Most scientific papers are probably wrong.
...there is less than a 50% chance that the results of any randomly chosen scientific paper are true.
Kurt Kleiner, NewScientist.com - www.newscientist.com/article.ns?id=dn7915

Artificial Biochemistry

# Probability Distributions


## Luca Cardelli

### Microsoft Research

The Microsoft Research - University of Trento
Centre for Computational and Systems Biology


**Trento, 2006-05-22..26**

**www.luca.demon.co.uk/ArtificialBiochemistry.htm**

# Exponential Decay

A quantity subject to exponential decay decreases at a rate proportional to its value:

$$\frac{dN}{dt} = -\lambda N.$$

where N is the quantity and $\lambda > 0$ is the decay rate

Solution of the equation:

$$N = Ce^{-\lambda t}.$$

where $C$ is the initial value of the quantity

Half life:

$$t_{1/2} = \frac{\ln 2}{\lambda}.$$

time of halving of the initial quantity $C$, independent of $C$

Mean lifetime:

$$\tau = \frac{1}{\lambda}.$$

average length of time an element remains in an exponentially decaying discrete set

Poisson processes:

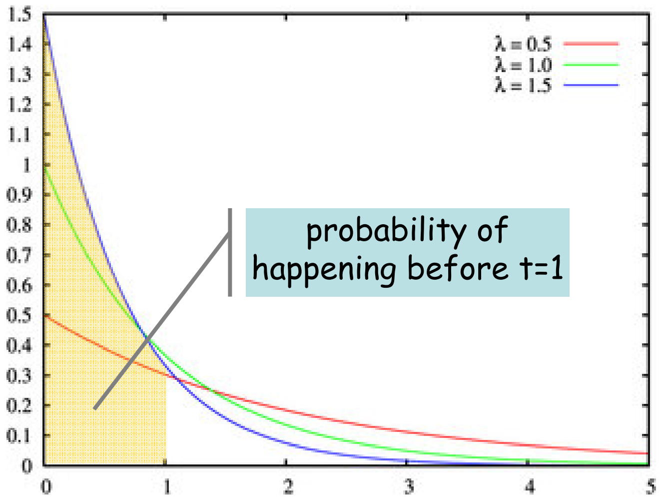
Exponential decay leads to the exponential distribution , which is used to model (homogeneous 1-dimensional) Poisson processes, which are situations in which an object initially in state A can change to state B with constant probability per unit time $\lambda$. The time at which the state actually changes is described by an exponential random variable with parameter $\lambda$. Therefore, the integral from 0 to $T$ over $f$ is the probability that the object is in state B at time $T$.

# Exponential Distribution

$$\frac{dN}{dt} = -\lambda N.$$

$$N = Ce^{-\lambda t}.$$

- http://en.wikipedia.org/wiki/Exponential_distribution
  - Probability density function (with *rate parameter* λ > 0)

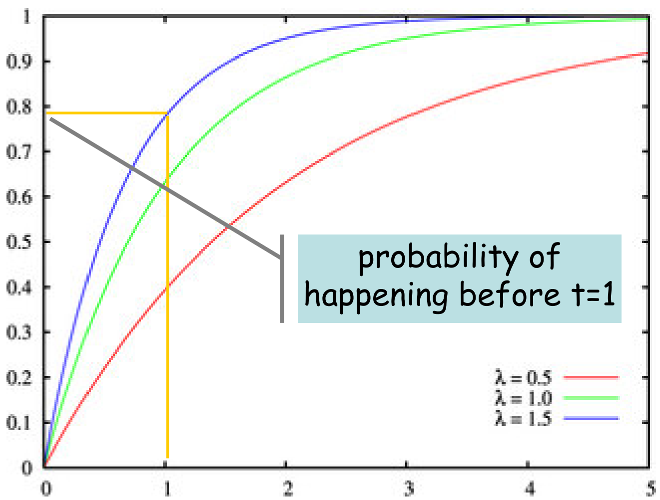



rate of decay λ

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & , \; x \ge 0, \\ 0 & , \; x < 0. \end{cases}$$

A probability density function is non-negative everywhere and its integral from −∞ to +∞ is equal to 1. If a probability distribution has density *f*(*x*), then intuitively the infinitesimal interval [*x*, *x* + d*x*] has probability *f*(*x*) d*x*.

  - Cumulative distribution function




$$F(x;\lambda) = \begin{cases} 1 - e^{-\lambda x} & , \; x \ge 0, \\ 0 & , \; x < 0. \end{cases}$$

For every real number *x*, the **cumulative distribution function** is given by

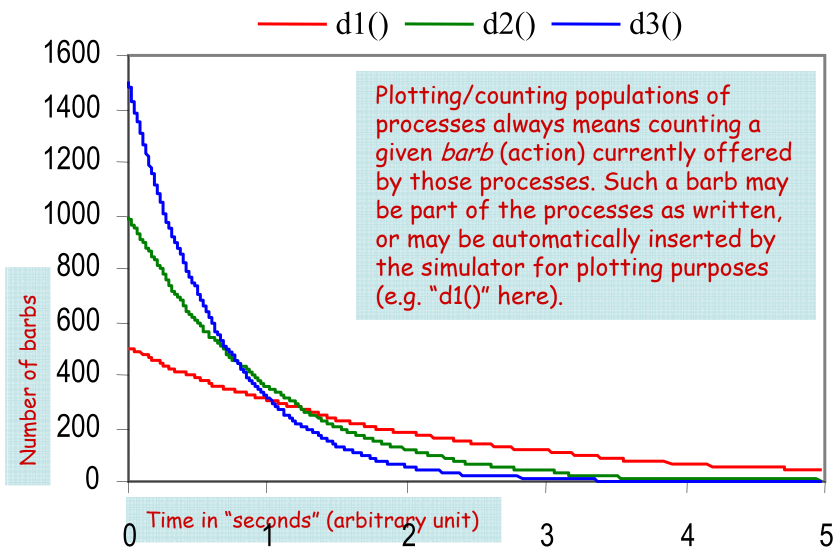
$$F(x) = \mathrm{P}(X \le x),$$

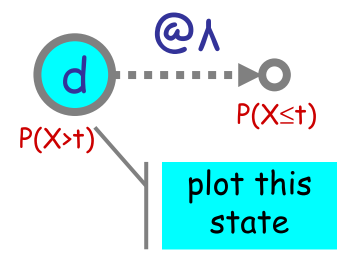
where the right-hand side represents the probability that the random variable *X* takes on a value less than or equal to *x*. The probability that *X* lies in the interval (*a*, *b*] is therefore *F*(*b*) − *F*(*a*) if *a* < *b*.

