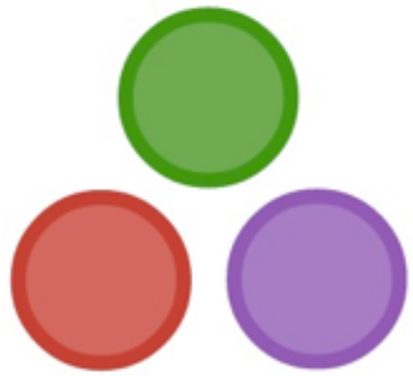
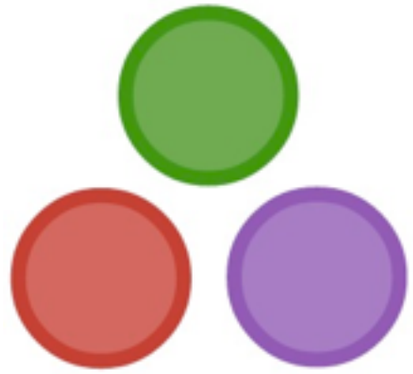


Implementing Machine Learning Algorithms in Julia

James Porter
Chicago Julia Meetup



“Julia in Anger”

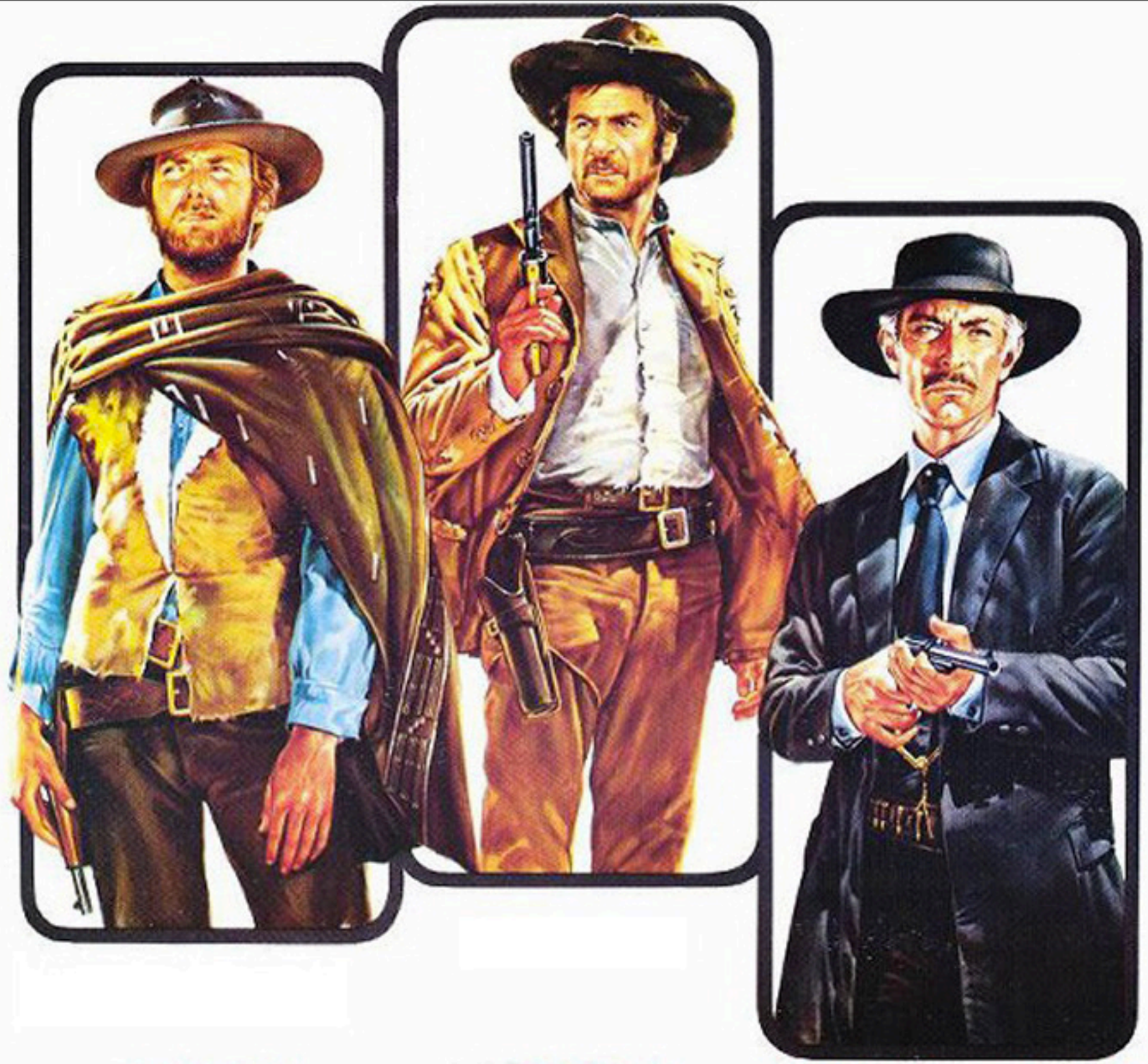
James Porter
Chicago Julia Meetup

who?


