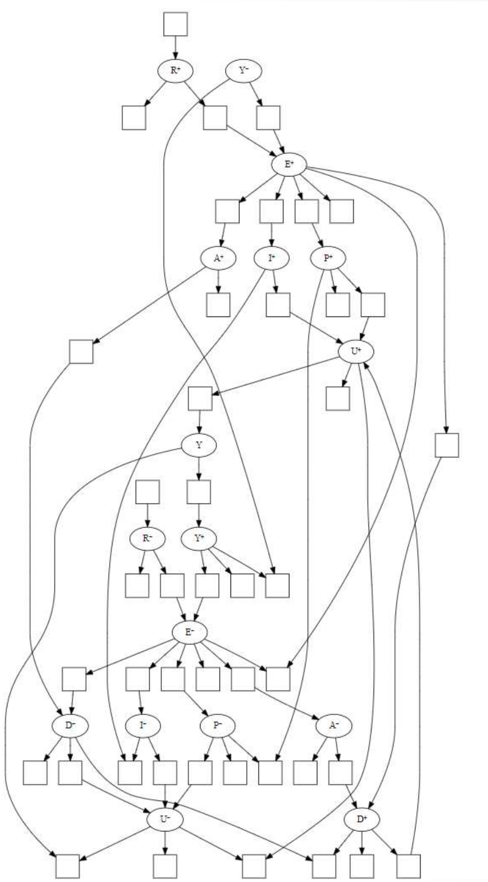
Programming Chemical Reaction Networks in Kaemika



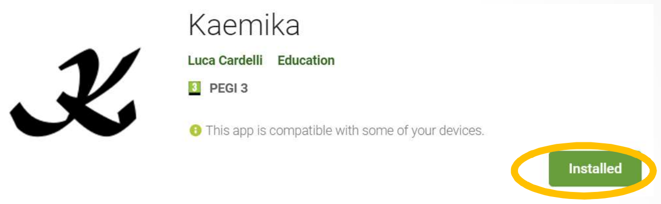
Luca Cardelli, University of Oxford

Future of Computing

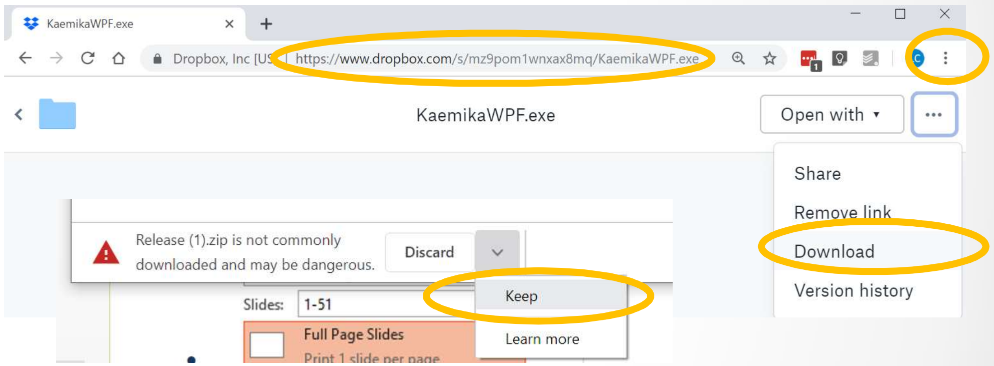
2019-07-04 Porto

Install Kaemika

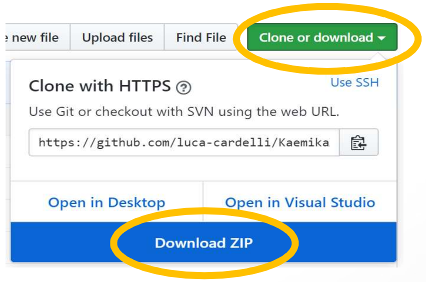
- Android version:
  - Search "Kaemika" in the Play Store
  - https://play.google.com/store/apps/details?id=com.kaemika.Kaemika



- Windows version (Dropbox): (runs without installation)
  - Download & Unzip: https://www.dropbox.com/s/qxity2e9hw4fw5c/Release.zip
  - Run Release\KaemikaWPF.exe



- Windows version (Github, probably need account):
  - https://github.com/luca-cardelli/KaemikaXM
  - Download & Unzip
  - Run ...\KaemikaXM-master\KaemikaWPF\bin\Release\KaemikaWPF.exe



Infinite Loop #1

- The first ever interesting "chemical algorithm" that had nothing to do with actual chemicals

UNDAMPED OSCILLATIONS DERIVED FROM THE LAW OF MASS ACTION.

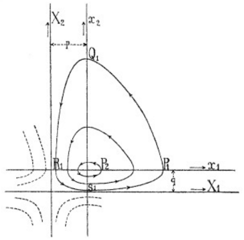
By Alfred J. Lotka.

Received June 2, 1920.

$$\frac{dX_1}{dt} = a_1X_1 - b_1X_1X_2$$

$$\frac{dX_2}{dt} = a_2X_1X_2 - b_2X_2.$$

The first single, homogeneous oscillating chemical reaction was discovered accidentally by Bray [6] in 1921.

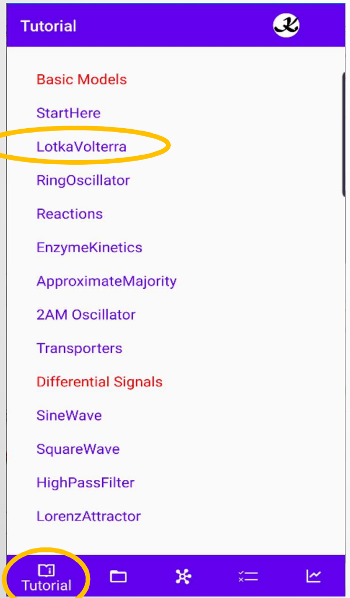


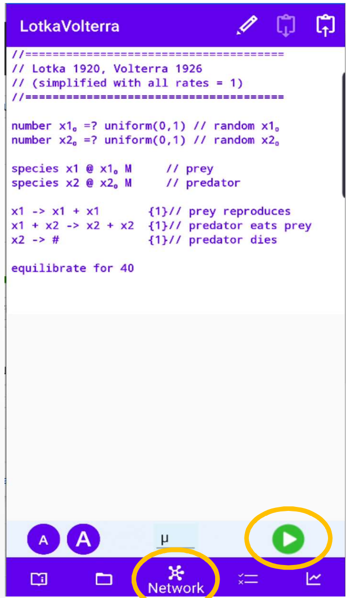
```
x1 -> x1 + x1        {1} // prey reproduces
x1 + x2 -> x2 + x2   {1} // predator eats prey
x2 -> #              {1} // predator dies
```

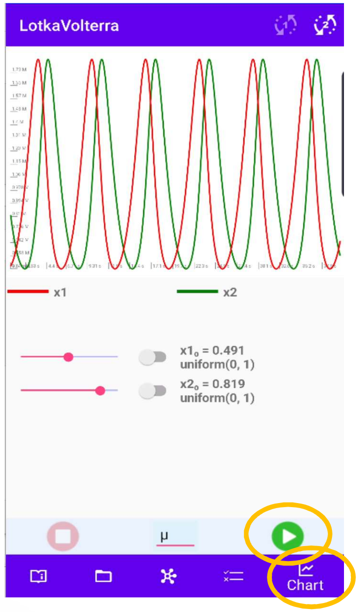
- First theoretical proof of oscillation, 1920 [Lotka]
- First experimental (accidental) chemical oscillator, 1921 [Bray]
- Ignored until the BZ reaction (accidental) discovery, 1958 [Belousov–Zhabotinsky]
- First non-accidentally-discovered chemical oscillator, 1981 [De Kepper]
- First protein/ATP-only oscillator, 2005 [Nakajima et al.]
- First DNA-only oscillator, 2017 [Srinivas et al.] (a version of Lotka's)

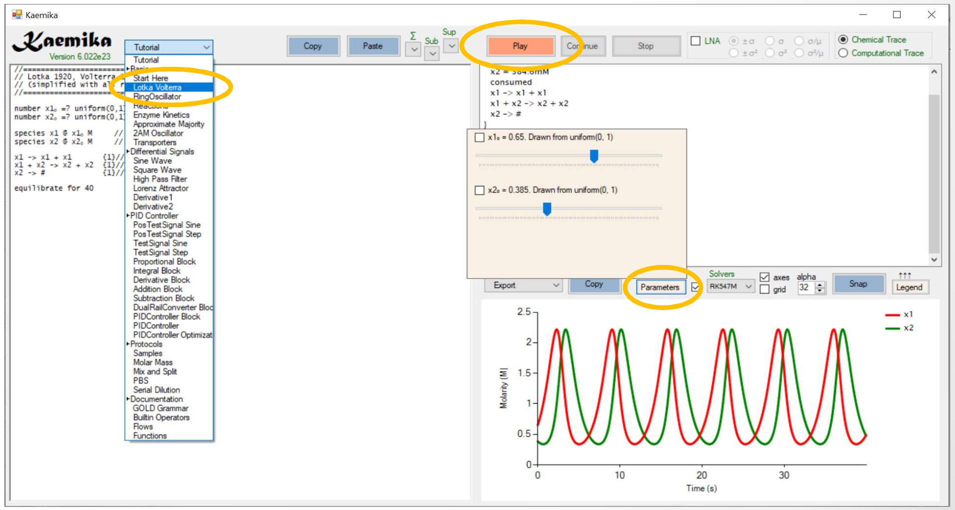
Lotka Volterra

- Try removing the prey (x1) (set prey reproduction rate, reaction #1, to 0)
- Try removing the predators (x2) (set predation rate, reaction #2, to 0)
- Try making predators immortal (set predator dieout rate, reaction #3, to 0)
- Try doubling the predation rate (prey go down, but predators too!)
- Try doubling the prey reproduction rate (prey go up, but predators go up more!)