Hence: P(X > x) = F(∞)-F(t) = $e^{-\lambda x}$

# Plotting Exponential Distributions

## Probability Density Function f(t) ( = $\lambda$P(X>t) )

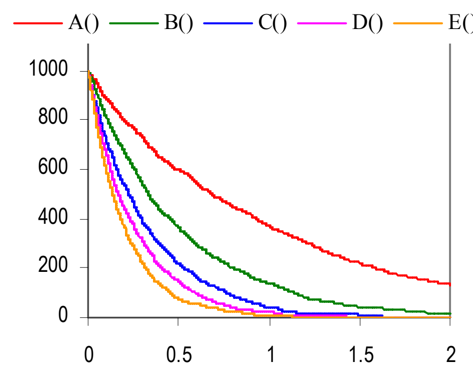



actually plotting 1000 * $\lambda$ * P($X$>$t$) where P($X$>$t$) = $e^{-\lambda t}$
(which just happens to be the same as 1000 * f(t) ! )

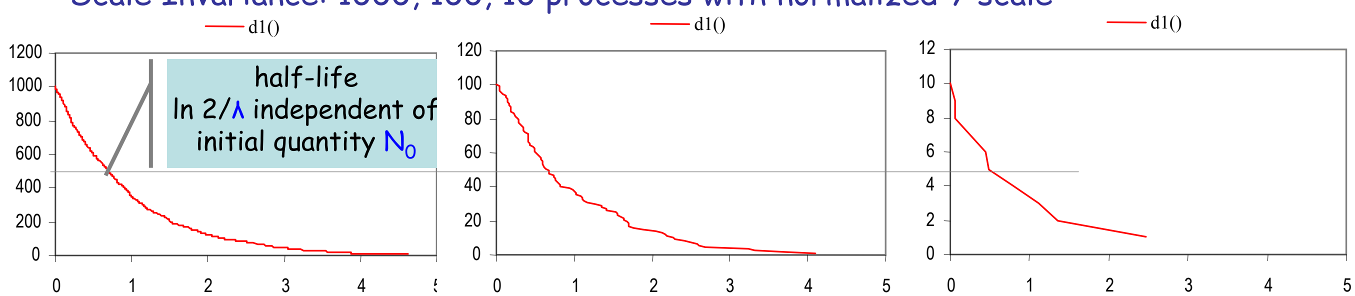
```
directive sample 5.0
directive plot d1(); d2(); d3()

let d1() = delay@0.5; ()
let d2() = delay@1.0; ()
let d3() = delay@1.5; ()

run  500 of d1()
run 1000 of d2()
run 1500 of d3()
```

For $\lambda$=1, if I start with 1000 things, and after 2sec I find 135 left, then P(delay > 2sec) = 135/1000 = 0.135 ~ $e^{-\lambda 2}$




```
directive sample 2.0
directive plot A(); B(); C(); D(); E()

let A() = delay@1.0; ()
and B() = delay@2.0; ()
and C() = delay@3.0; ()
and D() = delay@4.0; ()
and E() = delay@5.0; ()

run 1000 of (A() | B() | C() | D() | E())
```

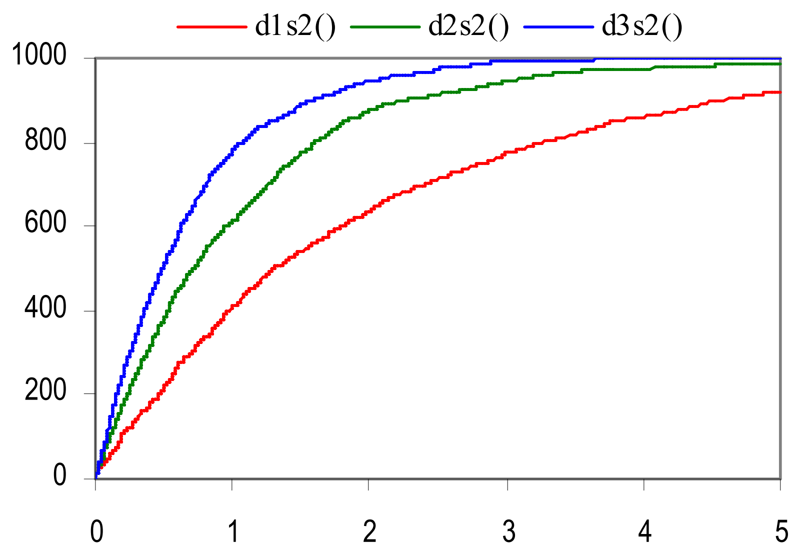
Scale Invariance: 1000, 100, 10 processes with normalized Y scale



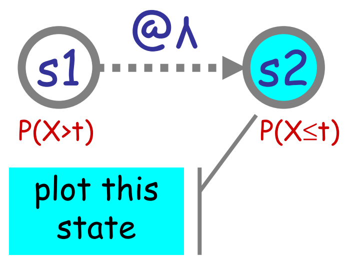

```
directive sample 5.0
directive plot d1()

let d1() = delay@1.0; ()

run 100 of d1()
```

# Plotting Exponential Distributions

## Cumulative Distribution Function P(X≤t)




For λ=1, if I start with 1000 things, and after 2sec I find 865 in S2, then P(delay ≤ 2sec) = 865/1000 = 0.865 ~ $1-e^{-\lambda 2}$




plotting 1000 * P($X \le t$)
where P($X \le t$) = $1-e^{-\lambda t}$

```
directive sample 5.0
directive plot d1s2(); d2s2(); d3s2()

let d1s2() = ()
let d2s2() = ()
let d3s2() = ()

let d1s1() = delay@0.5; d1s2()
let d2s1() = delay@1.0; d2s2()
let d3s1() = delay@1.5; d3s2()

run 1000 of d1s1()
run 1000 of d2s1()
run 1000 of d3s1()
```

# Exponential Distribution
## Basic Properties

- Characterized by a single positive real *rate* parameter $\lambda$
  - $P(X_\lambda \le t) = 1-e^{-\lambda t}$ — X is the *delay* before the event
- Memoryless (the <u>only</u> such continuous probability distribution)
  - $P(X > t_0+t \mid X>t_0) = P(X > t)$

    people knocking on my door at $\lambda$ = 1-knock-per-hour. $P(\text{Knock} > N_{hours})$ = "prob. of being knock-free for N hours"

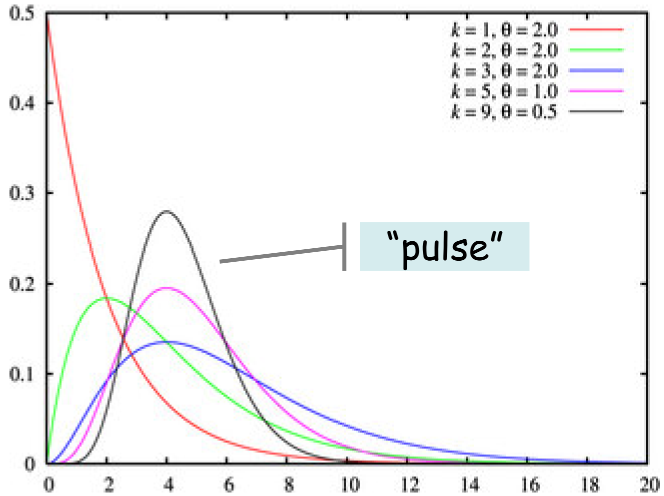
    $P(\text{Knock} > 5_{hours} \mid \text{Knock} > 3_{hours}) = P(\text{Knock} > 2_{hours}) = 13\%$
    $P(\text{Knock} > 48_{hours} \mid \text{Knock} > 46_{hours}) = P(\text{Knock} > 2_{hours}) = 13\%$
    We do not need to "remember" when we started counting! *memoryless*

    $P(\text{Knock} > 1_{hours}) = 36\%$
    $P(\text{Knock} > 5_{hours}) = 0.7\%$

    $P(\text{Knock} > 5_{hours} \mid \text{Knock} > 3_{hours}) = P(\text{Knock} > 2_{hours}) = 13\%$
    $P(\text{Knock} > 5_{hours} \mid \text{Knock} > 4_{hours}) = P(\text{Knock} > 1_{hours}) = 36\%$
    $P(\text{Knock} > 5_{hours} \mid \text{Knock} > 4.9_{hours}) = P(\text{Knock} > 0.1_{hours}) = 90\%$
    prob. gets better, but is just equal to the knock-free prob. for the remaining time
- Closed under min (cumulative exit rate of a choice):
  - $X = \min(X_1, \dots, X_n)$ is exponentially distributed if $X_i$ are independently exponential
  - $P(\min(X_\lambda, Y_\mu) \le t) = 1-e^{-(\lambda+\mu)t} = P(Z_{\lambda+\mu} \le t)$
- Comparisons between 2 variables (branch probabilities of a choice)
  - $P(X_\lambda < Y_\mu) = \lambda/(\lambda+\mu)$
  - $P(Y_\mu < X_\lambda) = \mu/(\lambda+\mu)$
  - $P(X_\lambda = Y_\mu) = 0$