Cres
cat Sci
entia
Vita
Exco
latur

OPEN SCIENCE DATA CLOUD

this talk



THE GOOD THE BAD AND THE UGLY

k-means clustering

perceptron

belief propagation

HMMs

eigendecomposition/SVDs

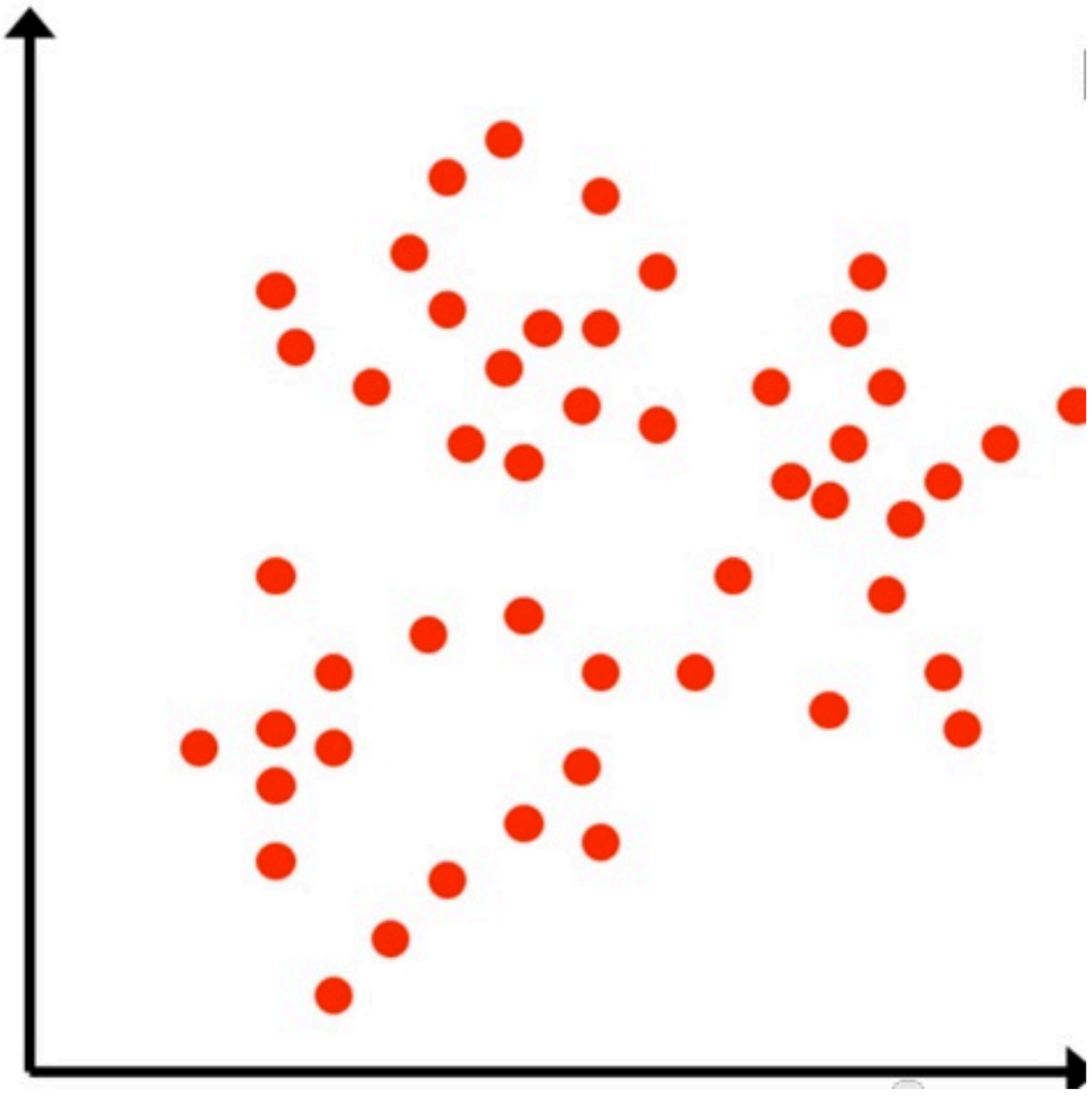
k-means clustering

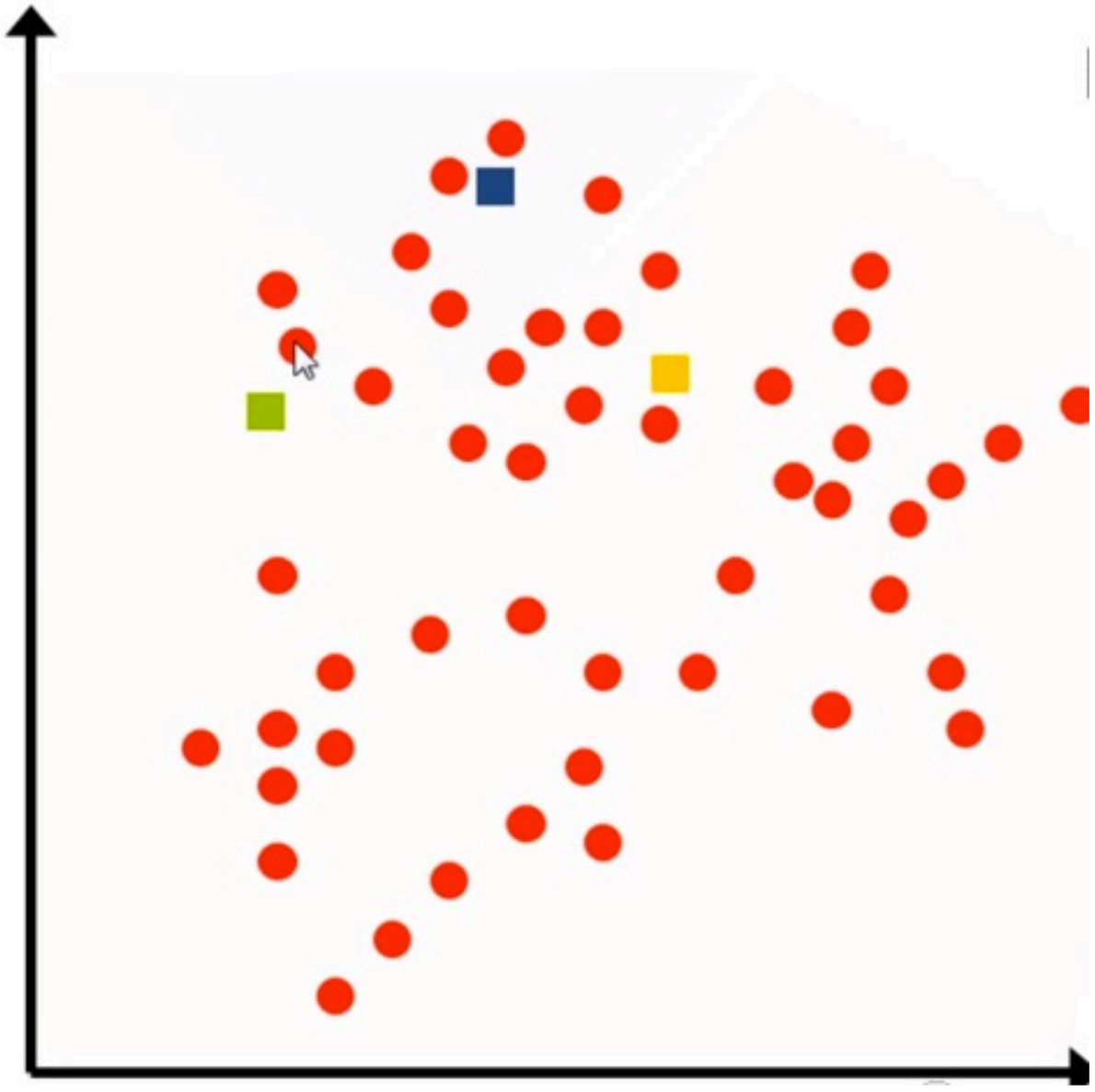
perceptron

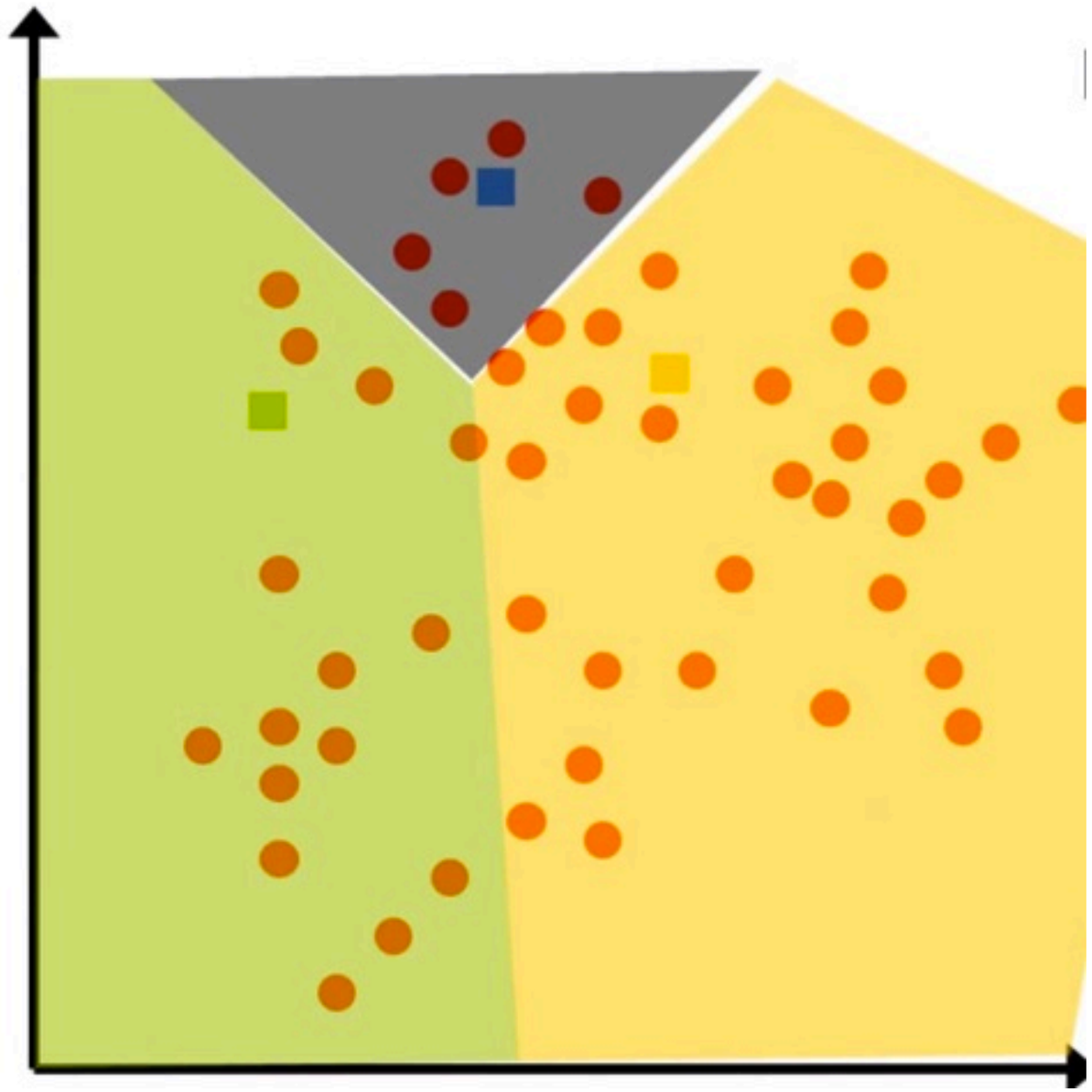
belief propagation

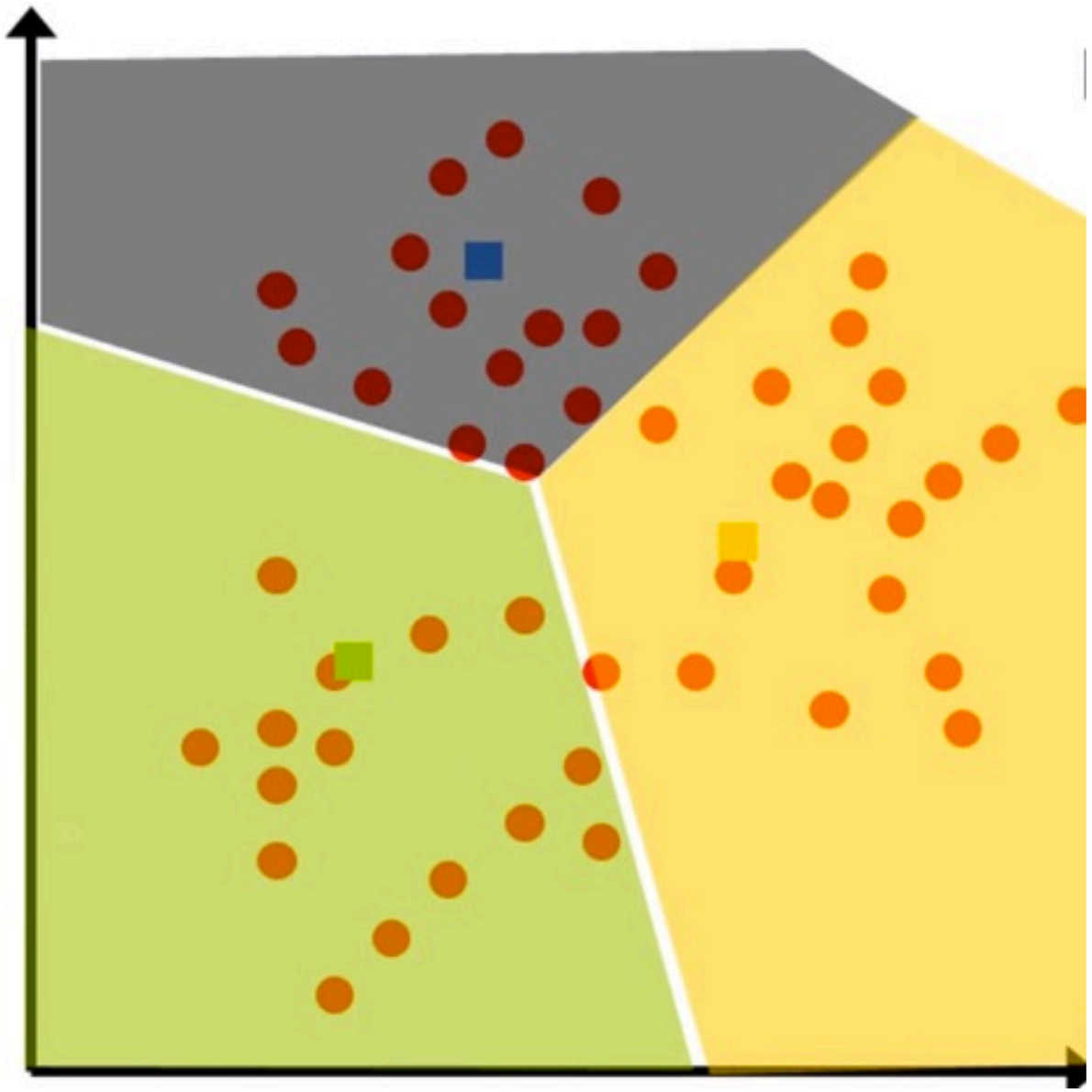
HMMs