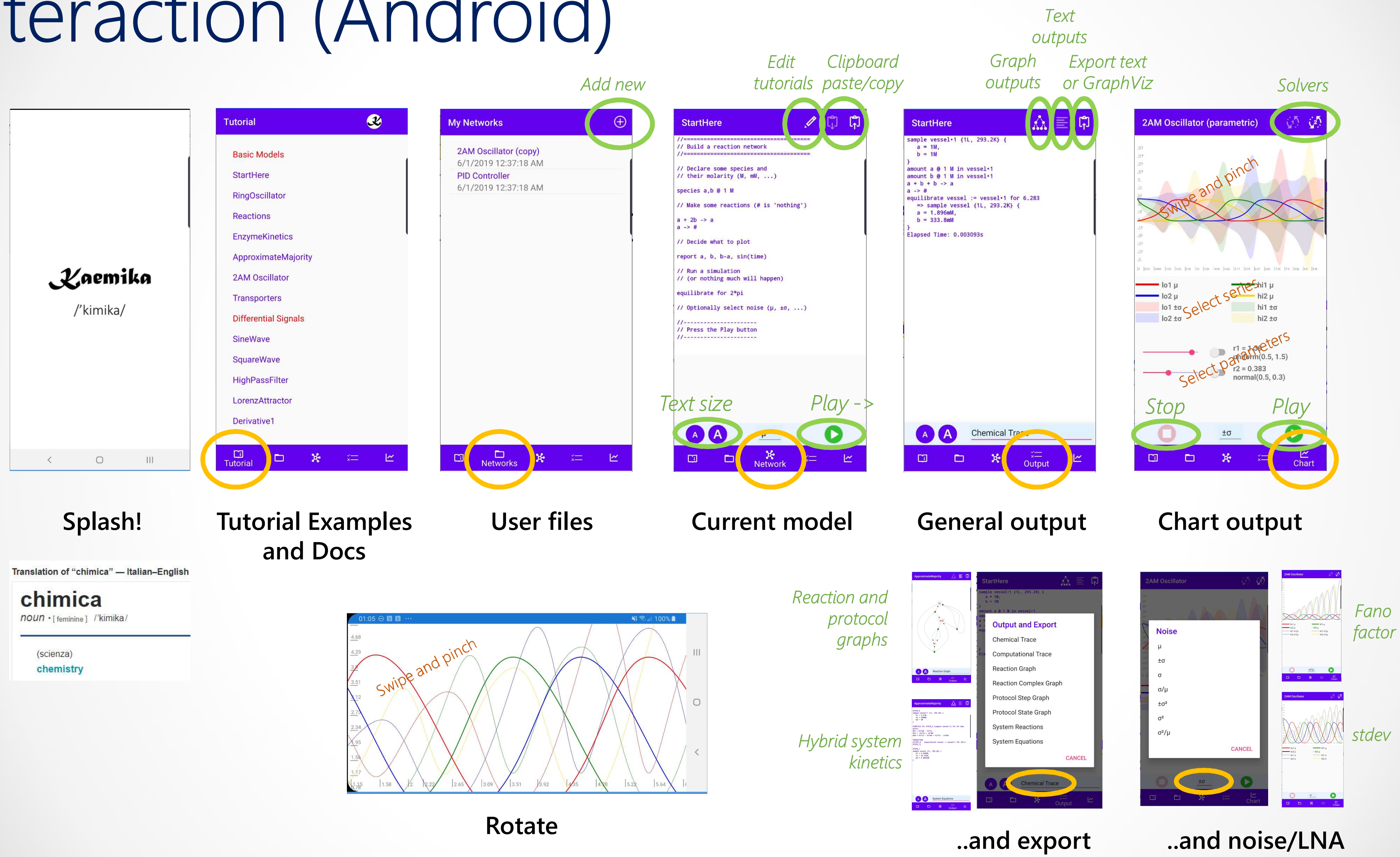




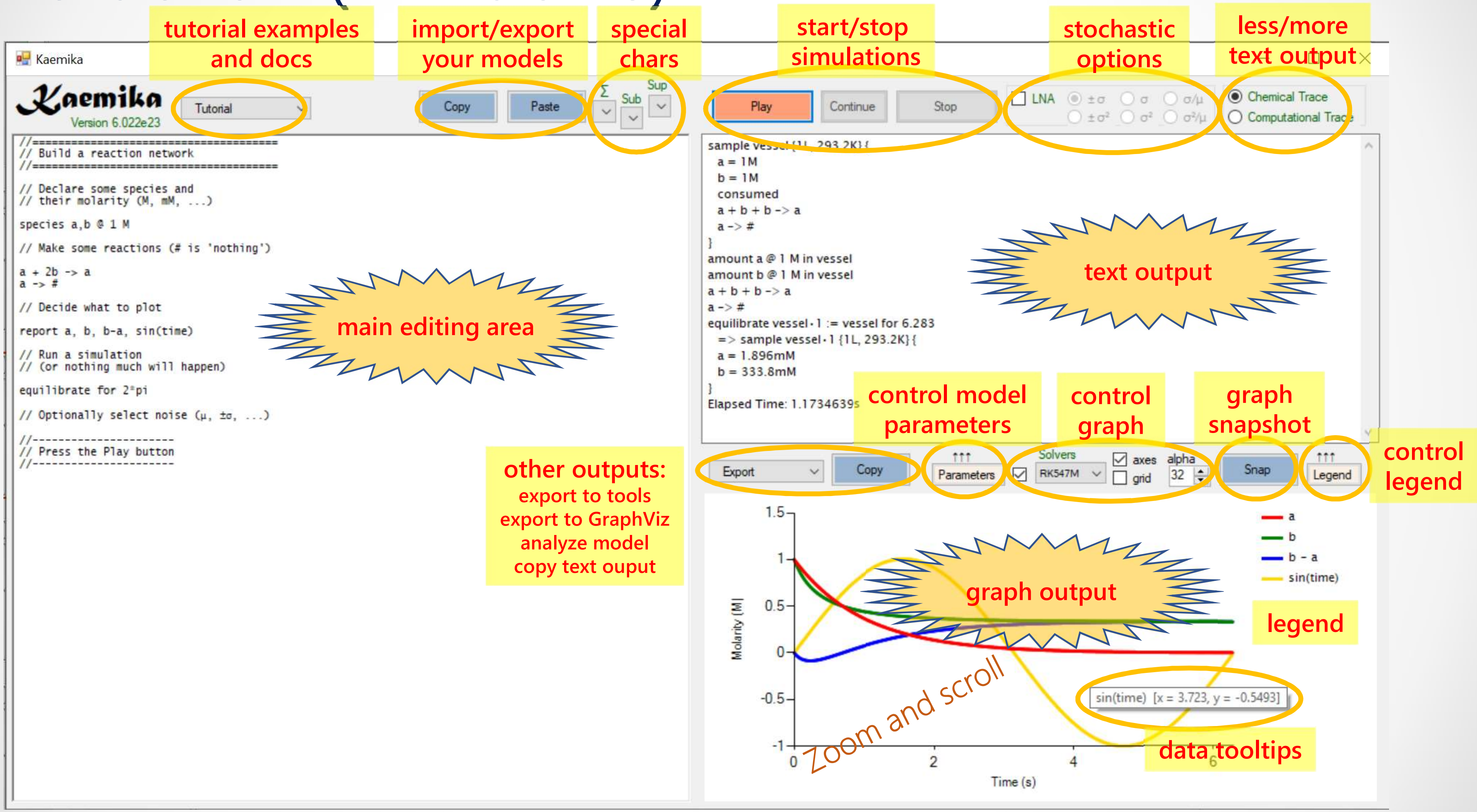




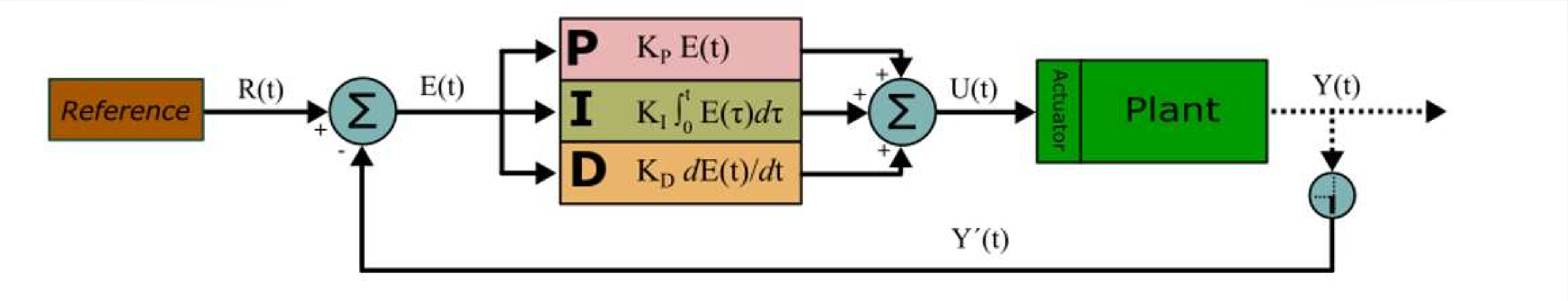
Interaction (Android)

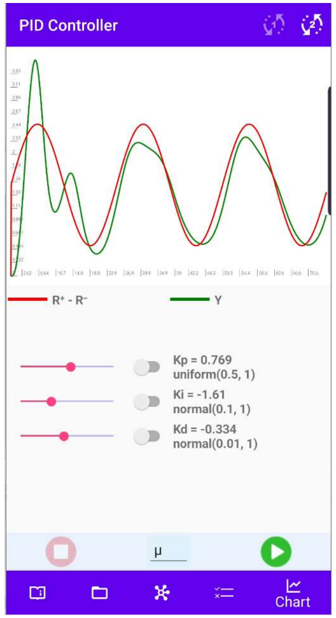


Interaction (Windows)



PID Controller

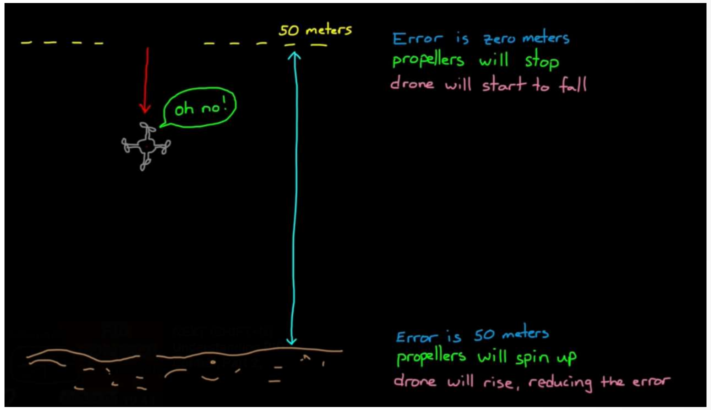




MATLAB YouTube Video

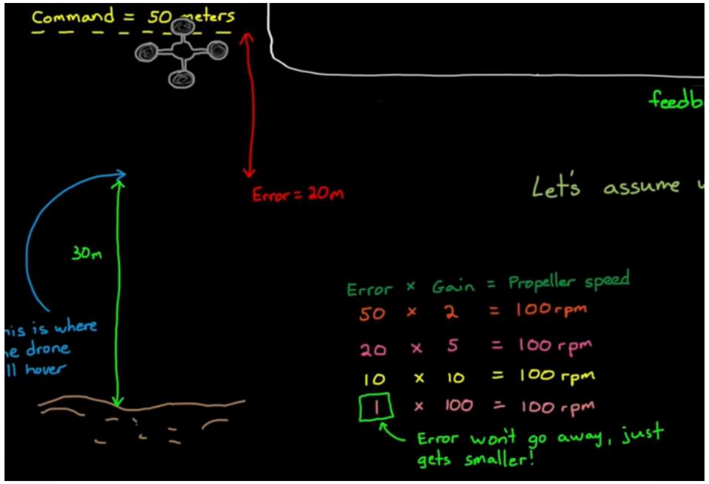