# Erlang Distribution

- [http://en.wikipedia.org/wiki/Erlang_distribution](http://en.wikipedia.org/wiki/Erlang_distribution)
  - Probability density function (with *rate parameter* λ > 0, *shape parameter k*)

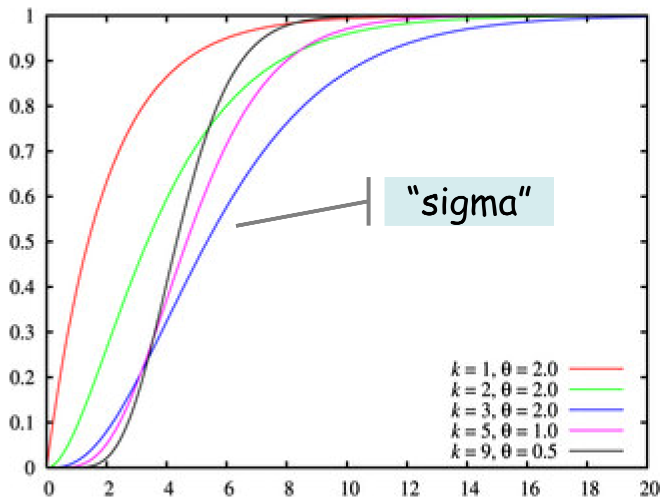



$$f(x;k,\lambda)=\frac{\lambda^k x^{k-1}e^{-\lambda x}}{(k-1)!}\quad \text{for } x>0.$$

$$f(x;k,\theta)=\frac{x^{k-1}e^{-\frac{x}{\theta}}}{\theta^k(k-1)!}\quad \text{for } x>0.$$ (θ = 1 / λ):

When the shape parameter *k* equals 1, the distribution simplifies to the exponential distribution.

  - Cumulative distribution function

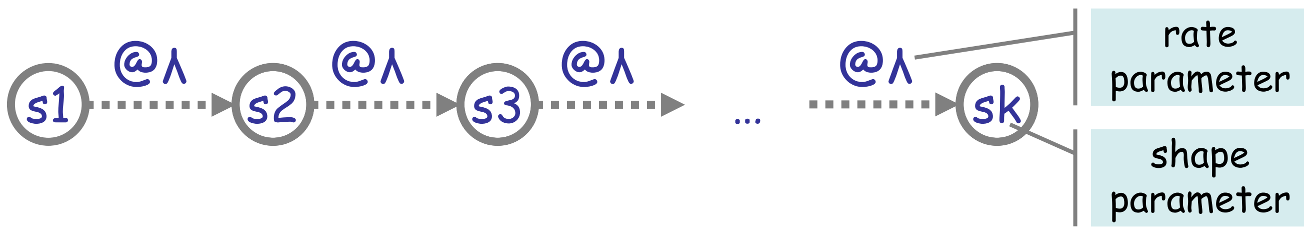


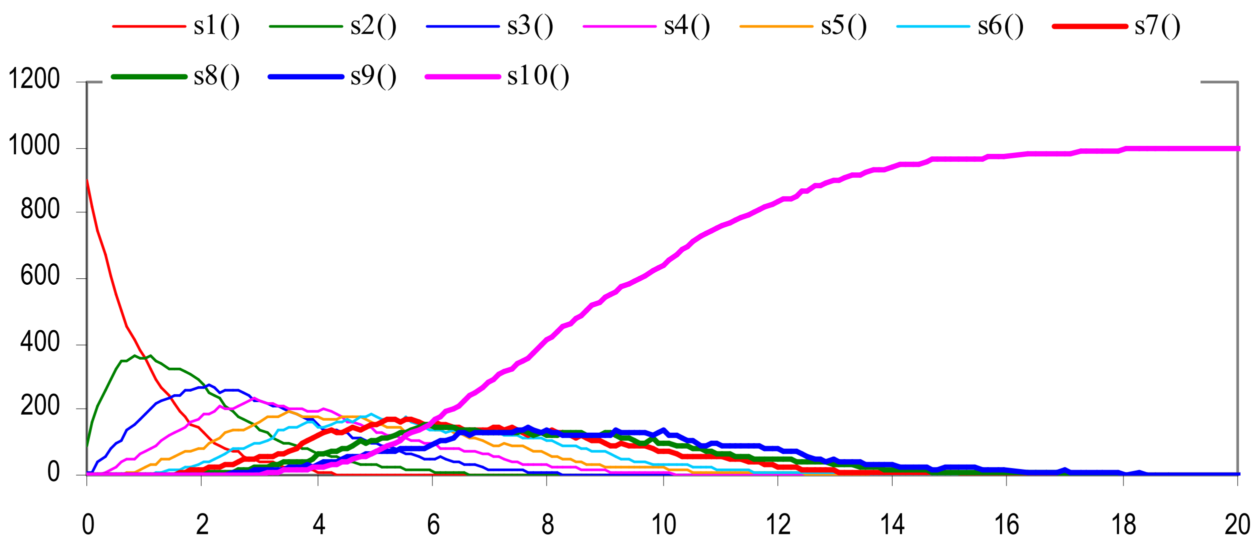

$$F(x;k,\lambda)=\frac{\gamma(k,\lambda x)}{(k-1)!}$$

where γ() is the incomplete gamma function.

An Erlang distribution (so named in honor of A. K. Erlang) is the probability distribution of the amount of time until the *n*-th event in a one-dimensional Poisson process with rate λ. I.e. the sum of n exponential distributions with the same rate λ.

