

Contents

1 Clojure	1
1.1 TODO Use Clojure2D -> Celular Automata (Open Sourced)	1
1.1.1 Complexity out of simplicity	1
1.1.2 Complexity out of simplicity 2	2
1.1.3 If time permits	2
1.1.4 Celular automata example	3
1.2 Graphical Study - calculus	6
1.2.1 Sin and the unit circle	6
1.2.2 A composed shape	9
1.3 Waves (light, quanta) - Eletromagnetism	9
1.4 Mathematical Fields	11
1.5 Gases or/and heat	13
1.6 Viscosity/effect of media	14
1.7 Scalar project / dot products	16
1.8 Biology (people)	18
2 Julia	23
2.1 Plots	23
2.2 Animated graphics	24
2.3 Differential Equations	25
3 Do you know Julian Assange?	25
4 Dita diagrams	25
4.1 Freqtrade API	26

1 Clojure

1.1 **TODO** Use Clojure2D -> Celular Automata (Open Sourced)

Hook with the ideia of "A new kind of science" by Stephen Wolfram.

1.1.1 Complexity out of simplicity

"Well, it's rather amazing, we have a very simple rule. We are starting from a single black cell. But what we are getting out is an incredibly complicated pattern that seems in many ways random. It just doesn't seem right. We're putting in so little in, and getting so much out."

"In our everyday experience, say, doing engineering what we are used to [think] is that to make something complicated, we somehow must start off with complicated plans or use complicated rules."

-Stephen Wolfram

"So then, if you see on the other hand, that existence (...) it is a play of all kinds of patterns, and we can look upon different creatures as we look at different games, as we look at chess, checkers, backgammon, tennis. There is the, the tree game, the beetle game, the grass game. Or you can look at them as different styles of music – mazurkas, waltzes, sonata, etc. All down the line there are all these different things doing their stuff. They're going, "do-do-do-do-do..." in different rhythms."

-Allan Watts

1.1.2 Complexity out of simplicity 2

But, what we are seeing here is that, even extremely simple rules can produce incredibly complicated behaviour. It took me years to come to terms with this phenomena. And, in fact, it has gradually overturned almost everything I thought I knew about the foundations of Science. And that's what led me to build a new intellectual structure. Really, a 'New Kind of Science'."

-Stephen Wolfram

"And one has to get used, fundamentally, to the notion that different things can be inseparable, and that what is explicitly two can at the same time be implicitly one. If you forget that, very funny things happen. If, therefore, we forget, you see, that black and white are inseparable, and that existence is constituted equivalently by being and non-being, then we get scared, and we have to play a game called "Uh-oh, Black Might Win." And once we get into the fear that black – the negative side – might win, we are compelled to play the game, "But White Must Win," and from that start all our troubles."

-Allan Watts

1.1.3 If time permits

"And we're doing that. If you were in a flying saucer from Mars, or somewhere, and you came and looked, tried to make out what was living on this world from about ten thousand feet late at night, or early morning, you would see these great ganglia with tentacles going out all over the place. And early in the morning you would see little blobs of luminous particles going into the middle of them. Then in the late afternoon or early evening

it would spit them all out again. And they'd say, "Well, this thing breathes, and it does it in a special rhythm. It goes in – and – out, in – and – out, and in – and – out, once every twenty – four hours. But then it rests a day and doesn't spit so much, it just spits in a different way. There is a kind of irregularity, and then it starts spitting all over again the same way. " They would say, "Well, that is very interesting, but that is just the kind of thing we have. This is something that goes this way, and then goes that way."

-Allan Watts

1.1.4 Celular automata example

1. Importações

```
(ns examples.NOC.ch07.wolframca-figures-7-2
  (:require [clojure2d.core :refer :all]
            [fastmath.core :as m]
            [fastmath.random :as r]))
```

2. Constantes

```
(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed)
(m/use-primitive-operators)

(def ^:const ^int scl 8)
(def ^:const ^int w 1000)
(def ^:const ^int h 800)

(def ^:const ^int cell-no (/ w scl))
(def ^:const ^int rows (/ h scl))

(def ^:const wrap? false) ;; change to wrap cells
```