https://www.youtube.com/watch?v=wkfEZmsQqiA

On/Off Control:



Proportional Control:

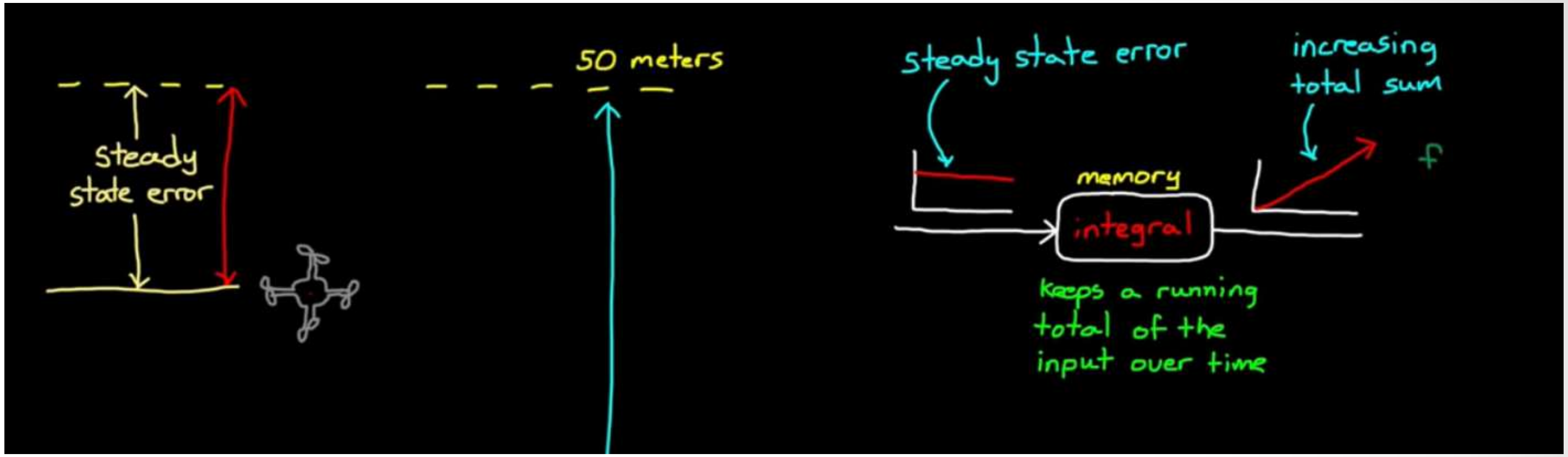
Propeller speed = Error * Gain
hovering speed => Error =/= 0



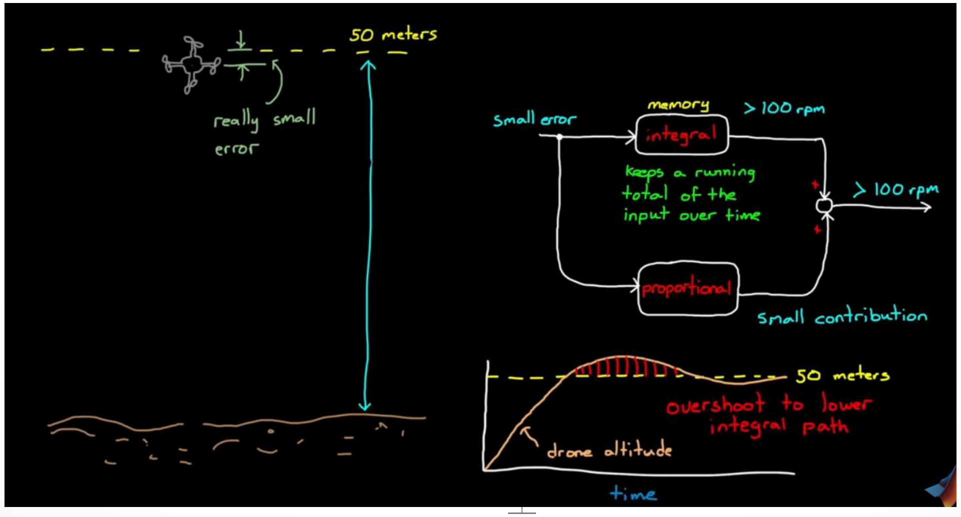
MATLAB YouTube Video

Proportional
+Integral
Control:

removes the steady state error (eventually)



but may overshoot:



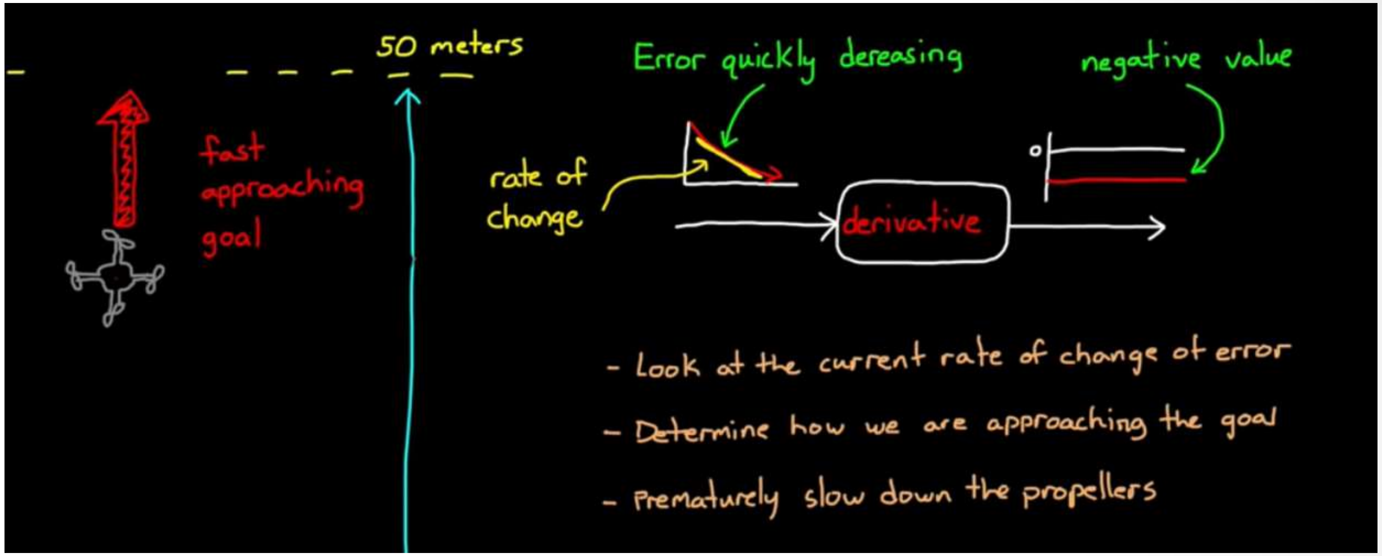
MATLAB YouTube Video

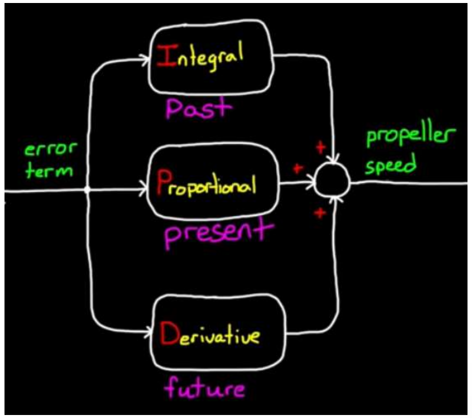
Proportional
+Integral
+Differential
Control:

if the error is *decreasing* fast
a large *negative* derivative
will be added

slowing down the ascent
and preventing overshooting

PID: present + past + future





Positive Arithmetic

```
// copy
species a @ 2 M
species a' @ 0 M
report a, a'

a -> a + a'
a' -> #

equilibrate for 5
```

```
// add
species a @ 2 M
species b @ 3 M
species c @ 0 M
report a, b, c

a -> c
b -> c

equilibrate for 5
```

```
// copy and add
species a @ 2 M
species b @ 3 M
species c @ 0 M
report a, b, c

a -> a + c
b -> b + c
c -> #

equilibrate for 5
```

∂a = 0
∂a' = a - a'

∂a'= 0 => a' = a

Try increasing the a level
How much longer will it take to reach steady state?

∂a = 0
∂b = 0
∂c = a + b - c

∂c = 0 => c = a + b

Positive Arithmetic

```
//mult
species a @ 2 M
species b @ 3 M
species c @ 0 M
report a, b, c

a + b -> a + b + c
c -> #

equilibrate for 5
```

```
//div
species a @ 2 M
species b @ 3 M
species c @ 0 M
report a, b, c

a -> a + c
b + c -> b

equilibrate for 5
```

∂a = 0
∂b = 0
∂c = a * b - c

∂c = 0 => c = a * b

∂a = 0
∂b = 0
∂c = a - b * c

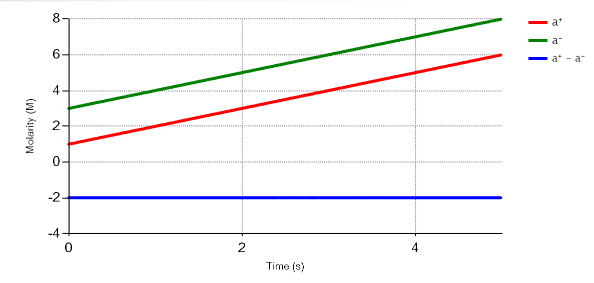
∂c = 0 => c = a / b
(b = 0 => c -> ∞)

Differential Signals

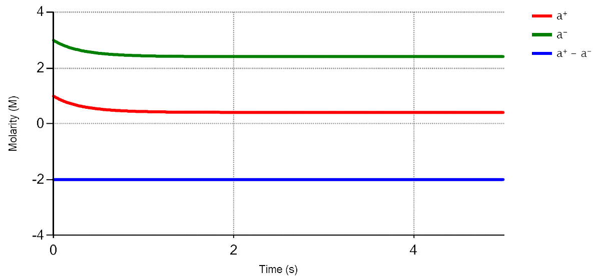
But ERRORS in the PID controller
can be positive or negative,
while concentrations can only be positive.

Solution: encode an integer number
as the difference of two
natural numbers (concentations)

```
// a = a⁺ - a⁻

species a⁺ @ 1 M
species a⁻ @ 3 M
report a⁺, a⁻
report a⁺ - a⁻

# -> a⁺
# -> a⁻

// normalization:
// a⁺ + a⁻ -> #

equilibrate for 5
```



Without normalization
(they keep growing)



With normalization
(we are still producing **a⁻**, so it does not go to zero, but it stabilizes)

Addition of Differential Signals

$$
\begin{aligned}
a + b &= (a^+ - a^-) + (b^+ - b^-) \\
&= (a^+ + b^+) - (a^- + b^-) \\
&= c^+ - c^- \\
&= c
\end{aligned}
$$

```
species a⁺ @ 2 M
species a⁻ @ 0 M

species b⁺ @ 0 M
species b⁻ @ 3 M

species c⁺ @ 0 M
species c⁻ @ 0 M

report a⁺ - a⁻, b⁺ - b⁻, c⁺ - c⁻

a⁺ -> c⁺
b⁺ -> c⁺

a⁻ -> c⁻
b⁻ -> c⁻

equilibrate for 5
```

Subtraction of Differential Signals

$$
\begin{aligned}
a - b &= (a^+ - a^-) - (b^+ - b^-) \\
&= (a^+ + b^-) - (a^- + b^+) \\
&= c^+ - c^- \\
&= c
\end{aligned}
$$

```
species a⁺ @ 2 M
species a⁻ @ 0 M

species b⁺ @ 0 M
species b⁻ @ 3 M

species c⁺ @ 0 M
species c⁻ @ 0 M

report a⁺ - a⁻, b⁺ - b⁻, c⁺ - c⁻

a⁺ -> c⁺
b⁻ -> c⁺

a⁻ -> c⁻
b⁺ -> c⁻

equilibrate for 5
```

Multiplication of Differential Signals

```
a * b = (a⁺ - a⁻) * (b⁺ - b⁻)
      = a⁺*b⁺ - a⁺*b⁻ - a⁻*b⁺ + a⁻*b⁻
      = (a⁺*b⁺ + a⁻*b⁻) - (a⁺*b⁻ + a⁻*b⁺)
      = c⁺ - c⁻
      = c
```

At this point we would want to use some "subroutines", since se have already seen how to multiply and add positive quantities and we need to do a whole bunch of those.