# Erlang Distribution



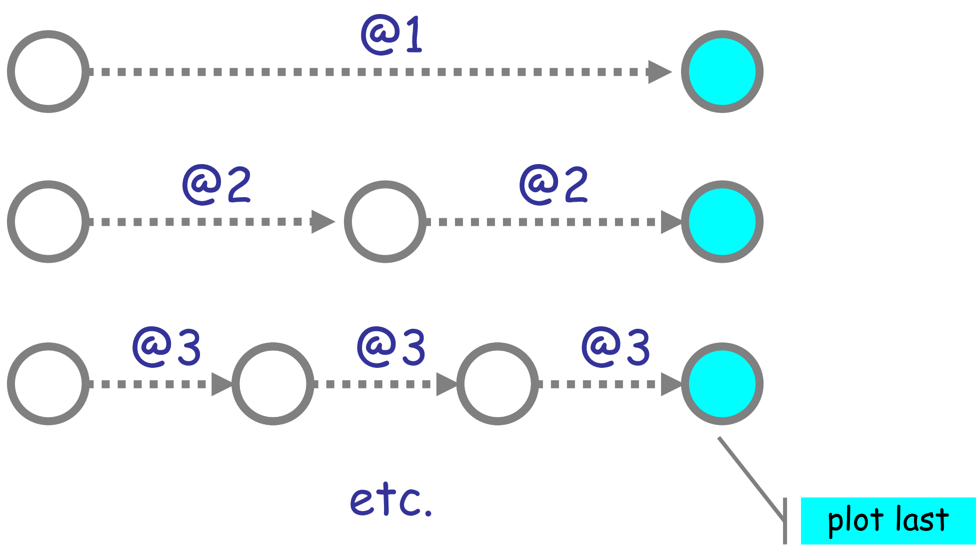



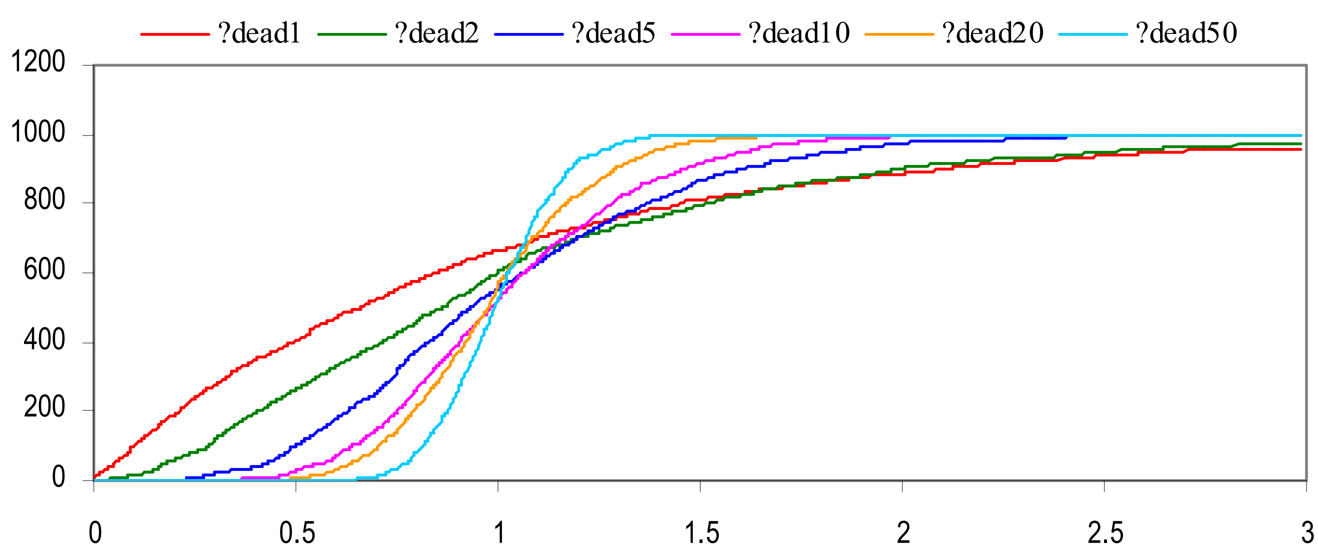

```
directive sample 100.0 1000
directive plot s1(); s2(); s3(); s4();
s5(); s6(); s7(); s8(); s9(); s10()

let s1() = delay@1.0; s2()
and s2() = delay@1.0; s3()
and s3() = delay@1.0; s4()
and s4() = delay@1.0; s5()
and s5() = delay@1.0; s6()
and s6() = delay@1.0; s7()
and s7() = delay@1.0; s8()
and s8() = delay@1.0; s9()
and s9() = delay@1.0; s10()
and s10() = ()

run 1000 of s1()
```

Erlang distribution a.k.a. Gamma distribution when n is a real number.

Erlang random variable $Y = X_1 + \ldots + X_n$ is the sum of n exponentially distributed random variables *with the same parameter*.
Expected value $E(Y) = E(X_1) + \ldots + E(X_n)$ (true for general random variables)
Standard deviation $\sigma(Y) = \sigma(X_1) + \ldots + \sigma(X_n)$ (true for general independent random variables)

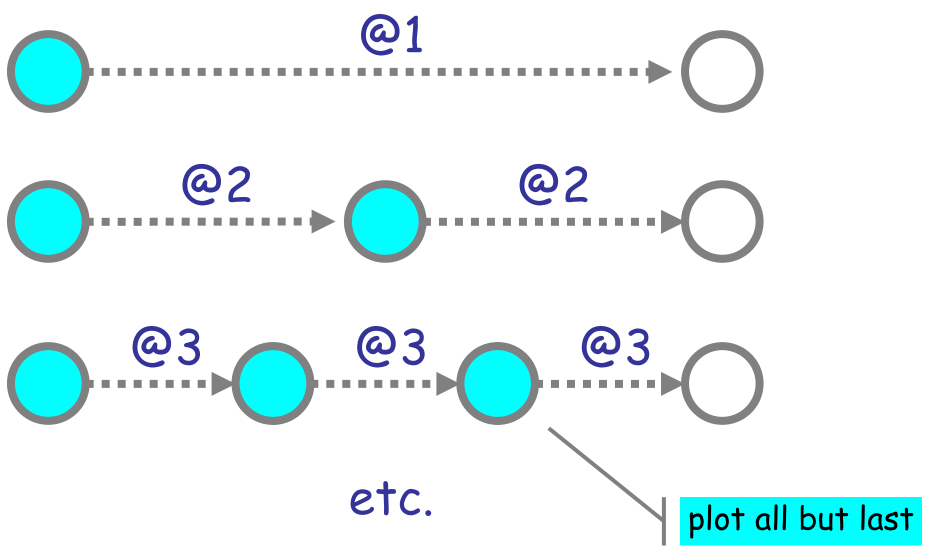
# Erlang Up-Transition



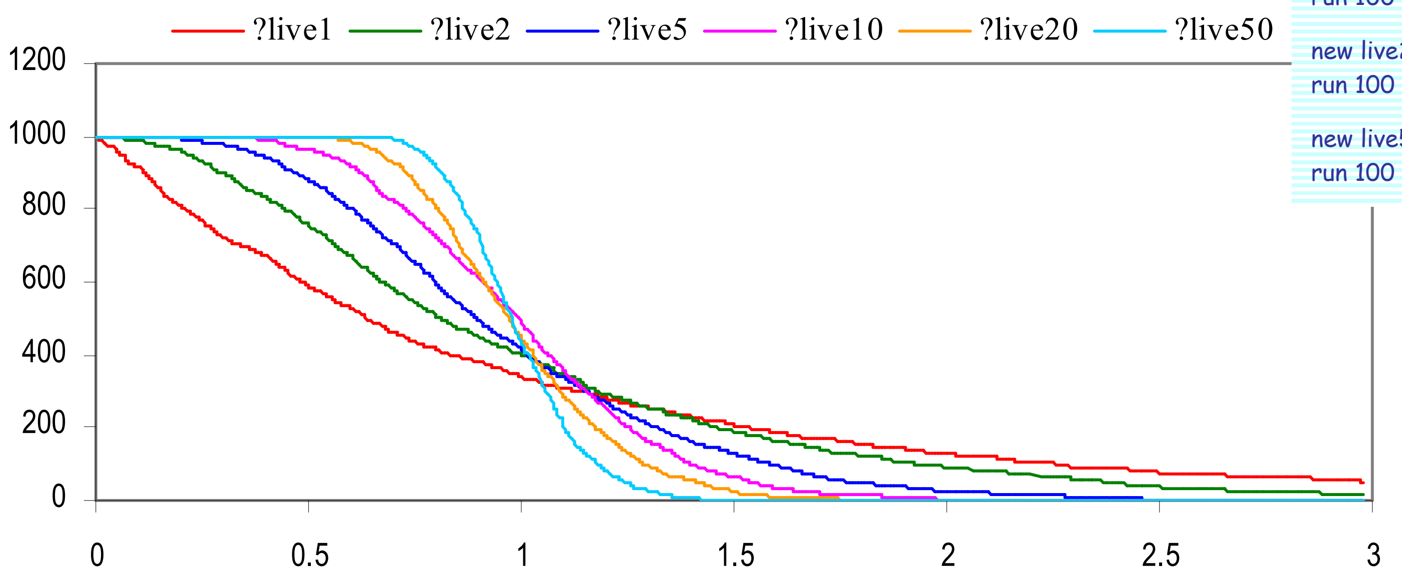




```
directive sample 3.0 1000
directive plot ?dead1; ?dead2; ?dead5;
?dead10; ?dead20; ?dead50

let s0(n:float, m:float, dead:chan()) =
  if n<=0.0 then ?dead
  else delay@m; s0(n-1.0, m, dead)

let s(n:float, dead:chan()) = s0(n,n,dead)

new dead1@1.0:chan()
run 100 of s(1.0,dead1)

new dead2@1.0:chan()
run 100 of s(2.0,dead2)

new dead5@1.0:chan()
run 100 of s(5.0,dead5)

new dead10@1.0:chan()
run 100 of s(10.0,dead10)

new dead20@1.0:chan()
run 100 of s(20.0,dead20)

new dead50@1.0:chan()
run 100 of s(50.0,dead50)
```