3. Criar uma regra, a partir da lista de possibilidades

```
(defn make-rule
  "Create rule table based on Celular Automata (C.A.) number."
  [^long id]
  (mapv
    #(if (zero? (bit-and id (bit-shift-left 1 ^long %)))
```

```

    0
    1)
  (range 7 -1 -1)))

(defn apply-rule
  "Create rule number from 'a', 'b', 'c', parameters (values 0 or 1)
  and return result from rule table."
  [rule a b c]
  (let [s (str a b c)
        idx (Integer/parseInt s 2)]
    (rule idx)))

(defn init-cells
  "Create first line with one single seed in the middle"
  [^long size]
  (mapv
   #(if (== ^long % (m/floor (/ size 2)))
       1
       0)
   (range size)))

(defn next-cells
  "Calculate next line based on previous and rule, wrap result."
  [cells rule]
  (let [s (count cells)]
    (mapv
     #(let [^long v %
            [l r] (if wrap?
                    [(cells (int (m/wrap 0 s (dec v))))
                     (cells (int (m/wrap 0 s (inc v))))]
                    [(if (zero? v) 0 (cells (dec v)))
                     (if (< v (dec s)) (cells (inc v)) 0)]]]
          (apply-rule rule l (cells v) r))
     (range s))))

(defn draw-cells
  "Draw cells."
  [canvas rule]
  (loop [cells (init-cells cell-no)
        row (int 0)]

```

```

      (when (< row rows)
        (dotimes [x cell-no]
          (if (== ^int (cells x) 1)
            (set-color canvas :black)
            (set-color canvas :white))
          (rect canvas (* x scl) (* row scl) scl scl)
          (set-color canvas :black)
          (rect canvas (* x scl) (* row scl) scl scl true))
          (recur (next-cells cells rule)
            (inc row))))))

(def cnvs (canvas w h))
(def window (show-window cnvs "Wolframca figures 7_1"))

(defn draw-rule
  "Two cases. Empty or with a rule."
  ([ ] (draw-rule (r/irand 256)))
  ([rule]
   (println (str "Rule: " rule))
   (with-canvas-> cnvs
     (draw-cells (make-rule rule))))))

(defmethod mouse-event
  ["Wolframca figures 7_1" :mouse-clicked] [_ _]
  (draw-rule))

(draw-rule 29)

```

4. Rederizar regras

```

(def cnvs (canvas w h))
(def window (show-window cnvs "Wolframca figures 7_1"))
(draw-rule 122)

(defn renderize-rule [nth-rule]
  (do
    (def cnvs (canvas 1000 800))
    (def window (show-window cnvs (str "rule number " nth-rule)))
    (draw-rule nth-rule)))

```

```

(renderize-rule 331)

(renderize-rule 332)

(renderize-rule 421)

(renderize-rule 900)

(renderize-rule 993)

(renderize-rule 99)

(renderize-rule 102)

(renderize-rule 103)

(renderize-rule 105)

(renderize-rule 106)

(renderize-rule 115)

(renderize-rule 114)

(renderize-rule 118)

(renderize-rule 120)

```

1.2 Graphical Study - calculus

1.2.1 Sin and the unit circle

```

(ns GG.M.M-2-1-01
  (:require [clojure2d.core :refer :all]
    [fastmath.core :as m]
    [fastmath.vector :as v]
    [clojure2d.color :as c]))

(def ^:const wname "M_2_1_01")

```

```

(defn draw
  ""
  [canvas window ^long frame _]
  (let [{:keys [phi
~double freq
draw-animation?]} (get-state window)
point-count (if draw-animation?
  (- (width canvas) 400)
  (width canvas ))
;; Define the shape of sin
shape (for [i (range point-count)
:let [angle (m/norm i
0 (/ point-count 1)
0 (/ m/TWO_PI 1))
y (m/sin (+
  (* angle freq)
  (m/radians phi)))]
(v/vec2 i (* y 100.0)))]

  (-> canvas
(set-background :white)
(set-color :black)
(set-stroke 2.0)
(translate (if draw-animation?      ;; translate x y <-|
  240                                ;; v
  0)
  (/ (height canvas) 2))
(path shape)                        ;; define f(y)=sin
)

  (when draw-animation?
    (let [t (m/frac (/ (double frame) point-count))
angle (* t m/TWO_PI)
v (+ (* angle freq) (m/radians phi))
x (- (* 100.0 (m/cos v)) 125.0)
y (* 100.0 (m/sin v))
tpc (* t point-count)
phi-x (- (* 100 (m/cos (m/radians phi))) 125)
phi-y (* 100 (m/sin (m/radians phi)))]

