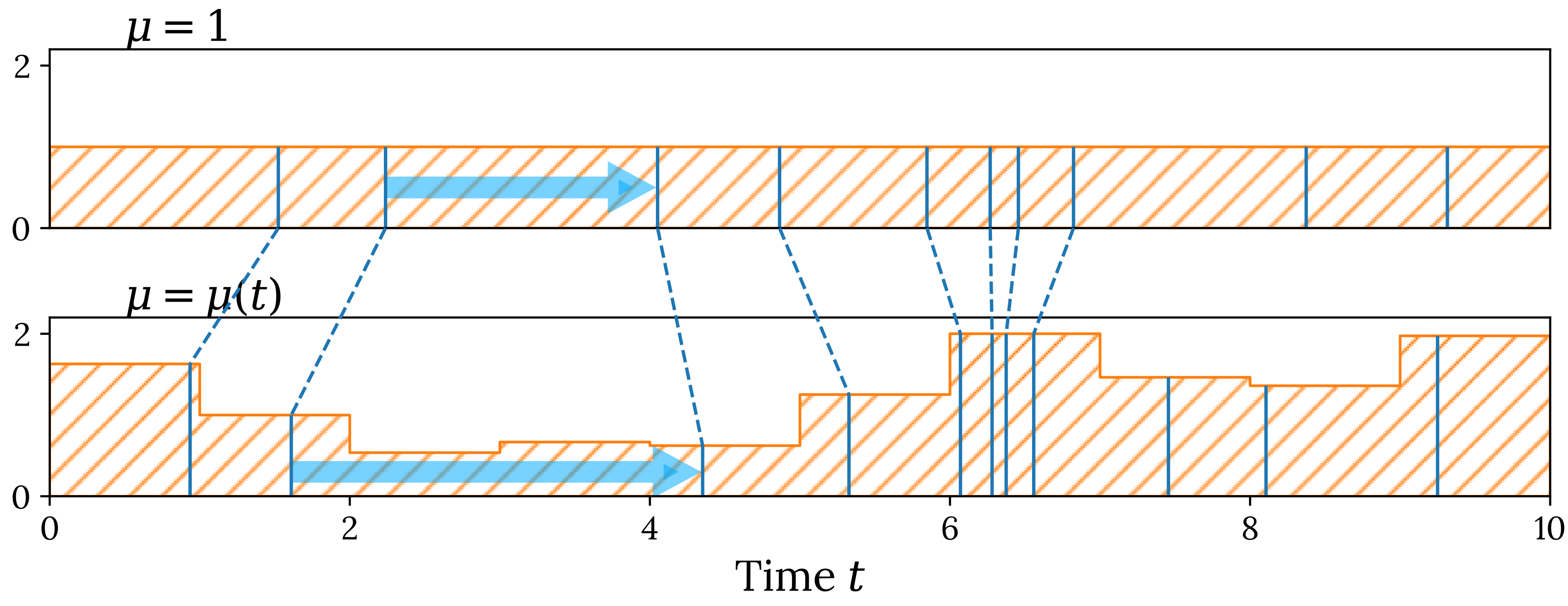
Rate $\mu = 1/\tau$

Generation of spike trains of different rates



Poisson process of different rates can be generated by filling different widths of strips with the same area sequences.

Inhomogeneous Poisson process



Using the same area sequences, inhomogeneous Poisson process can be generated by filling a strip of varying width. An implementation is available at:

https://networks.tir.tw/wiki/technical:poisson_spike_train

Implementation details of varying μ

```python
def gen_spikes(r,dt,rng=None,seed=123):
  '''Generate spike train from Poisson rate

  Parameters
  ----------
  r:  Array of spike rates
  dt: Time step size

  Returns
  -------
  Event-based spike train
  '''
  if rng is None: rng = np.random.default_rng(seed)
  i = 0
  s = 0
  spks = []
  while True:
    s += rng.exponential() # Area to advance for next spike
    while s>r[i]*dt: # Advance to next frame if enough area
      s -= r[i]*dt
      i += 1
      if i>=len(r): break # No more frame
    else:
      spks.append(i*dt+s/r[i]) # Spike time in this frame
      continue
    break
  return np.array(spks)
```

- Advancing area under the curve by exponentially distributed random amount.
- No rejections of random points.
- Better spike-time resolution upto machine precision.