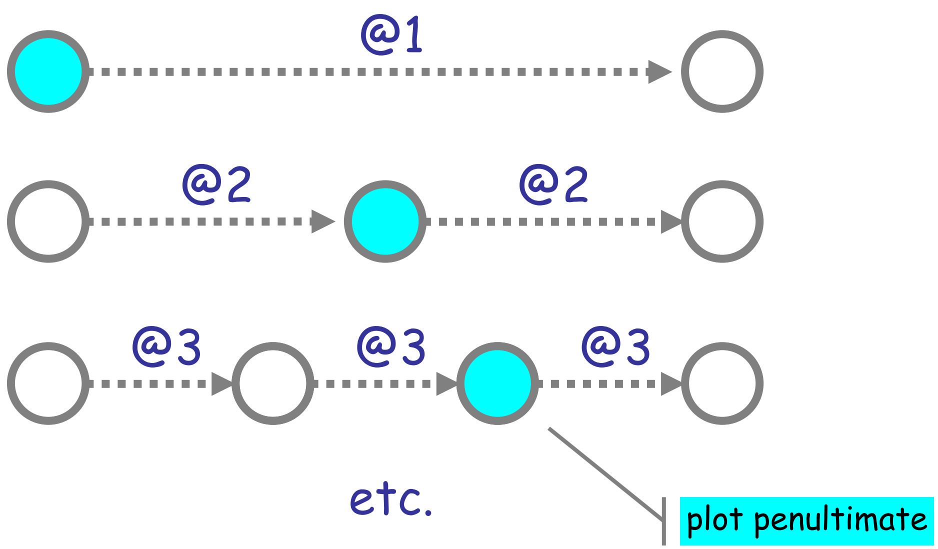
# Erlang Down-Transition



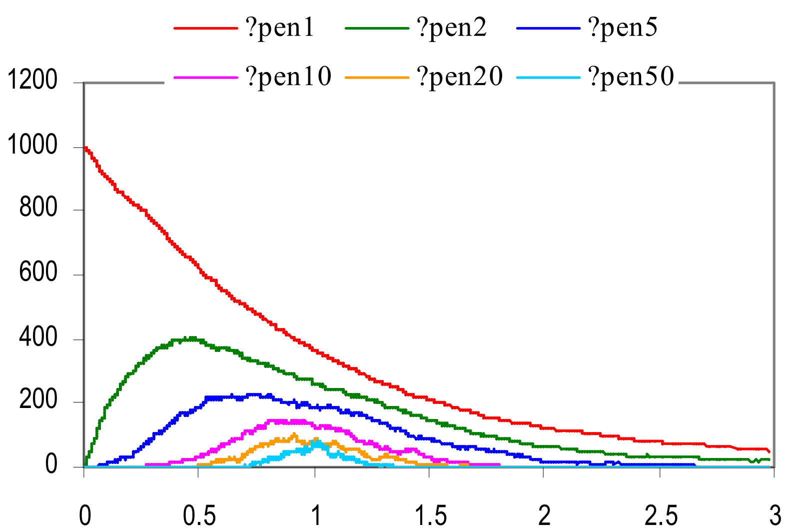

```
directive sample 3.0 1000
directive plot ?live1; ?live2; ?live5;
   ?live10; ?live20; ?live50

let s0(n:float, m:float, live:chan()) =
   if n<=0.0 then ()
   else do ?live or delay@m; s0(n-1.0, m, live)

let s(n:float, live:chan()) = s0(n,n,live)

new live1@1.0:chan()
run 100 of s(1.0,live1)

new live2@1.0:chan()
run 100 of s(2.0,live2)

new live5@1.0:chan()
run 100 of s(5.0,live5)

new live10@1.0:chan()
run 100 of s(10.0,live10)

new live20@1.0:chan()
run 100 of s(20.0,live20)

new live50@1.0:chan()
run 100 of s(50.0,live50)
```

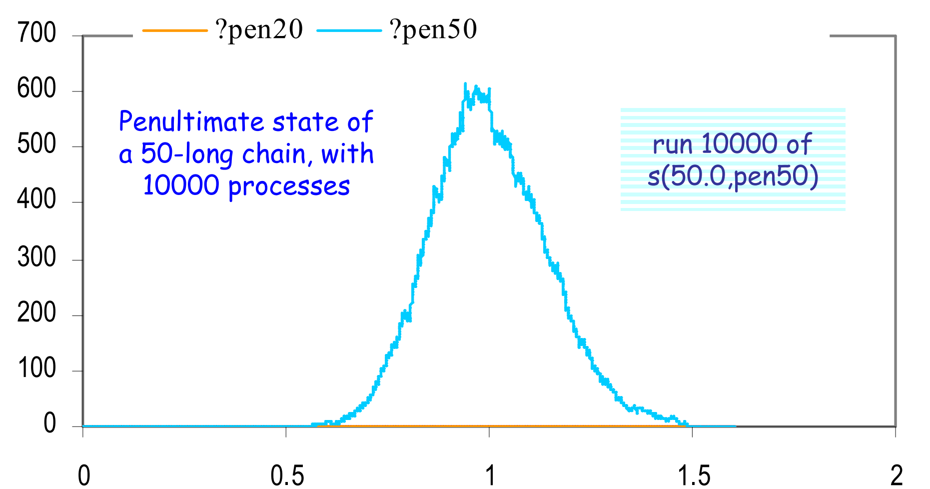


# Erlang Pulse




```
directive sample 3.0 10000
directive plot ?pen1; ?pen2; ?pen5; ?pen10; ?pen20; ?pen50

let s0(n:float, m:float, pen:chan()) =
  if n<=0.0 then ()
  else if n<=1.0 then do ?pen or delay@m; s0(n-1.0, m, pen)
  else delay@m; s0(n-1.0, m, pen)

let s(n:float, pen:chan()) = s0(n,n,pen)

new pen1@1.0:chan()
run 100 of s(1.0,pen1)

new pen2@1.0:chan()
run 100 of s(2.0,pen2)

new pen5@1.0:chan()
run 100 of s(4.0,pen5)

new pen10@1.0:chan()
run 100 of s(10.0,pen10)

new pen20@1.0:chan()
run 100 of s(20.0,pen20)

new pen50@1.0:chan()
run 100 of s(50.0,pen50)
```