```

```

(-> canvas
  (set-stroke 1.0)
  (ellipse -125 0 200 200 true)

  (set-color :black 128)
  (line 0 -100 0 100)      ;; y-axis -> cartesian
  (line 0 0 point-count 0) ;; x-axis -> cartesian
  (line -225 0 -25 0)      ;; x-axis -> circle
  (line -125 -100 -125 100) ;; y-axis -> circle
  (line x y -125 0)        ;; r      -> circle

  (set-color 0 130 164)
  (set-stroke 2.0)
  (line tpc y tpc 0)      ;; height -> sin-curve
  (line x y x 0)          ;; height -> sin-circle

  (set-stroke 1.0)
  (set-color :black 128)
  (line -125 0 phi-x phi-y) ;; initial angle

  (set-stroke 2.0)
  ;; start-dot
  (filled-with-stroke :black :white
  ellipse 0 phi-y 8 8)
  ;; phi-dot
  (filled-with-stroke :black :white
  ellipse phi-x phi-y 8 8)
  ;; curve-dot
  (filled-with-stroke :black :white
  ellipse tpc y 10 10)
  ;; circle-dot
  (filled-with-stroke :black :white
  ellipse x y 10 10)
  ;; (filled-with-stroke :black :white
  ;; ellipse x (/ y 2) 10 10)
  )))))

(def window (show-window {:canvas (canvas 800 400)
  :window-name wname

```



```

:draw-fn #(draw %1 %2 %3 %4)
:state {:phi 0.0
:freq 2.0
:draw-animation? true}}))

(defmethod key-pressed [wname \a] [_ s]
  (update s :draw-animation? not))

(defmethod key-pressed [wname \1] [_ s]
  (update s
    :freq #(max 1 (dec ^double %))))

(defmethod key-pressed [wname \2] [_ s]
  (update s :freq inc))

(defmethod key-pressed [wname virtual-key] [e s]
  (case (key-code e)
    :left (update s :phi #(+ ^double % 15.0))
    :right (update s :phi #(- ^double % 15.0))
    s))

```

1.2.2 A composed shape

1.3 Waves (light, quanta) - Eletromagnetism

```

(ns GG.M.M-2-3-01
  (:require [clojure2d.core :refer :all]
    [fastmath.core :as m]
    [fastmath.vector :as v]))

(def ^:const wname "M_2_3_01")

(defn draw-shapes
  ""
  [canvas {:keys [phi freq mod-freq draw-frequency? draw-modulation?]}]
  (let [scaling (* (/ (height canvas) 4))
    info-fn #(m/sin (+ (* % freq) (m/radians phi)))
    carrier-fn #(m/cos (+ (* % mod-freq)))]
    angles (map #(vector % (m/norm % 0 (width canvas) 0 m/TWO_PI)) (range (width canvas))))

```

```

    (-> canvas
      (set-background :white)
      (translate 0 (* scaling 2)))

    (when draw-frequency?
      (-> canvas
        (set-color 0 130 164)
        (path (for [[i angle] angles]
          (v/vec2 i (* scaling (info-fn angle)))))))

    (when draw-modulation?
      (-> canvas
        (set-color 0 130 164 128)
        (path (for [[i angle] angles]
          (v/vec2 i (* scaling (carrier-fn angle)))))))

    (-> canvas
      (set-color :black)
      (set-stroke 2.0)
      (path (for [[i angle] angles]
        :let [info (info-fn angle)
              carrier (carrier-fn angle)]
          (v/vec2 i (* info carrier scaling))))))

(def cnvs (canvas 800 400))
(def window (show-window {:canvas cnvs
  :window-name wname
  :state {:phi 0.0
    :freq 2.0
    :mod-freq 12.0
    :draw-frequency? true
    :draw-modulation? true}}))

(defn draw
  ""
  [s]
  (with-canvas-> cnvs (draw-shapes s))
  s)

```

```

(defmethod key-pressed [wname \i] [_ s] (draw (update s :draw-frequency? not)))
(defmethod key-pressed [wname \c] [_ s] (draw (update s :draw-modulation? not)))

(defmethod key-pressed [wname \1] [_ s] (draw (update s :freq #(max 1 (dec %)))))
(defmethod key-pressed [wname \2] [_ s] (draw (update s :freq inc)))

(defmethod key-pressed [wname \7] [_ s] (draw (update s :mod-freq #(max 1 (dec %)))))
(defmethod key-pressed [wname \8] [_ s] (draw (update s :mod-freq inc)))

(defmethod key-pressed [wname virtual-key] [e s]
  (case (key-code e)
    :left (draw (update s :phi #(+ % 15.0)))
    :right (draw (update s :phi #(- % 15.0)))
    s))

(draw (get-state window))