Fortunately we will not need this multiplication for the PID controller, but we will still need to modularize reactions.

```
species a⁺ @ 2 M
species a⁻ @ 0 M

species b⁺ @ 0 M
species b⁻ @ 3 M

species c⁺ @ 0 M
species c⁻ @ 0 M

...

equilibrate for 5
```

Modular Chemical Programs

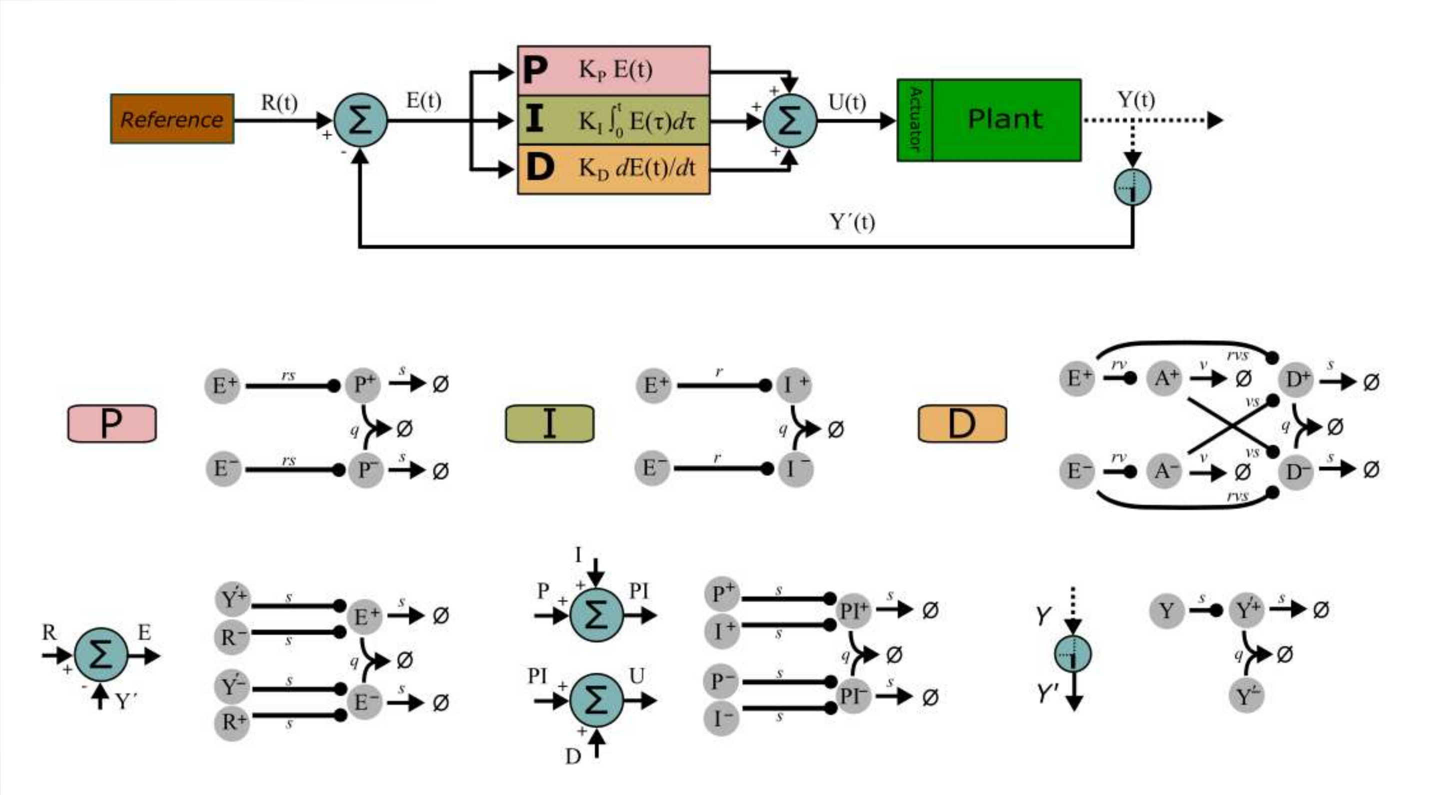
```
function signal(number n) {
  define
    species n⁺ @ pos(n) M
    species n⁻ @ pos(-n) M
    n⁺ + n⁻ -> #
  yield
    [n⁺, n⁻]
}

function copy([species a⁺ a⁻]) {
  define
    [species b⁺ b⁻] = signal(0)
    a⁺ -> a⁺ + b⁺; b⁺ -> #
    a⁻ -> a⁻ + b⁻; b⁻ -> #
  yield
    [b⁺, b⁻]
}
```

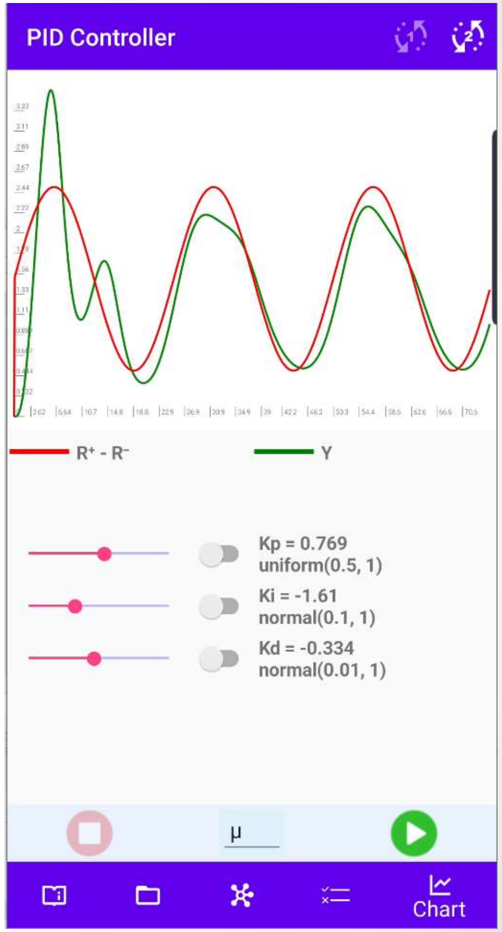
```
function add([species a⁺ a⁻], [species b⁺ b⁻]) {
  define
    [species c⁺ c⁻] = signal(0)
    a⁺ -> c⁺; b⁺ -> c⁺
    a⁻ -> c⁻; b⁻ -> c⁻
  yield
    [c⁺, c⁻]
}

function sub([species a⁺ a⁻], [species b⁺ b⁻]) {
  define
    [species c⁺ c⁻] = signal(0)
    a⁺ -> c⁺; b⁻ -> c⁺
    a⁻ -> c⁻; b⁺ -> c⁻
  yield
    [c⁺, c⁻]
}
```

Modular Chemical Programs

```
list a = signal(3)    // [a⁺, a⁻]
list b = signal(-2)   // [b⁺, b⁻]
list d = copy(a)      // [d⁺, d⁻]
list c = add(a, b)    // [c⁺, c⁻]
list e = sub(d, c)    // [e⁺, e⁻]


report a(0) - a(1) as "a"
report b(0) - b(1) as "b"
report c(0) - c(1) as "c"
report d(0) - d(1) as "d"
report e(0) - e(1) as "e"

equilibrate for 5
```

PID Controller





Proportional Block

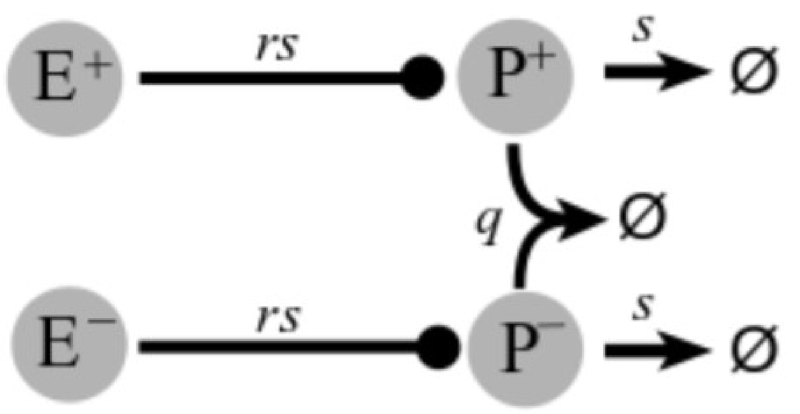
- Amplify the error **E** = (**E**$^+$ - **E**$^-$) into **P** = (**P**$^+$ - **P**$^-$) by a tunable "gain" **Kp**

```
//--------------- Proportional Block ----------------
network PBlock (species E⁺ E⁻ P⁺ P⁻, number Kp) {
        E⁺ -> E⁺ + P⁺                        {precision * Kp}
        E⁻ -> E⁻ + P⁻                        {precision * Kp}
        P⁺ -> #                              {precision}
        P⁻ -> #                              {precision}
        P⁻ + P⁺ -> #                         {precision}
}
```