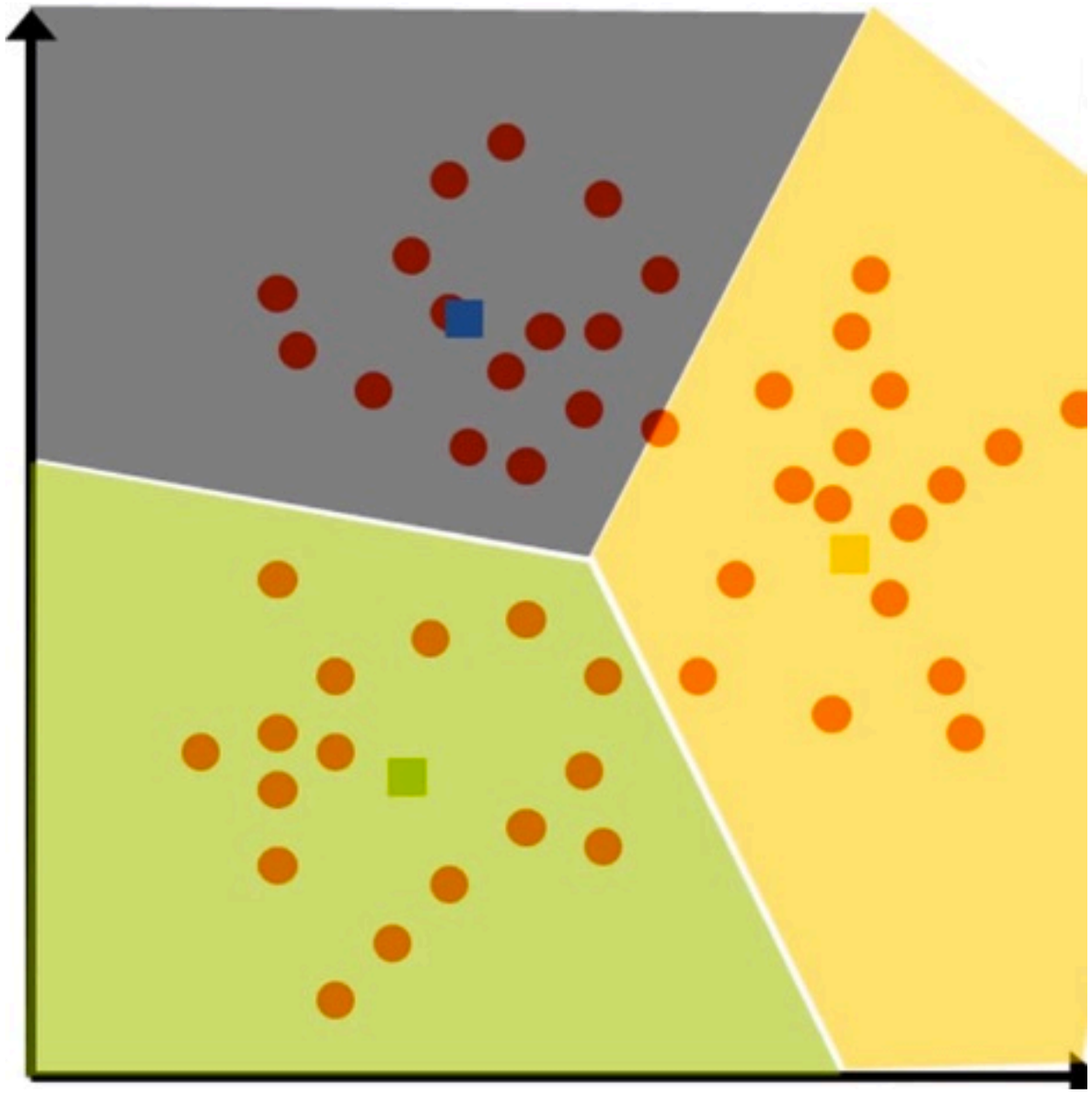
eigendecomposition/SVDs











```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = 0
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = 0
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end

julia> @elapsed distance(a,b)
0.073922868
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = 0
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end
```

```
julia> @elapsed distance(a,b)
0.073922868
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = zero(eltype(x))
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end

julia> @elapsed distance(a,b)
0.001616521
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = zero(eltype(x))
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end
```

```
julia> @elapsed distance(a,b)
0.001616521
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = zero(eltype(x))
    for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end
```

```
julia> @elapsed distance(a,b)
0.001616521
```

```
# euclidean distance between vectors
function distance(x,y)
    @assert length(x) == length(y)
    res = zero(eltype(x))
    @inbounds for i = 1:length(x)
        res += (x[i]-y[i])^2
    end
    return sqrt(res)
end

julia> @elapsed distance(a,b)
0.000273437
```