```

1.4 Mathematical Fields

```

(ns GG.M.M-1-5-01
  (:require [clojure2d.core :refer :all]
    [clojure2d.color :as c]
    [fastmath.random :as r]
    [fastmath.core :as m]
    [fastmath.fields :as f]))

(def ^:const wname "M_1_5_01")

(def ^:const w 800)
(def ^:const h 800)
(def ^:const arc-color (c/color 0 130 164 100))
(def ^:const tile-size 40.0)
(def ^:const tile-size-75 (* 0.75 tile-size))
(def ^:const tile-size-25 (* 0.25 tile-size))
(def ^:const grid-resolution-x (m/round (/ w tile-size)))
(def ^:const grid-resolution-y (m/round (/ h tile-size)))

(def arrow (transcode-svg (load-svg "src/GG/data/arrow.svg") tile-size-75 tile-size-75))

(defn draw

```

```

""
[canvas window _ _]
  (let [{:keys [noise debug]} (get-state window)
        noise-x-range (/ (max 1 (mouse-x window)) 100.0)
        noise-y-range (/ (max 1 (mouse-y window)) 100.0)]
    (set-background canvas :white)

    (dotimes [gy (inc grid-resolution-y)]
      (dotimes [gx (inc grid-resolution-x)]
        (let [noise-x (m/norm gx 0 grid-resolution-x 0 noise-x-range)
              noise-y (m/norm gy 0 grid-resolution-y 0 noise-y-range)
              ^double noise-value (noise noise-x noise-y)
              angle (* noise-value m/TWO_PI)]

          (-> canvas
              (push-matrix)
              (translate (* tile-size gx) (* tile-size gy)))

          (when debug
            (-> canvas
              (set-color (c/gray (* noise-value 255.0)))
              (ellipse 0 0 tile-size-25 tile-size-25)))

          (-> canvas

              (set-stroke 1.0 :square)
              (set-color arc-color)
              (arc 0 0 tile-size-75 tile-size-75 0 angle)

              (rotate angle)
              (image arrow 0 0)
              (pop-matrix))))))

(def window (show-window {:canvas (canvas w h)
                          :window-name wname
                          :draw-fn draw
                          :state (let [nc (r/random-noise-cfg)]
                                    {:noise-cfg nc
                                     :noise (r/fbm-noise nc)})
                          :noise (r/fbm-noise nc)

```

```

:debug true}}}))

(defmethod key-pressed [wname \space] [_ s]
  (let [nc (r/random-noise-cfg)
        ns (assoc s :noise-cfg nc :noise (r/fbm-noise nc))]
    (println ns)
    ns))

(defmethod key-pressed [wname \d] [_ s] (update s :debug not))

(defmethod key-pressed [wname virtual-key] [e s]
  (let [^double falloff (get-in s [:noise-cfg :gain])
        ^long octaves (get-in s [:noise-cfg :octaves])
        ^double lacunarity (get-in s [:noise-cfg :lacunarity])]
    ns (condp = (key-code e)
          :up (assoc-in s [:noise-cfg :gain] (m/constrain (+ falloff 0.05) 0.0 1.0))
          :down (assoc-in s [:noise-cfg :gain] (m/constrain (- falloff 0.05) 0.0 1.0))
          :left (assoc-in s [:noise-cfg :octaves] (max 1 (dec octaves)))
          :right (assoc-in s [:noise-cfg :octaves] (inc octaves))
          :page_up (assoc-in s [:noise-cfg :lacunarity] (+ lacunarity 0.1))
          :page_down (assoc-in s [:noise-cfg :lacunarity] (- lacunarity 0.1))
          s)
    (println (:noise-cfg ns))
    (assoc ns :noise (r/fbm-noise (:noise-cfg ns)))))

```

1.5 Gases or/and heat

```

(ns examples.NOC.ch01.bouncingball-vectors-1-2
  (:require [clojure2d.core :refer :all]
             [fastmath.vector :as v])
  (:import fastmath.vector.Vec2))

(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed)

(defn boundary-check
  "Return -1.0 if out of borders, 1.0 otherwise"
  [^double mx1 ^double mx2 ^Vec2 v]

```

```

      (Vec2. (if (< -1.0 (.x v) mx1) 1.0 -1.0)
        (if (< -1.0 (.y v) mx2) 1.0 -1.0)))

(defn draw
  "Bounce ball"
  [canvas _ _ state]
  (let [[position velocity] (or state [(Vec2. 100 100)
    (Vec2. 2.5 5.0)])]
    ^Vec2 nposition (v/add position velocity))

    (-> canvas
      (set-background 255 255 255 10)
      (set-color 175 175 175)
      (ellipse (.x nposition) (.y nposition) 16 16)
      (set-color 0 0 0)
      (ellipse (.x nposition) (.y nposition) 16 16 true))

    [nposition
      (v/emult velocity (boundary-check (width canvas) (height canvas) nposition))]))

(def window (show-window (black-canvas 200 200) "Example 1-2: Bouncing Ball, with Vec2