You may recognize the pattern:

```
E⁺ is copied into P⁺  (reaction 1 & 3)
E⁻ is copied into P⁻  (reaction 2 & 4)
P⁺-P⁻ is normalized   (reaction 5)
```

But the "copying" adds an amplification Kp
(rates of 1 & 2)



Unit Testing the P-Block

- Change Kp
- Change the "DSignal" function

Integral Block

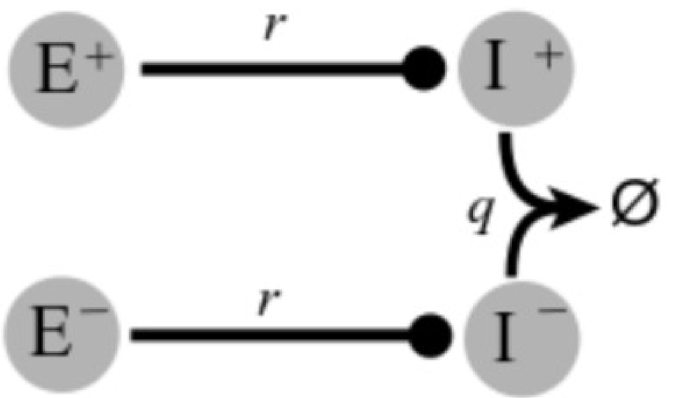
- Integrate the error **E** = (**E⁺** - **E⁻**) into **I** = (**I⁺** - **I⁻**) with a tunable "gain" **`Ki`**

```
//-------------------- Integral Block ---------------------
network IBlock (species E⁺ E⁻ I⁺ I⁻, number Ki) {
       E⁺ -> E⁺ + I⁺                 {Ki}
       E⁻ -> E⁻ + I⁻                 {Ki}
       I⁻ + I⁺ -> #                  {precision}
}
```

This is even easier:

```
E⁺ accumulates into I⁺   (reaction 1)
E⁻ accumulates into I⁻   (reaction 2)
I⁺-I⁻ is normalized      (reaction 3)
```

But the "accumulation" adds an amplification `Ki` `(rates of 1 & 2)`



Unit Testing the I-Block

- Change Ki
- Change the "DSignal" function

Derivative Block

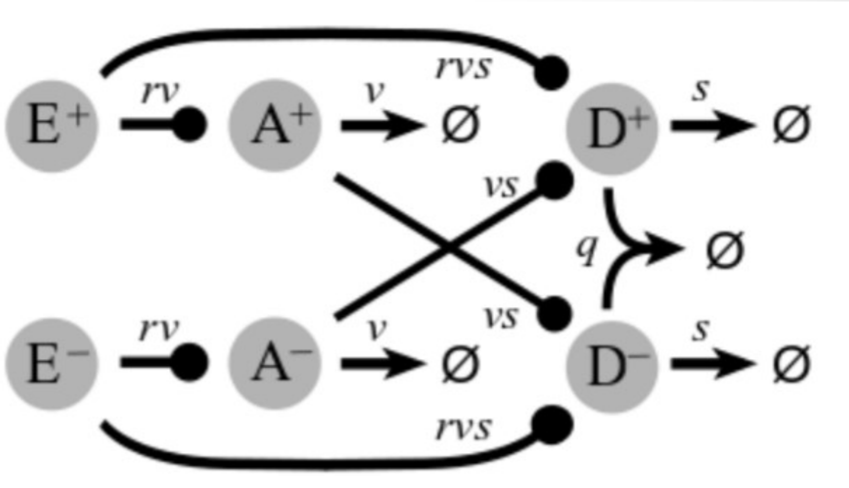
- Differentiate the error **E** = (**E**$^+$ - **E**$^-$) into **D** = (**D**$^+$ - **D**$^-$) with a tunable "gain" **Kd**

```
//------------------ Derivative Block -------------------
network DBlock(species E+ E- D+ D-, number Kd) {
        species A+, A- @ 0M  // D block auxiliary species
        E+->E+ + A+          {precision}
        A+ -># 	             {precision}
        E- ->E- + A-         {precision}
        A- -># 	             {precision}
        E+ ->E+ + D+         {precision*precision*Kd}
        A- ->A- + D+         {precision*precision*Kd}
        D+ -> #              {precision}
        E- -> E- + D-        {precision*precision*Kd}
        A+ -> A+ + D-        {precision*precision*Kd}
        D- -> #              {precision}
        D+ + D- -> #         {precision}
}
```

PID Control of Biochemical Reaction Networks

Max Whiby[1], Luca Cardelli[1], Marta Kwiatkowska[1], Luca Laurenti[1], Mirco Tribastone[2], Max Tschaikowski[3]

- E^+,E^- is copied into A^+,A^-
- E^+,E^- is also copied into D^+,D^-
- A^+,A^- is swap-copied (negated-summed) into D^+,D^-
- So D is the difference of two copies of E taken at different times t and t+s (because E->D is faster than E->A->D)
- Appropriate rates ensure that
  $$D(t) = (E(t)-E(t-s))/s$$
  which converges to the deriviative for s->0 (where s is a rate)



Derivative Block

- Informal argument

```
//------------------ Derivative Block -------------------
network DBlock(species E⁺ E⁻ D⁺ D⁻, number Kd) {
       species A⁺, A⁻ @ 0M  // D block auxiliary species
       E⁺->E⁺ + A⁺          {s}
       A⁺ -># 	           {s}
       E⁻ ->E⁻ + A⁻         {s}
       A⁻ ->#              {s}
       E⁺ ->E⁺ + D⁺         {s*s*k}
       A⁻ ->A⁻ + D⁺         {s*s*k}
       D⁺ -> #             {s}
       E⁻ -> E⁻ + D⁻        {s*s*k}
       A⁺ -> A⁺ + D⁻        {s*s*k}
       D⁻ -> #             {s}
       D⁺ + D⁻ -> #        {s}
}
```

- $\partial A^+ = sE^+ - sA^+$
- $\partial A^- = sE^- - sA^-$
- $\partial D^+ = ks^2E^+ + ks^2A^- - sD^+ - sD^+D^-$
- $\partial D^- = ks^2E^- + ks^2A^+ - sD^- - sD^+D^-$

- $\partial(D^+-D^-) = ks(sE^+-sE^-) + ks(sA^--sA^+) - s(D^+-D^-)$
- $\partial(D^+-D^-) = ks(sE^+-sA^+) - ks(sE^--sA^-) - s(D^+-D^-)$
- $\partial(D^+-D^-) = ks\partial(A^+ - A^-) - s(D^+-D^-)$

- at "steady state" (when $\partial A^+ = \partial A^- = \partial(D^+-D^-) = 0$)
- $A^+ = E^+$
- $A^- = E^-$
- $D^+ - D^- = k\partial(E^+ - E^-)$

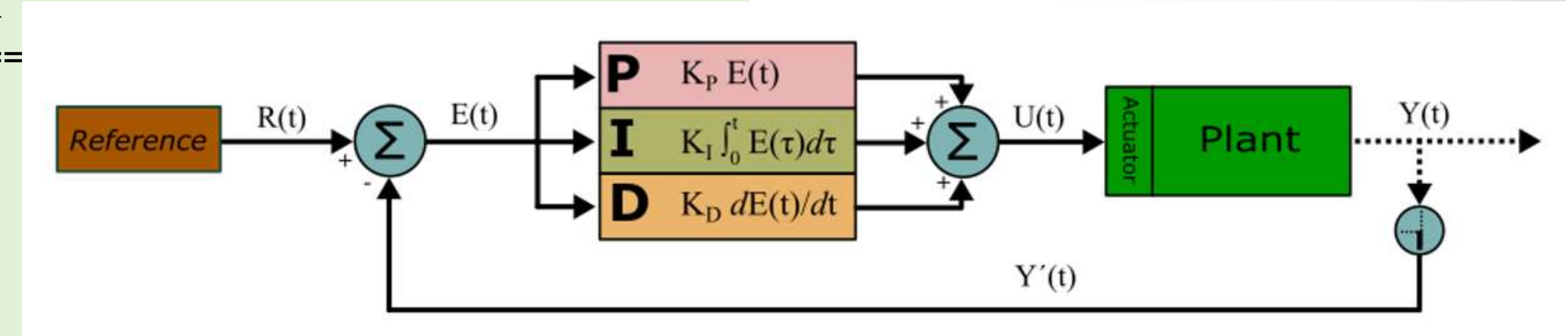
- (but $\partial(D^+-D^-)$ may never reach steady state, so this needs a more formal argument)