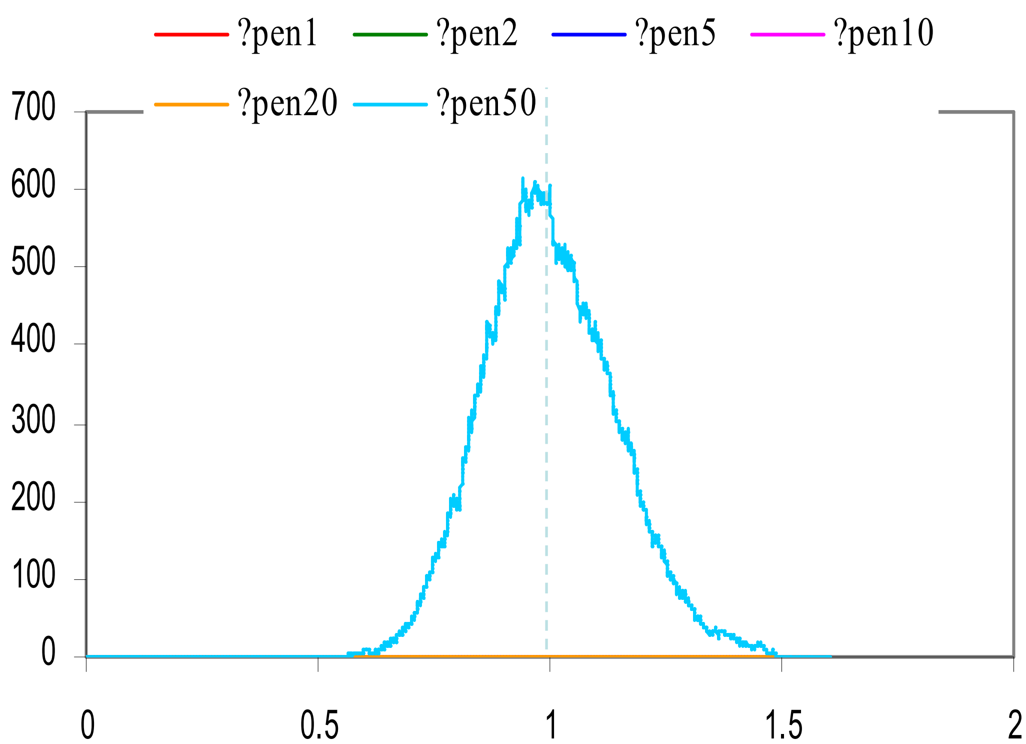


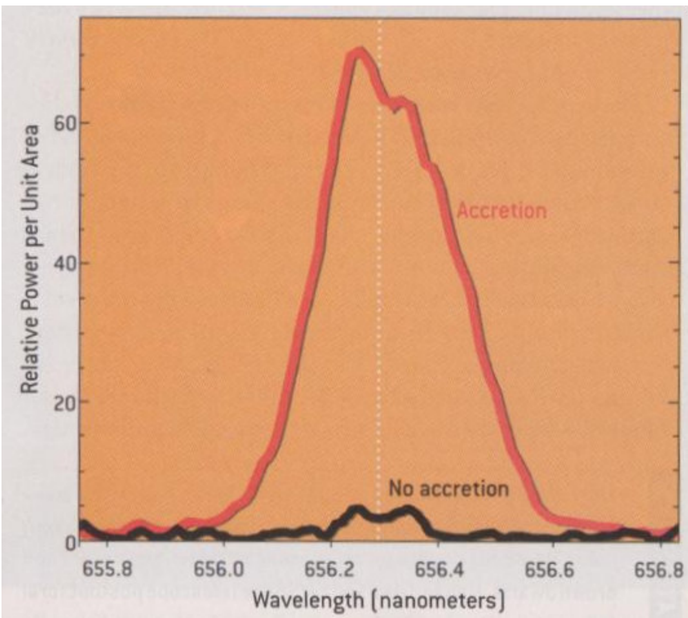



# …the next day I was reading Scientific American…

Number of processes (out of 10000) over time in the penultimate state of a 50-long chain of states

Spectral line of hydrogen in a brown dwarf with accretion disk




SPECTRAL LINE OF HYDROGEN can reveal whether a brown dwarf has a gas disk. Hydrogen atoms at rest emit light at distinct wavelengths (*dotted line*), but when a gas is moving, this light gets smeared out into a range of wavelengths reflecting the range of velocities within the gas. Gas on the dwarf surface, being comparatively slow moving, generates a narrow spectral bump (*lower curve*). A broad hump (*upper curve*) is a telltale sign of gas plummeting in from a disk. Most young brown dwarfs appear to have disks, suggesting they form in much the same way full-fledged stars do.

30 SCIENTIFIC AMERICAN JANUARY 2006

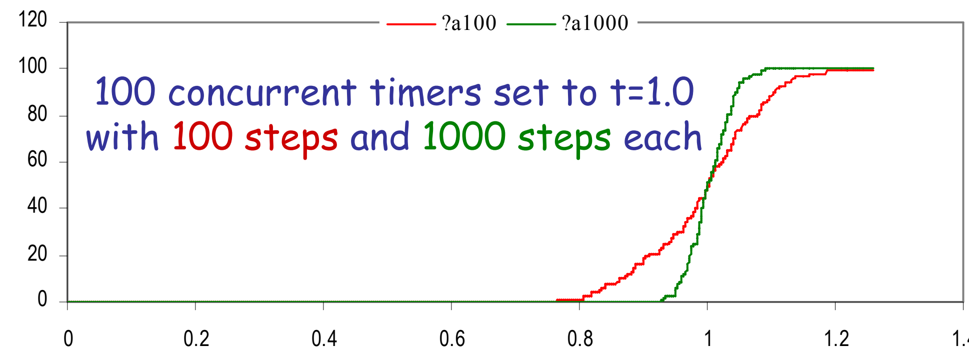
Well, ok, it's just a routine gamma distribution, which is the continuous version of an Erlang distribution. But look at all the matching stochastic bumps!

# Erlang Timers

An Erlang Timer timer(t,s,r) “rings” r (by !r) at time t, with a “precision” of s steps (each with mean lifetime t/s).

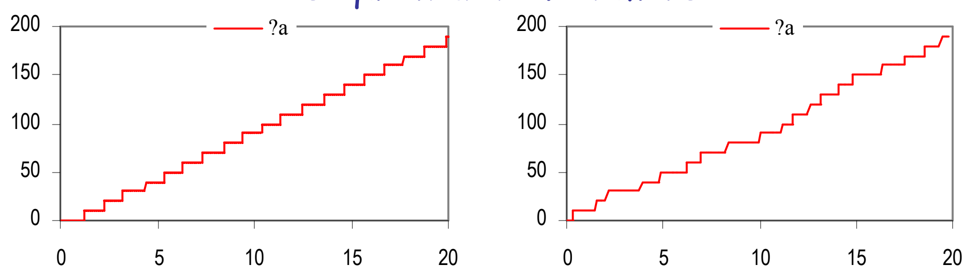
```
directive sample 2.0 10000
directive plot ?a100; ?a1000

let timer(time:float, steps:float, ring:chan) =
  (val ti = time/steps   (* break expected time into steps *)
   val del = 1.0/ti       (* rate for step (inv. of mean lifetime) *)
   let step(n:float) = if n<=0.0 then !ring else delay@del; step(n-1.0)
   run step(steps))

new s100:chan new a100@1.0:chan
new s1000:chan new a1000@1.0:chan
run 100 of (timer(1.0, 100.0, s100) | ?s100; ?a100)
run 100 of (timer(1.0, 1000.0, s1000) | ?s1000; ?a1000)
```

Expected value $E(Y) = E(X_1) + \ldots + E(X_n)$




This reringer keeps invoking a timer, each time producing 10 of ?a.