```

1.6 Viscosity/effect of media

```

(ns examples.NOC.ch02.fluidresistance-2-5
  (:require [clojure2d.core :refer :all]
    [fastmath.core :as m]
    [fastmath.random :as r]
    [fastmath.vector :as v])
  (:import fastmath.vector.Vec2))

(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed)

(def ^:const ^int w 640)
(def ^:const ^int h 360)
(def ^:const ^int h2 (/ h 2))

(def ^:const ^int number-of-movers 9)

```

```

(def gravity (Vec2. 0.0 0.1))
(def ^:const ^double c 0.1)

(deftype Mover [position velocity ^double mass]
  Object
  (toString [_] (str position " : " velocity)))

(defn make-mover
  "Create Mover"
  []
  (->Mover (Vec2. (r/drand w) 0.0)
    (Vec2. 0.0 0.0)
    (r/drand 1.0 4.0)))

(defn apply-force
  "Apply force"
  [a f mass]
  (v/add a (v/div f mass)))

(defn check-edges
  "Check window boundaries"
  [^Vec2 velocity ^Vec2 pos]
  (if (> (.y pos) h)
    [(Vec2. (.x velocity) (* -0.9 (.y velocity))) (Vec2. (.x pos) h)]
    [velocity pos]))

(defn move-mover
  "Move mover"
  [^Mover m]
  (let [acc (-> (Vec2. 0.0 0.0)
    (apply-force (if (> (.y ^Vec2 (.position m)) h2)
      (let [drag-magnitude (* c (m/sq (v/mag (.velocity m)))]
        (-> (.velocity m)
          (v/mult -1.0)
          (v/normalize)
          (v/mult drag-magnitude)))
        (Vec2. 0.0 0.0)) (.mass m))
    (apply-force (-> (.velocity m)
      (v/normalize)
      (v/mult -0.05)) (.mass m))
    ]))
    ]))

```

```

(v/add gravity))
vel (v/add (.velocity m) acc)
pos (v/add (.position m) vel)
[new-vel new-pos] (check-edges vel pos)]
  (->Mover new-pos new-vel (.mass m))))

(defn draw-and-move
  "Draw mover, move and return new one."
  [canvas ^Mover m]
  (let [size (* 16.0 ^double (.mass m))]
    (-> canvas
      (set-color 127 127 127 200)
      (ellipse (.x ^Vec2 (.position m)) (.y ^Vec2 (.position m)) size size false)
      (set-stroke 2)
      (set-color :black)
      (ellipse (.x ^Vec2 (.position m)) (.y ^Vec2 (.position m)) size size true))
      (move-mover m)))

(defn draw
  "Draw movers on canvas"
  [canvas window _ _]
  (-> canvas
    (set-background :white)
    (set-color 50 50 50)
    (rect 0 h2 w h2))
    (set-state! window (mapv (partial draw-and-move canvas) (get-state window))))

(def window (show-window {:canvas (canvas w h)
  :window-name "NOC_2_5_fluidresistance"
  :draw-fn draw
  :state (repeatedly number-of-movers make-mover)}))

(defmethod mouse-event ["NOC_2_5_fluidresistance" :mouse-released] [_ _]
  (repeatedly number-of-movers make-mover))

```

1.7 Scalar project / dot products

```

(ns examples.NOC.ch06.simplescalarprojection
  (:require [clojure2d.core :refer :all]
    [fastmath.core :as m])

```



```

[fastmath.vector :as v])
(:import fastmath.vector.Vec2))

(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed)

(def ^Vec2 a (Vec2. 20 300))
(def ^Vec2 b (Vec2. 500 250))

(defn scalar-projection
  ""
  [p a b]
  (let [ap (v/sub p a)
        ab (v/normalize (v/sub b a))]
    (-> ab
      (v/mult (v/dot ap ab))
      (v/add a))))

(defn draw
  ""
  [canvas window _ _]
  (let [^Vec2 mouse (mouse-pos window)
        ^Vec2 norm (scalar-projection mouse a b)]
    (-> canvas
      (set-background :white)
      (set-color :black)
      (set-stroke 2.0)
      (line (.x a) (.y a) (.x b) (.y b))
      (line (.x a) (.y a) (.x mouse) (.y mouse))
      (ellipse (.x a) (.y a) 8 8)
      (ellipse (.x b) (.y b) 8 8)
      (ellipse (.x mouse) (.y mouse) 8 8)
      (set-color 50 50 50)
      (set-stroke 1.0)
      (line (.x mouse) (.y mouse) (.x norm) (.y norm))
      (set-color :red)
      (ellipse (.x norm) (.y norm) 16 16))))

(def window (show-window (canvas 600 360) "Simple scalar projection" draw))