Technical aspect: Function usage

Use function to: • Reduce redundancy (reuse code) • Improve readability (proper naming, decomposition, information hiding) • Enable recursion

Do document your function using docstring. It can be accessed in jupyter-notebook through the Shift-TAB key.

```python
def gau_win(x,sigma):
  '''Gaussian window function with sigma as STD

  Parameters
  ----------
  x: Variable
  sigma: Standard deviation

  Returns
  -------
  Values of Gaussian distribution function at x
  '''
  return np.exp(-(x/sigma)**2/2)/(np.sqrt(2*np.pi)*sigma)
```

☆ Function can also be anonymous in python. E.g.

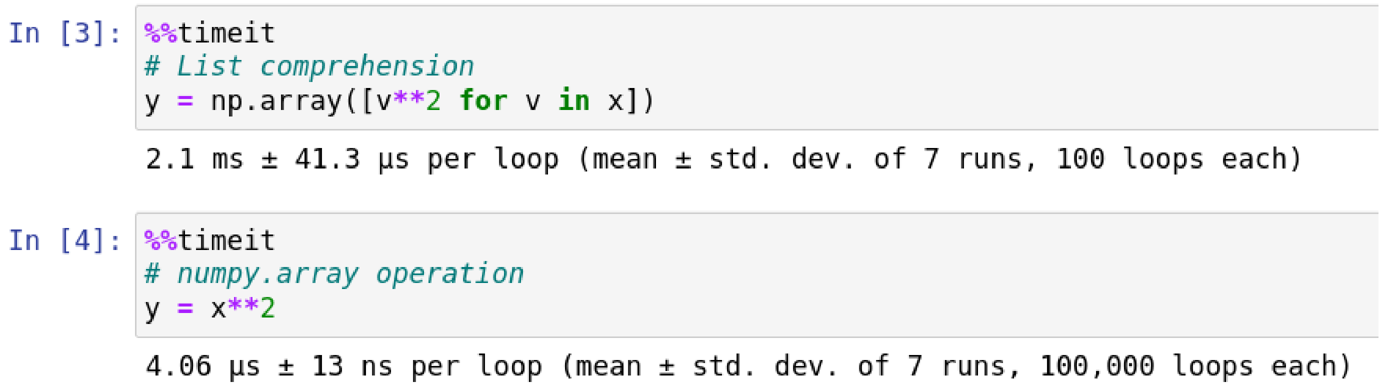
```python
gau_win = lambda x,sigma:np.exp(-(x/sigma)**2/2)/(np.sqrt(2*np.pi)*sigma)
curve_fit(lambda x,a,b,c:a*x**2+b*x+c,xdat,ydat)
```

Technical aspect: Loop vs vectorized operations

```
# List appending loop
spk0 = []
t = 0
while True:
  t += rng.exponential()
  if t>100: break
  spk0.append(t)
# Operating with numpy.array
spk1 = np.cumsum(rng.exponential(size=100))
# List comprehension
t = 0
spk2 = [t:=t+rng.exponential()
    for _ in range(100)]
```

It's preferable to convert loops to operations with `numpy.array` as they can be vectorized by the library if possible.

We can use the magic command `%%timeit` to confirm this.



```
In [3]: %%timeit
        # List comprehension
        y = np.array([v**2 for v in x])

        2.1 ms ± 41.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [4]: %%timeit
        # numpy.array operation
        y = x**2

        4.06 µs ± 13 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Technical aspect: Iterables and generators

Iterables: objects holding values that can be retrieved one by one. **Iterators:** indicators for going through the values in iterables. Examples in native types are lists, tuples, strings, dictionaries, sets, and objects like range, zip, map...

```
aitb = ['dog','pig','cat','rat']
i = iter(aitb)
try:
  while True: print(next(i))
except StopIteration:
  pass
```

Generators: functions that will return an iterable with values that are produced by the `yield` statements in the functions

```
def gen_poisson(tt):
  t = 0
  while (t:=t+rng.exponential())<tt:
    yield t
# Convert it to a list
print(list(gen_poisson(10)))
```