276x speedup!

```
while !converged && iterations < max_itr
    # loop over all pixels and assign each to a cluster
    for i = 1:pixels
        assignments[i] = assign(data[:,i],centers)
    end

    differences = Array(Float64,size(centers,2))
    # now recalculate the centers
    for i = 1:size(centers,2)
        newcenter = center(assignments,data,i)
        differences[i] = distance(centers[:,i],newcenter)
        centers[:,i] = newcenter
    end

    if !bool(findfirst(x -> x>tol, differences))
        converged = true
    end

    iterations += 1
end
```

```
while !converged && iterations < max_itr
    # loop over all pixels and assign each to a cluster
    for i = 1:pixels
        assignments[i] = assign(data[:,i],centers)
    end

    differences = Array(Float64,size(centers,2))
    # now recalculate the centers
    for i = 1:size(centers,2)
        newcenter = center(assignments,data,i)
        differences[i] = distance(centers[:,i],newcenter)
        centers[:,i] = newcenter
    end

    if !bool(findfirst(x -> x>tol, differences))
        converged = true
    end

    iterations += 1
end
```

```
while !converged && iterations < max_itr
    # loop over all pixels and assign each to a cluster
    for i = 1:pixels
        assignments[i] = assign(data[:,i],centers)
    end

    differences = Array(Float64,size(centers,2))
    # now recalculate the centers
    for i = 1:size(centers,2)
        newcenter = center(assignments,data,i)
        differences[i] = distance(centers[:,i],newcenter)
        centers[:,i] = newcenter
    end

    if !bool(findfirst(x -> x>tol, differences))
        converged = true
    end

    iterations += 1
end
```

```
while !converged && iterations < max_itr
    # loop over all pixels and assign each to a cluster
    for i = 1:pixels
        assignments[i] = assign(data[:,i],centers)
    end

    differences = Array(Float64,size(centers,2))
    # now recalculate the centers
    for i = 1:size(centers,2)
        newcenter = center(assignments,data,i)
        differences[i] = distance(centers[:,i],newcenter)
        centers[:,i] = newcenter
    end

    if !bool(findfirst(x -> x>tol, differences))
        converged = true
    end

    iterations += 1
end
```

problems?

```
for i = 1:size(centers,2)
    newcenter = center(assignments,data,i)
    centers[:,i] = newcenter
end
```

```
for i = 1:size(centers,2)
    newcenter = center(assignments,data,i)
    centers[:,i] = newcenter
end
```

```
for i = 1:size(centers,2)
    newcenter = center(assignments,data,i)
    centers[:,i] = newcenter
end
```

memory allocation!

```
for i = 1:size(centers,2)
    newcenter = center(assignments,data,i)
    centers[:,i] = newcenter
end
```

```
nextcenter = zeros(length(data[:,1]))
for i = 1:size(centers,2)
    center!(nextcenter,assignments,data,i)
    centers[:,i] = nextcenter
end
```

```
julia> @elapsed compress(img,20)
23.605304138
```

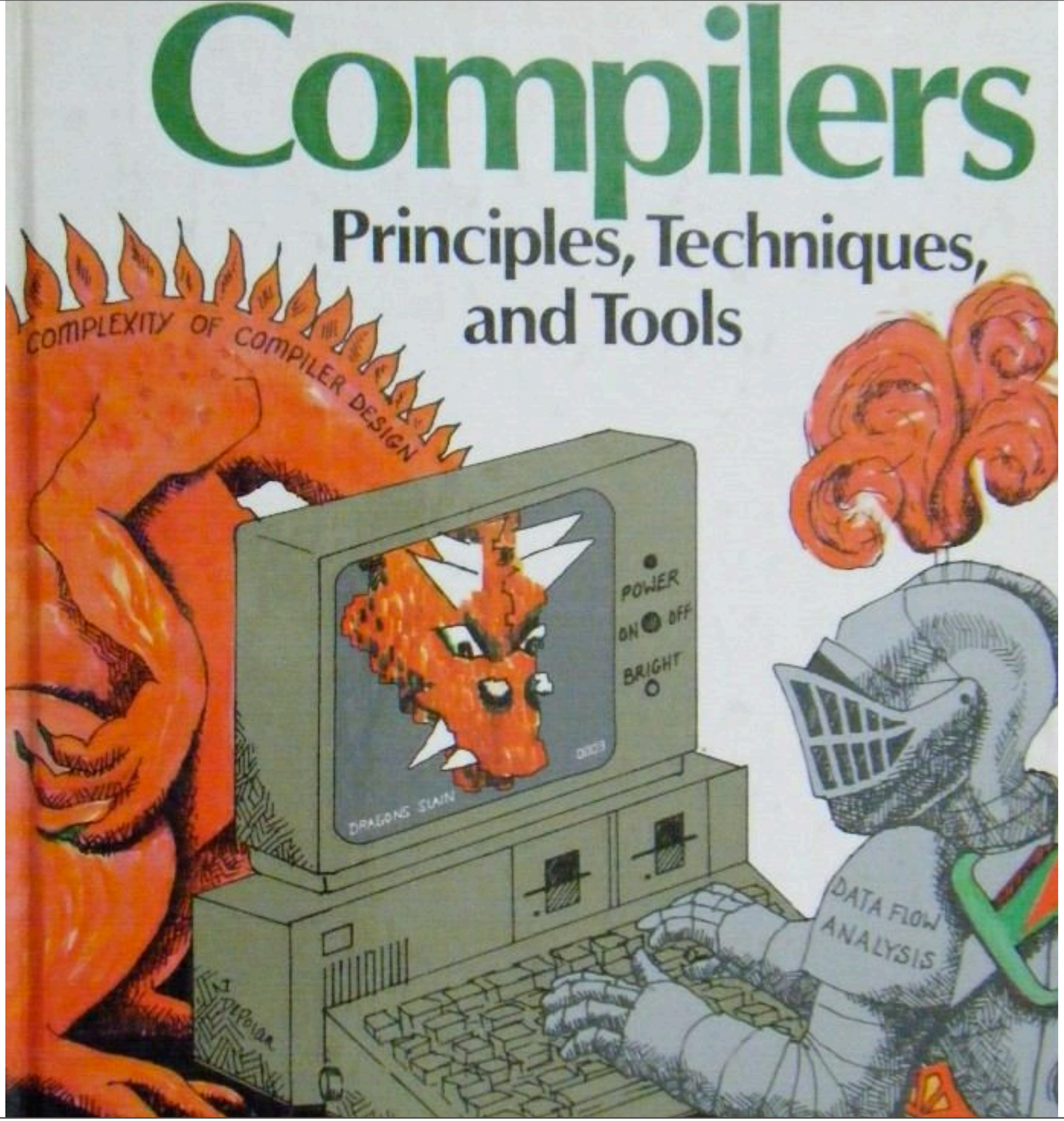
```
julia> @elapsed compress(img,20)
15.409054303
```

```
julia> @elapsed compress(img,20)
15.409054303
```