```

1.8 Biology (people)

;; This is the limited port of paperjs example <http://paperjs.org/examples/tadpoles/>
;; it is not implementing movement along a path.

```
(ns examples.ex57-flocking
  (:require [clojure2d.core :refer :all]
            [fastmath.core :as m]
            [fastmath.random :as r]
            [fastmath.vector :as v])
  (:import [fastmath.vector Vec2]))

(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed)
(m/use-primitive-operators)

(def ^:const ^double w 1000)
(def ^:const ^double h 600)

(def ^Vec2 zero-vec (Vec2. 0 0))

(defn mk-boid [^Vec2 position ^double max-speed ^double max-force]
  (let [strength (r/drnd 0 0.5)
        amount (+ (* strength 10) 10)]
    {:acceleration (Vec2. 0 0)
     :vector (Vec2. (r/drnd -2 2) (r/drnd -2 2))
     :position position
     :radius 30
     :max-speed (+ max-speed strength)
     :max-force (+ max-force strength)
     :amount amount
     :count 0
     :head {:size [13 8]}
     :path (mapv (fn [_] zero-vec) (range amount) )
     :short-path (mapv (fn [_] zero-vec) (range (m/min 3 amount))))
    )))
```

```

(defn steer [this ^Vec2 target slowdown]
  (let [desired (v/sub target (:position this))
        distance (v/mag desired)
        dl (if (and slowdown (< distance 100))
                (* ^double (:max-speed this) (/ distance 100))
                (:max-speed this))
        steer-v (v/sub (v/set-mag desired dl) (:vector this))]
    (v/limit steer-v ^double (:max-force this))))

(defn seek [{ acc :acceleration :as this} ^Vec2 target]
  (update this :acceleration (partial v/add (steer this target false))))

(defn arrive [{ acc :acceleration :as this} ^Vec2 target]
  (update this :acceleration (partial v/add (steer this target true))))

(defn align [this boids]
  (let [nd 35.0
        [s ^double c] (reduce (fn [[^Vec2 ste ^double cnt] b]
                                (let [dst (v/dist (:position this) (:position b))]
                                  (if (and (pos? dst) (< dst nd) )
                                      [(v/add ste (:vector b)) (inc cnt)]
                                      [ste cnt]))) [zero-vec 0] boids)
        s' (if (pos? c) (v/div s c) s)]
    (if (not= 0 (v/mag s'))
        (let [s1 (v/set-mag s' (:max-speed this))
              sv (v/sub s1 (:vector this))]
          (v/limit sv (:max-force this)))
        s'))))

(defn cohesion [this boids]
  (let [nd 120
        ^Vec2 s ^double c] (reduce (fn [[ste ^double cnt] b]
                                      (let [dst (v/dist (:position this) (:position b))]
                                        (if (and (pos? dst) (< dst nd) )
                                            [(v/add ste (:vector b)) (inc cnt)]
                                            [ste cnt]))) [zero-vec 0] boids)
    (if (and (pos? c) (< c nd) )
        (let [s1 (v/set-mag s' (:max-speed this))
              sv (v/sub s1 (:vector this))]
          (v/limit sv (:max-force this)))
        s'))))

```

```

    [(v/add ste (:position b)) (inc cnt)]
    [ste cnt]))) [zero-vec 0] boids)]
    (if (pos? c)
        (steer this (v/div s c) false)
        s)))

(defn separate [this boids]
  (let [des-sep 80
        [s ^double c] (reduce (fn [[^Vec2 ste ^double cnt] b]
                                [(v/add ste (v/mult (v/normalize vect) (/ 1.0 dst))) (inc cnt)]
                                [ste cnt]))) [zero-vec 0] boids)
        [vect (v/sub (:position this) (:position b))
         dst (v/mag vect)]
        (if (and (pos? dst) (< dst des-sep))
            [(v/add ste (v/mult (v/normalize vect) (/ 1.0 dst))) (inc cnt)]
            [ste cnt]))) [zero-vec 0] boids)
    s' (if (pos? c) (v/div s c) s)]
    (if (not= 0 (v/mag s'))
        (let [sl (v/set-mag s' (:max-speed this))
              sv (v/sub sl (:vector this))]
          (v/limit sv ^double (:max-force this)))
        s'))))

(defn flock [this boids]
  (let [s (v/mult (separate this boids) 0.6)
        a (align this boids)
        c (cohesion this boids)]
    (assoc this :acceleration (v/add (:acceleration this) (v/add s (v/add a c))))))

(defn update-boid [{:keys [vector position acceleration max-speed] :as b}]
  (let [speed (v/add vector acceleration)
        vec (v/limit speed max-speed)]
    (assoc b :vector vec :position (v/add position vec) :acceleration zero-vec)))

(defn draw-head [cvs {:keys [head] :as b}]
  (let [ang (v/heading (:vector b))
        [x y] (:position b)
        [ew eh] (:size head)]
    (draw-head cvs x y ang ew eh)))