sequential t=1.0 timers




with 100 steps each    with 10 steps each

more steps within each interval gives
more precise timing

```
directive sample 20.0 10000
directive plot ?a

let timer(time:float, steps:float, ring:chan) =
  (val ti = time/steps   (* break expected time into steps *)
   val del = 1.0/ti       (* rate for step (inv. of mean lifetime) *)
   let step(n:float) = if n<=0.0 then !ring else delay@del; step(n-1.0)
   run step(steps))

new s:chan new a@1.0:chan

let rering() =
  (timer(1.0, 100.0, s) | ?s; (rering() | 10 of ?a))

run rering()
```

# Erlang Clocks and Signal Shaping

```
directive sample 100.0 10000
directive plot !a; !b

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))

new a@1.0:chan  new b@1.0:chan
let A(tick:chan) = do !a; A(tick) or ?tick; B(tick)
(* Offers !a,  as many as needed, until the next tick *)
and B(tick:chan) = do !b; B(tick) or ?tick; A(tick)

run 10 of (new tick:chan run (clock(10.0, tick) | A(tick)))

(* each signal with its own "new" tick (an infinite speed channel)*)
```

An Erlang Clock clock(t,r) is a repeating Erlang Timer; it signals !r every t.

The signal A(t) offers !a <u>as often as needed</u>, but only until a timeout t (provided by a concurrently running clock)

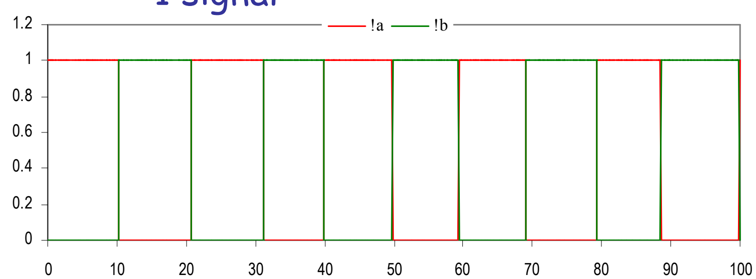
Then A(t) becomes B(t) until the next tick, and then it goes back to A(t)...

Each signal has its own private clock (new tick), or things get confused.

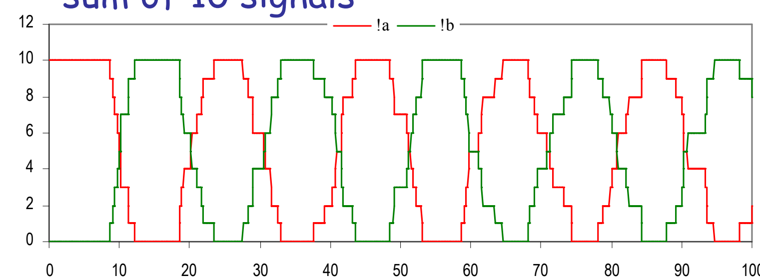
Signal A repeatedly offers !a until the next “tick”, then repeatedly offers !b until the next tick, and so on.

multiple clocks eventually get out of phase

1 signal



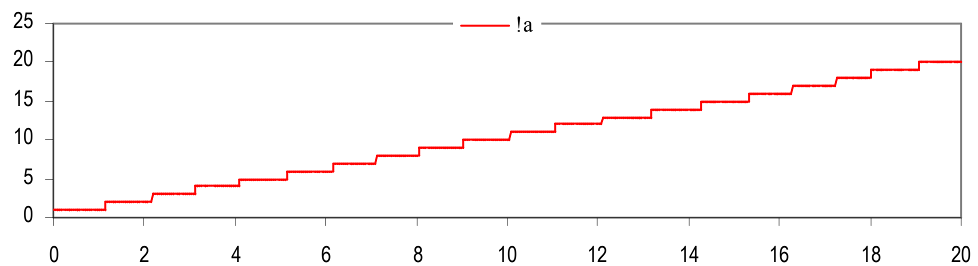

sum of 10 signals



# Erlang-Clocked Raising Signal

```
directive sample 20.0 10000
directive plot !a

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))

let S(a:chan, tick:chan) =
  do !a; S(a,tick) or ?tick; (S(a,tick) | S(a,tick))
(* Offers !a, as many as needed, until the next tick,
    then spawns one additional such signal. *)

let raising(a:chan, t:float) =
  (new tick:chan run (clock(t,tick) | S(a,tick)))
(* Encapsulating a clock with a raising signal *)

new a@1.0:chan
run raising(a,1.0)
```

An raising signal S(a,t) offers !a's until the next Erlang tick t, then it spawns off one more copy of itself. Since all the copies *share the same clock*, they increase by 1 each tick (linearly).

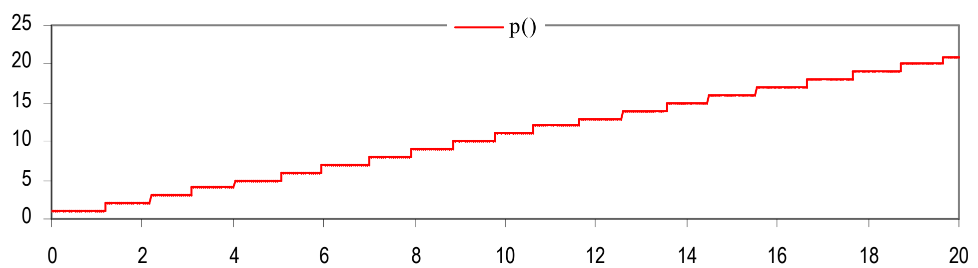


# Erlang-Clocked Raising Concentration

```
directive sample 20.0 10000
directive plot p()

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))

let S(p:proc(), tick:chan) =
  (p() | ?tick; S(p,tick))
(* Spawns a process p() every tick. *)

let raising(p:proc(), t:float) =
  (new tick:chan run (clock(t,tick) | S(p,tick)))
(* Encapsulating a clock with a raising concentration *)

new a@1.0:chan
let p() = !a
run raising(p,1.0)
```

A variation S(p,t) that spawns an arbitrary (parameterless) process p at every tick t.

An example of higher-order processes.



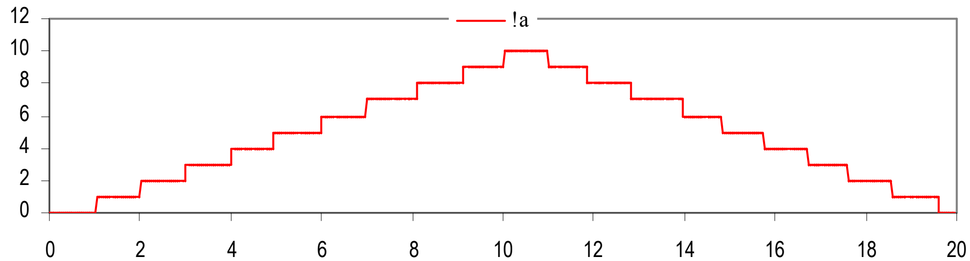
# Erlang-Clocked Raising and Falling

```
directive sample 20.0 10000
directive plot !a

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))

let S1(a:chan, tock:chan) =  do !a; S1(a,tock) or ?tock; ()
(* Offers !a, as many as needed, until the next tock. *)

let SN(n:int, t:float, a:chan, tick:chan, tock:chan) =
  if n=0 then clock(t, tock) else ?tick; (S1(a,tock) | SN(n-1,t,a,tick,tock))
(* For n ticks, starts an S1.
   At the end, starts a tock-clock to stop one S1 at each tock. *)

let raisingfalling(a:chan, n:int, t:float) =
  (new tick:chan new tock:chan
   run (clock(t,tick) | SN(n,t,a,tick,tock)))
(* Encapsulating a clock with a raising and falling signal *)

new a@1.0:chan
run raisingfalling(a,10,1.0)
```

This is a "test signal" that we will use a lot. raisingfalling(a,n,t) produces a linearly increasing !a signal with n steps of length t; then it decreases back to 0 in similar steps.