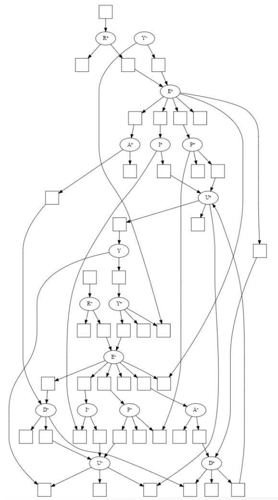
Unit Testing the D-Block

- Increase the precision (say, to 100) for better fidelity
- Change the function to differentiate
  - 2*time, whose derivative is constant 2
  - time^2, whose derivative is 2 (slope 2)
  - exp(time), whose derivative is exp(time)! (STOP it before it crashes!)
  - the *second* derivative of sin(time) by using two D-Blocks

Finally, the whole PID controller

```
//========================================
// PID Controller Block
//======================

network PIDController(
   species R⁺ R⁻,
   number Kp Ki Kd,
   network Plant){

   species E⁺,E⁻,P⁺,P⁻,I⁺,I⁻,  D⁺,D⁻,U⁺,U⁻,Y,Y⁺,Y⁻ @ 0M

   PBlock(E⁺, E⁻, P⁺, P⁻, Kp)
   IBlock(E⁺, E⁻, I⁺, I⁻, Ki)
   DBlock(E⁺, E⁻, D⁺, D⁻, Kd)
   SumBloc(P⁺, P⁻, I⁺, I⁻, D⁺, D⁻, U⁺, U⁻)
   Plant(U⁺, U⁻, Y)
   DualRail(Y, Y⁺, Y⁻)
   SubBlock(R⁺, R⁻, Y⁺, Y⁻, E⁺, E⁻)
}
```





The Trivial Plant

```
network Plant(species U⁺ U⁻ Y) {
   U⁺ -> U⁺ + Y
   U⁻ + Y -> U⁻
}
```

- U^+ increases the output Y
- U^- decreases the output Y
- However it is not symmetrical:

  $\partial Y = U^+ - Y * U^-$

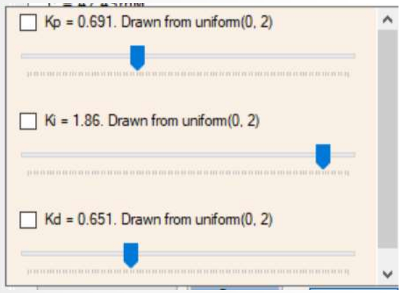
  so, although we can control the plant, we do not have direct control of Y and the control task is still mildly non-trivial

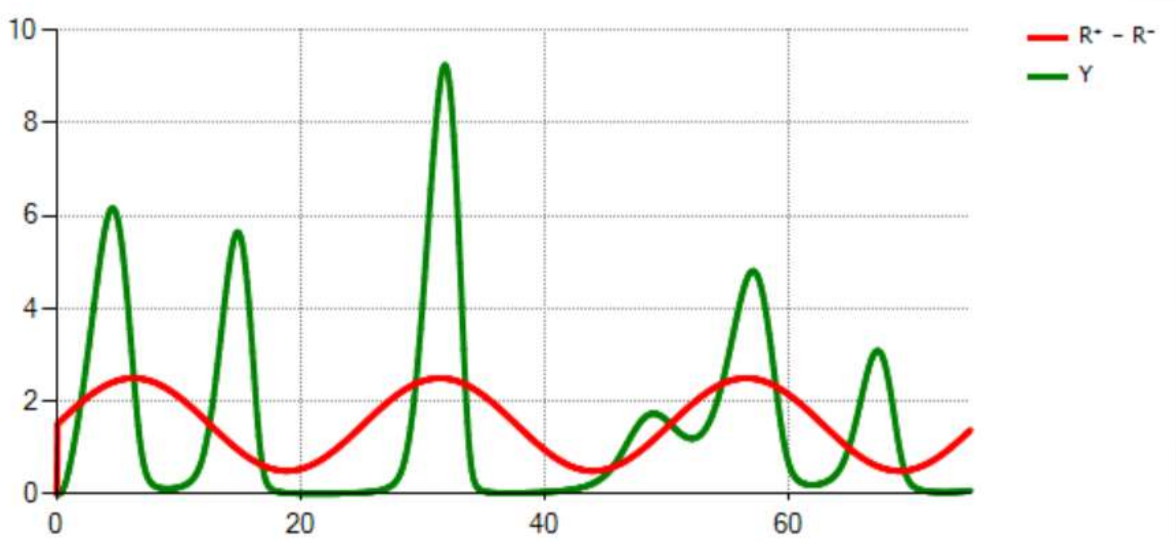
Testing the Controller

- Oscillating reference signal
- Just click the Play button a few times

  The parameters Kp, Ki, Kd are drawn from uniform random distributions.

  They can be individually frozen by checking the checkboxes





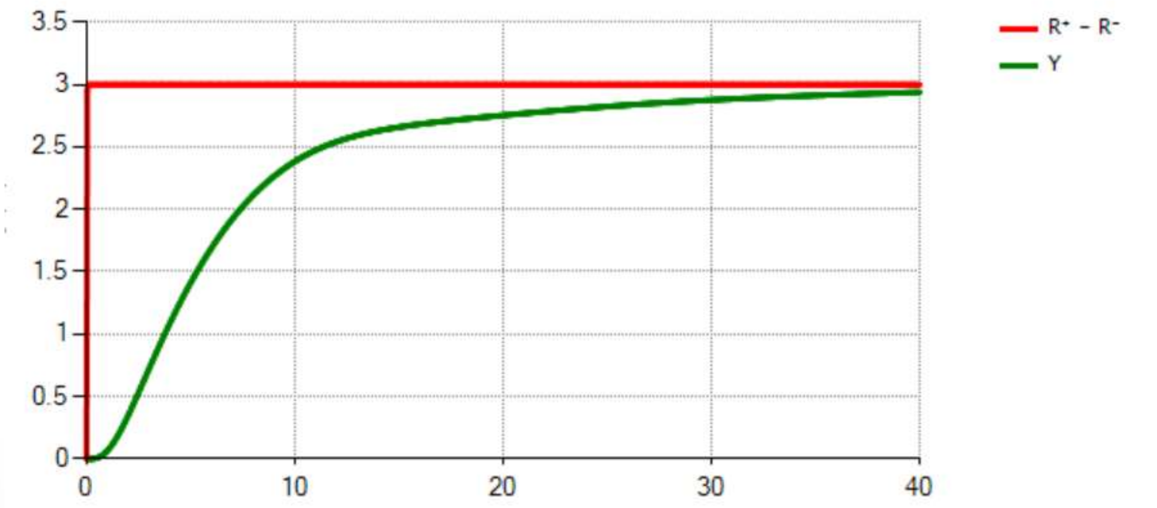
- Try to find and freeze some good parameters

Testing the Controller

- Constant reference signal
- This is easier

  Try changing the parameters by hand in this line (these are already pretty good):

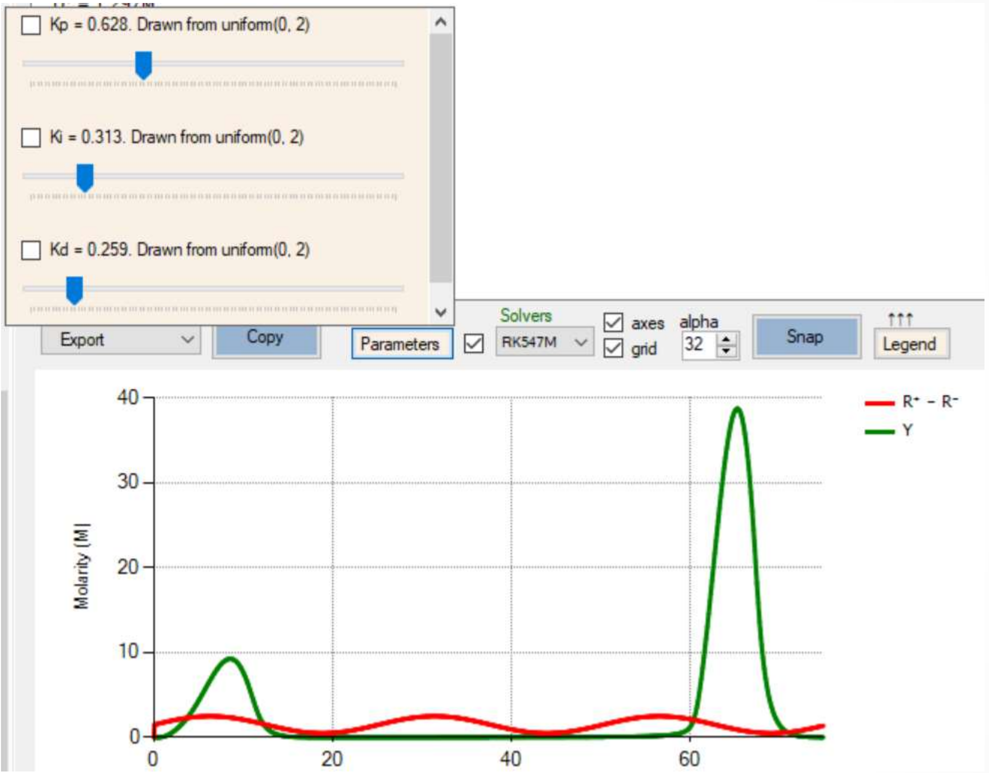
```
PIDController(R⁺,R⁻, Y, 0.1, 0.02, 0.02, Plant)
```



- Likely, you will need a different set of parameters here.