```

```

(with-canvas-> cvs
  (push-matrix)
  (translate x y)
  (rotate ang)
  (ellipse 0 0 ew eh)
  (pop-matrix)))

```

b)

```

(defn initial-state []
  {:boids (repeatedly 30 #(mk-boid (Vec2. (r/drand w) (r/drand h)) 10 0.05))
   :group false})

```

```

(defn borders [{:keys [position ^double radius] :as boid}]
  (let [[^double px ^double py] position
        vv
        (->> [0 0]
          ((fn [[x y]] [(if (neg? (+ px radius)) (+ w radius) x) y]))
          ((fn [[x y]] [x (if (neg? (+ py radius)) (+ h radius) y))])
          ((fn [[x y]] [(if (> px (+ w radius)) (+ (- w) (- radius)) x) y]))
          ((fn [[x y]] [x (if (> py (+ h radius)) (+ (- h) (- radius)) y))])
          (apply v/vec2 )))
    (if (not= (v/mag vv) 0)
      (assoc boid :position (v/add position vv) :path (mapv #(v/add vv %) (:path boid))
              boid)))

```

```

(defn calc-tail [cvs this]
  (let [speed (v/mag (:vector this))
        pl (+ 5 (/ speed 3.0))]

```

```

[seg ss c] (loop [point (:position this)
                  last-vec (v/mult (:vector this) -1)
                  seg (assoc (:path this) 0 point)
                  s-seg (assoc (:short-path this) 0 point)
                  ^double cnt (:count this)
                  i 1]
  (if (< i ^double (:amount this))
    (let [vect (v/sub (nth seg i) point)]

```

```

    c (+ cnt (* speed 10))
    wave (m/sin (/ (+ c (* i 3)) 300))
    sway (v/mult (v/normalize (v/rotate last-vec m/HALF_PI)) wave)
    p (v/add point (v/add (v/mult (v/normalize last-vec) pl) sway))
    (recur p vect (assoc seg i p) (if (< i 3) (assoc s-seg i p) s-seg) c (inc i))
    [seg s-seg cnt]]]
  (set-stroke cvs 4)
  (path cvs ss)
  (set-stroke cvs 2)
  (path cvs seg)
  (assoc this :path seg :short-path ss :count c)
) )

```

```

(defn run-boids [canvas boid {:keys [group boids] :as state}]
  (let [b (assoc boid :last-loc (:position boid))]

```

```

    (->> b
      ((fn [b] (if group
        b
        (flock b boids))))
      (borders)
      (update-boid)
      (calc-tail canvas)
      (draw-head canvas ))))

```

```

(defn get-path-target [^long i ^long n ^long f]
  (let [f' (long (/ f 30))
        a (* m/TWO_PI (/ (double(mod (+ i f') n)) (double n))) ]
    (v/add (v/vec2 (/ w 2) (/ h 2) ) (v/mult (v/vec2 (m/cos a) (m/sin a)) (* h 0

```

```

(let [canvas (canvas w h :high)
      draw (fn [cvs wnd frm state]
        (let [ev (get-state wnd)
              gr (:group state)

```

```

state (assoc state :group (if (= ev :change) (not gr) gr))
{:keys [boids group]} state
cb (count boids)]
  (set-state! wnd :none)
  (set-background cvs :black)
  (set-color cvs :white)
  (text cvs "click in wndow for a surprise" 10 16)

  (assoc state :boids (vec (map-indexed
(fn [i b]
  (let [b' (if group (arrive b (get-path-target i cb frm)) b)]

    (run-boids cvs b' state))) boids))))))

  wnd (show-window {:canvas canvas
:draw-fn draw
>window-name "boids"
:draw-state (initial-state)}))
  (defmethod mouse-event ["boids" :mouse-pressed] [e _]
    (set-state! wnd :change)))

```

2 Julia

2.1 Plots

using GR

```

x = 8 .* rand(100) .- 4
y = 8 .* rand(100) .- 4
z = sin.(x) .+ cos.(y)
# Draw the surface plot
surface(x, y, z)
# Create example grid data
x = LinRange(-2, 2, 40)
y = LinRange(0, pi, 20)
z = sin.(x') .+ cos.(y)
# Draw the surface plot
GR.surface(x, y, z)
# Draw the surface plot using a callable