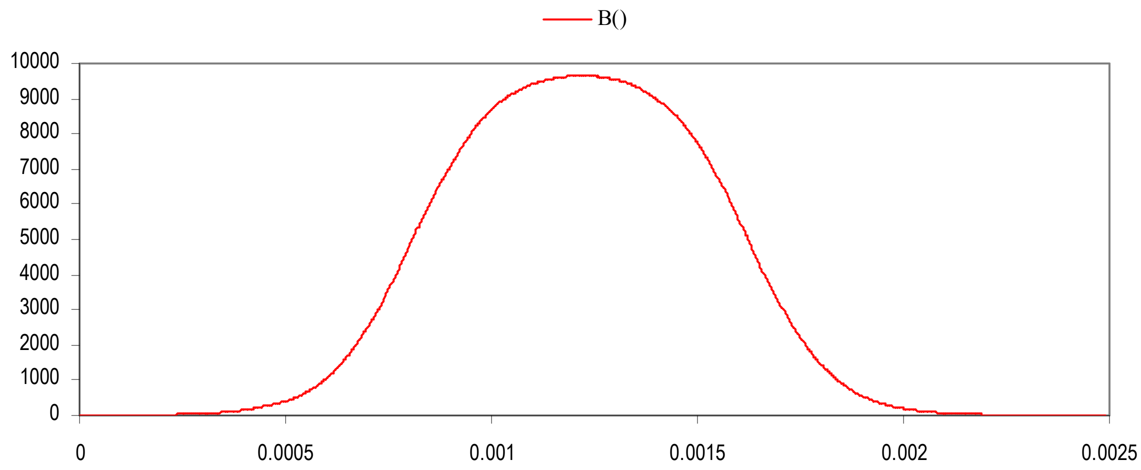
S1(a,tock) offers !a until the first tock.

SN (which is tick-clocked) starts an S1 for n ticks, then it starts a tock-clock that will stop them all in turn.



# Exercise (hard): Bell

Build a *small* network where one node has a distribution like this:




(The solution plotted here has 3 nodes and 2 channels; it uses communication.)

# Further Reading: Phase-Type Distributions

- Erlang-like distributions are "universal":

Queueing Theory

Ivo Adan and Jacques Resing

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

February 14, 2001

http://mia.ece.uic.edu/~papers/WWW/Flexi-Tunes/tarballs/queue.pdf

We mention two important classes of phase-type distributions which are *dense in the class of all non-negative distribution functions*. This is meant in the sense that for any non-negative distribution function $F(\cdot)$ a sequence of phase-type distributions can be found which pointwise converges at the points of continuity of $F(\cdot)$. The denseness of the two classes makes them very useful as a practical modelling tool. A proof of the denseness can be found in [5, 6]. The first class is the class of *Coxian distributions*, notation $C_k$, and the other class consists of *mixtures of Erlang distributions with the same scale parameters*. The phase representations of these two classes are shown in the figures 4 and 5.

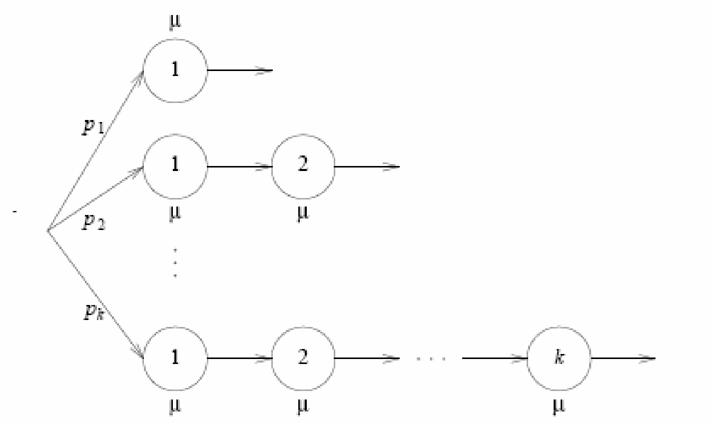



Figure 5: Phase diagram for the mixed Erlang distribution

- Possible connection between process calculi and data fitting:
  - The EM (Expectation-Maximization) algorithm fits data to general phase-type distributions.

Efficient fitting of long-tailed data sets into hyperexponential distributions

Alma Riska Vesselin Diev Evgenia Smirni
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
e-mail {riska,vdiev,esmirni}@cs.wm.edu

http://citeseer.ifi.unizh.ch/cache/papers/cs/29643/http:zSzzSzwww.cs.wm.eduzSz~esmirnizSzdocszSzglobecom02.pdf/ecient-fitting-of-long.pdf

# SPiM Basic Syntax

| | | | |
|---|---|---|---|
| *Program* | ::= | {**directive sample** *Float* {*Integer*}} | Sample Directive |
| | | {**directive plot** $Point_1 \ldots Point_N$} | Plot Directive |
| | | $Declaration_1 \ldots Declaration_N$ | Declarations, $N \geq 1$ |
| *Declaration* | ::= | **new** *Name*{@*Rate*}:*Type* | Channel Declaration |
| | \| | **type** *Name* = *Type* | Type Declaration |
| | \| | **val** *Pattern* = *Value* | Value Declaration |
| | \| | **run** *Process* | Process Declaration |
| | \| | **let** $Definition_1$ **and** … **and** $Definition_N$ | Definitions, $N \geq 1$ |
| *Definition* | ::= | *Name* ($Pattern_1$, … ,$Pattern_N$) = *Process* | Definition, $N \geq 0$ |
| *Process* | ::= | () | Null Process |
| | \| | ($Process_1$ \| … \| $Process_M$) | Parallel, $M \geq 2$ |
| | \| | *Name* ($Value_1$, … ,$Value_N$){; *Process*} | Instantiation, $N \geq 0$ |
| | \| | *ActionProcess* | Action Process |
| | \| | **do** $ActionProcess_1$ **or** … **or** $ActionProcess_M$ | Choice, $M \geq 2$ |
| | \| | **replicate** *ActionProcess* | Replicated Action |
| | \| | **if** *Value* **then** *Process* {**else** *Process*} | Conditional Process |
| | \| | **match** *Value* **case** $Case_1 \ldots$ **case** $Case_N$ | Matching, $N \geq 1$ |
| | \| | *Integer* **of** *Process* | Repetition |
| | \| | ($Declaration_1 \ldots Declaration_N$ *Process*) | Declarations, $N \geq 0$ |
| *ActionProcess* | ::= | *Action*{; *Process*} | Action Process |
| *Action* | ::= | !*Channel* {($Value_1$, … ,$Value_N$)} | Output, $N \geq 0$ |
| | \| | ?*Channel* {($Pattern_1$, … ,$Pattern_N$)} | Input, $N \geq 0$ |
| | \| | `delay`@*Rate* | Delay |
| *Channel* | ::= | *Name* | |
| *Rate* | ::= | *Float* | |
| | \| | *Name* | |

# Summary

- Exponential Distributions
  - Simplest (memoryless) distributions
  - The only memoryless distributions
  - Fully general when networked
- Erlang Distributions
  - Useful for building clocks and other signal shapes when all you got are exponential distributions
- SPiM
  - A language for (among other things) programming with exponential distribution

# Q?