Biochemical Plant

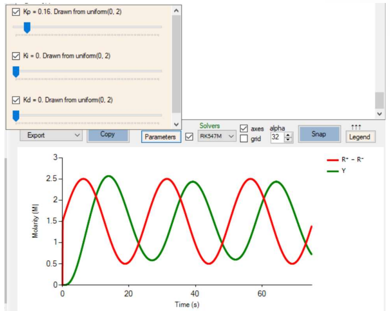
```
network Plant(species U⁺ U⁻ Protein) {
   species mRNA @ 0 M
   species microRNA @ 0 M
   U⁺ -> U⁺ + mRNA
   U⁻ -> U⁻ + microRNA

   mRNA -> mRNA + Protein
   Protein-> #
   mRNA + microRNA -> #
}


// OSCILLATING REFERENCE
number Kp =? uniform(0,2)
number Ki =? uniform(0,2)
number Kd =? uniform(0,2)

DSignal(R⁺,R⁻, fun(){1.5+sin(0.25*time)})
PIDController(R⁺,R⁻, Y, Kp, Ki, Kd, Plant)
```



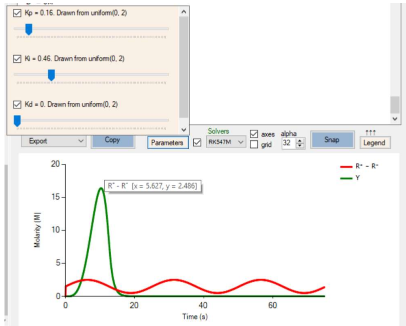
- Random sampling: Not so easy!

Biochemical Plant - my PD strategy

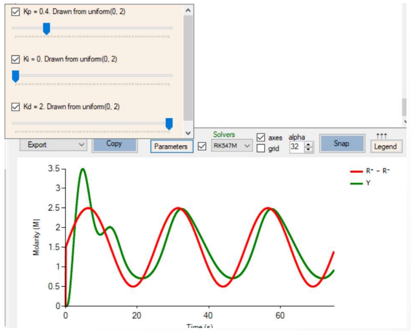
- 1. Adjust Kp until the amplitudes match (with Ki, Kd zero)



- 2. Adjusting Ki does not seem to help at all, leave it at zero

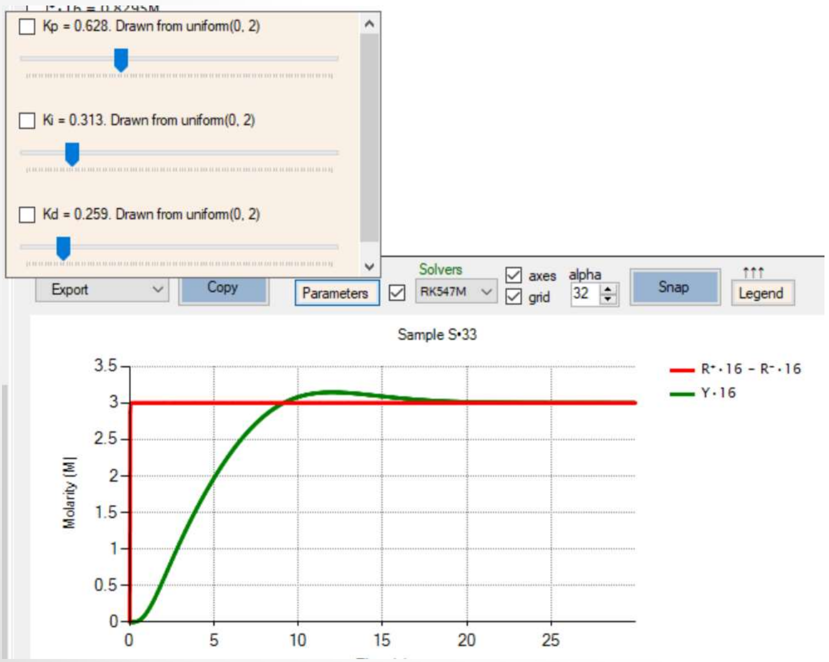


- 3. Adjusting Kd gives better results to put the oscillation in phase



Automatic Parameter Search

- Kaemika has a multi-dimensional gradient descent search (the "argmin" primitive, implementing the BFGS algorithm)
- Gradients (partial derivatives) must be provided for Kp, Ki, Kd; see the "PIDController Optimization" tutorial about how to do that

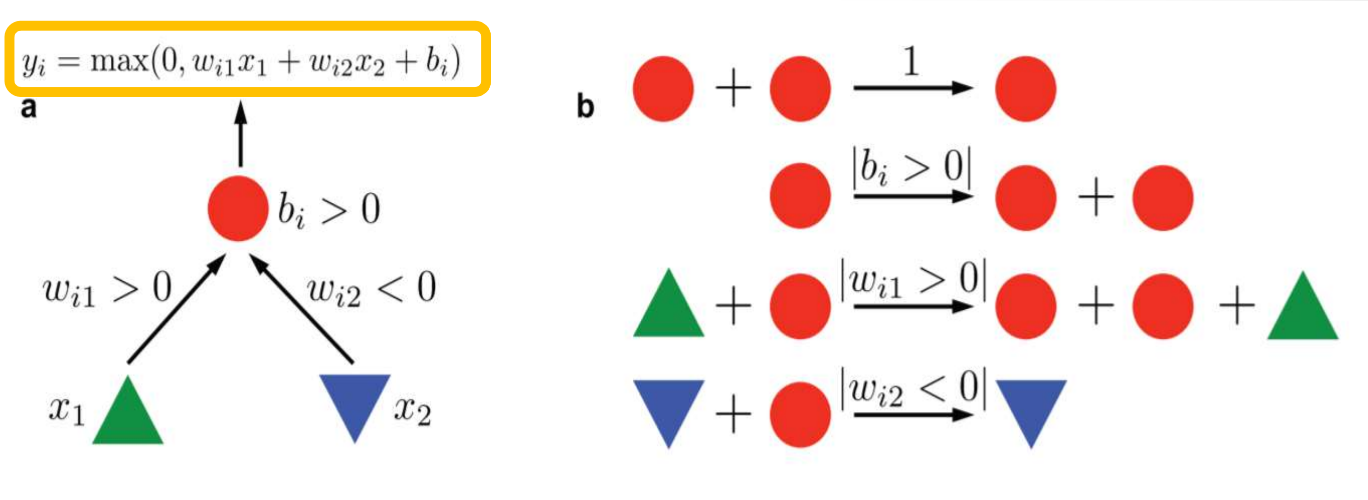


Was asked to zero-out error already at time 20.
So it overshoots a bit to get there in time.
It is using each of P, I, and D.

Found (local) minimum in only 5 tries.

Chemical Perceptron

```
species a @ 0.01 M
number bias = 0.3
number pw = 1.1
number nw = 0.9
species exc @ 1.5 M
species inh @ 0.4 M

a + a -> a                  {1}
a -> a + a                  {bias}
exc + a -> a + a + exc      {pw}
inh + a -> inh              {nw}

equilibrate for 40
```



Nanoscale artificial intelligence: creating artificial neural networks using autocatalytic reactions

Filippo Simini[1]

Conclusions

- Programming chemistry is fun. But it is no fun without modularization! Chemical reactions provide a nice almost-high-level language if properly modularized.
- There will always be "cheaper" ways of implementing those programs by direct "low-level chemical hacking" (c.f. trade-off between high-level and assembly languages).
- But a compiler could optimize higher-level programs for specific architectures (e.g. DNA strand displacement).
- There are already higher-level languages. Synthetic biology "programs" (gene assemblies) can be compiled from libraries of standard parts into molecules (plasmids). Chemical reactions there figure prominently as an "intermediate language" between gene specification and analysis.
- Control of biochemical "plants" is a major issue in synthetic biology.