```

```

GR.surface(x, y, (x,y) -> sin(x) + cos(y))

function hailLength(x::Int)
    n = 0
    while x != 1
    if x % 2 == 0
        x = Int(x/2)
    else
        x = 3x +1
    end
    n += 1
    end
    return n
end

lengths = [hailLength(x0) for x0 in 2:10^7]

(setq ein:output-area-inlined-images t)

GR.histogram(lengths,xlabel="Length", ylabel="Frequency")

```

2.2 Animated graphics

```

using Plots
# define the Lorenz attractor
Base.@kwdef mutable struct Lorenz
    dt::Float64 = 0.02
    ::Float64 = 10
    ::Float64 = 28
    ::Float64 = 8/3
    x::Float64 = 1
    y::Float64 = 1
    z::Float64 = 1
end

function step!(l::Lorenz)
    dx = l.y - l.x
    dy = l.x * (l.z - 1) - l.y
    dz = l.x * l.y - l.z
    l.x += l.dt * dx

```



```

        l.y += l.dt * dy
        l.z += l.dt * dz
    end

    attractor = Lorenz()

    # initialize a 3D plot with 1 empty series
    plt = plot3d(
        1,
        xlim = (-30, 30),
        ylim = (-30, 30),
        zlim = (0, 60),
        title = "Lorenz Attractor",
        marker = 2,
    )

    # build an animated gif by pushing new points to the plot, saving every 10th frame
    @gif for i=1:1500
        step!(attractor)
        push!(plt, attractor.x, attractor.y, attractor.z)
    end every 10

```

2.3 Differential Equations

```

# Carrega o gerenciador de pacotes, Pkg e instala, se preciso, o pacote.
using Pkg
Pkg.add("DifferentialEquations")
# Porta as utilidades do pacote, sem necessitar de referí-lo no meio do código
using DifferentialEquations

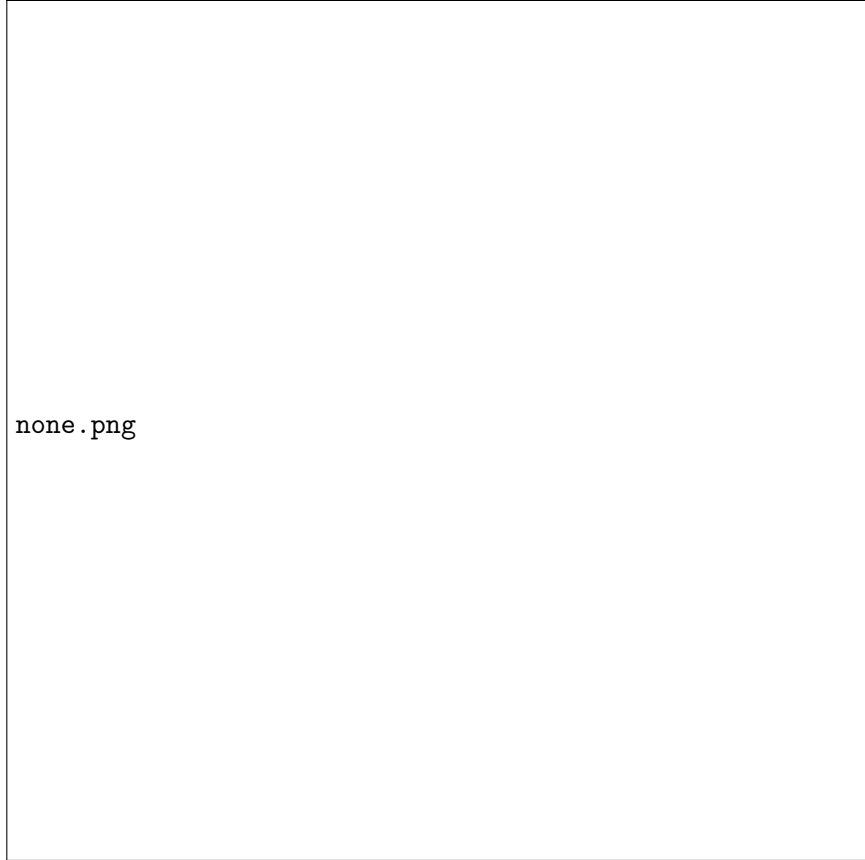
```

3 Do you know Julian Assange?

Julian Assange used and extend Emacs

4 Dita diagrams

4.1 Freqtrade API



none.png

`images/hello-world.png`