"why can't the compiler do this for me?"

Compilers

Principles, Techniques, and Tools




Compilers
Principles, Techniques,
and Tools
COMPLEXITY OF COMPILER DESIGN
DRAGON SLAIN
DATA FLOW
ANALYSIS

two kinds

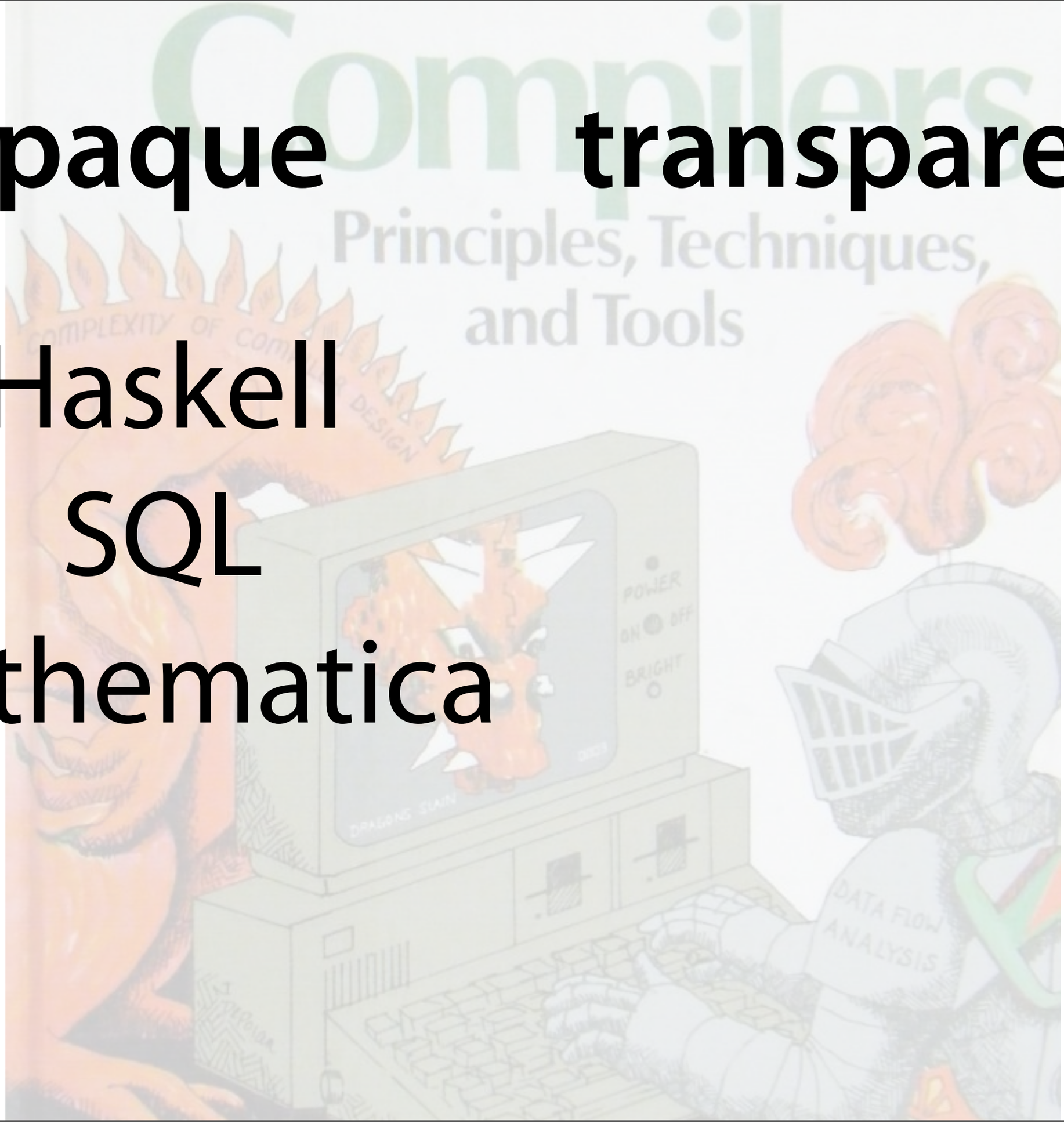
opaque **transparent**

opaque **transparent**

Haskell

SQL

Mathematica

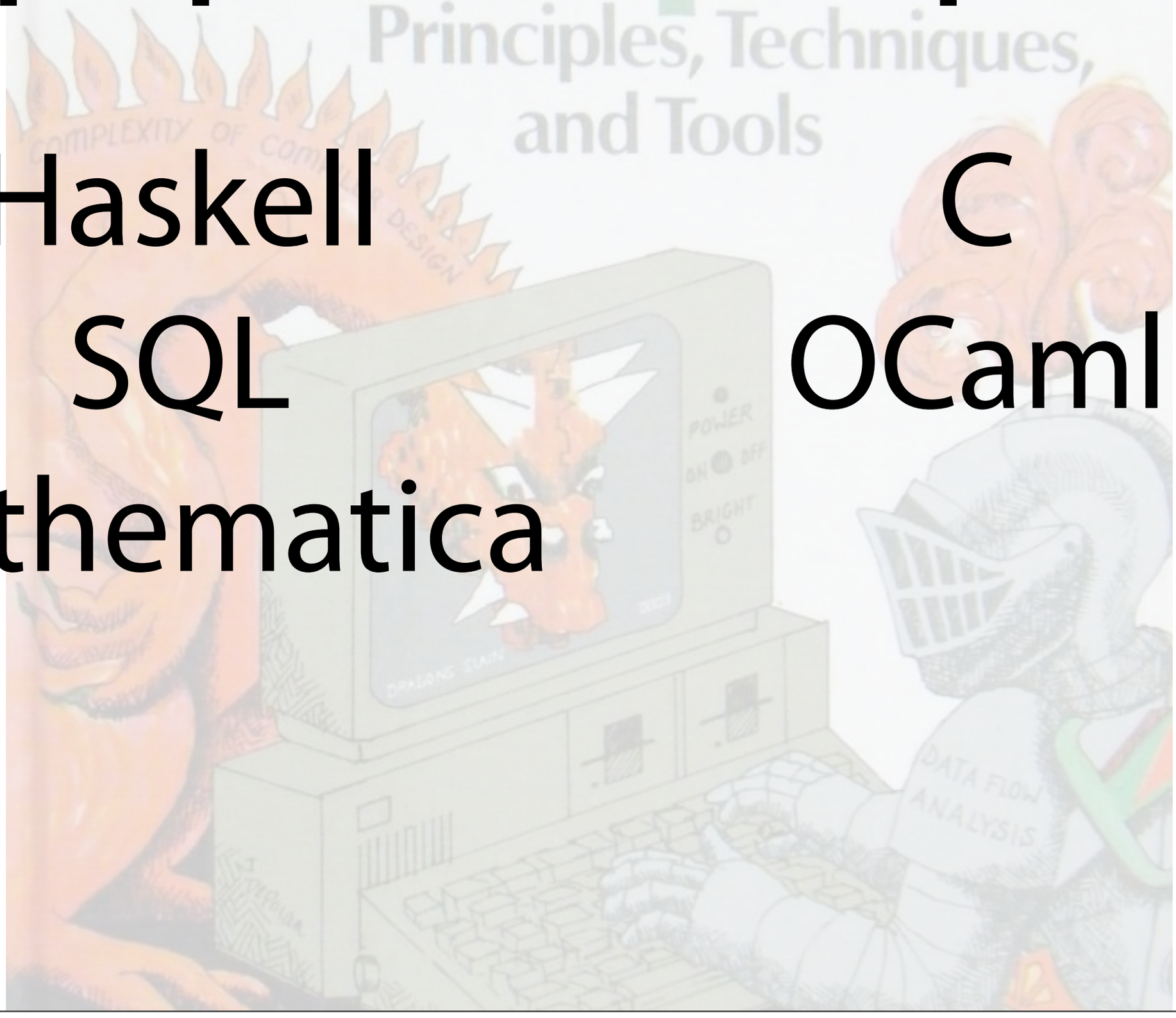


opaque **transparent**

Haskell C

SQL OCaml

Mathematica

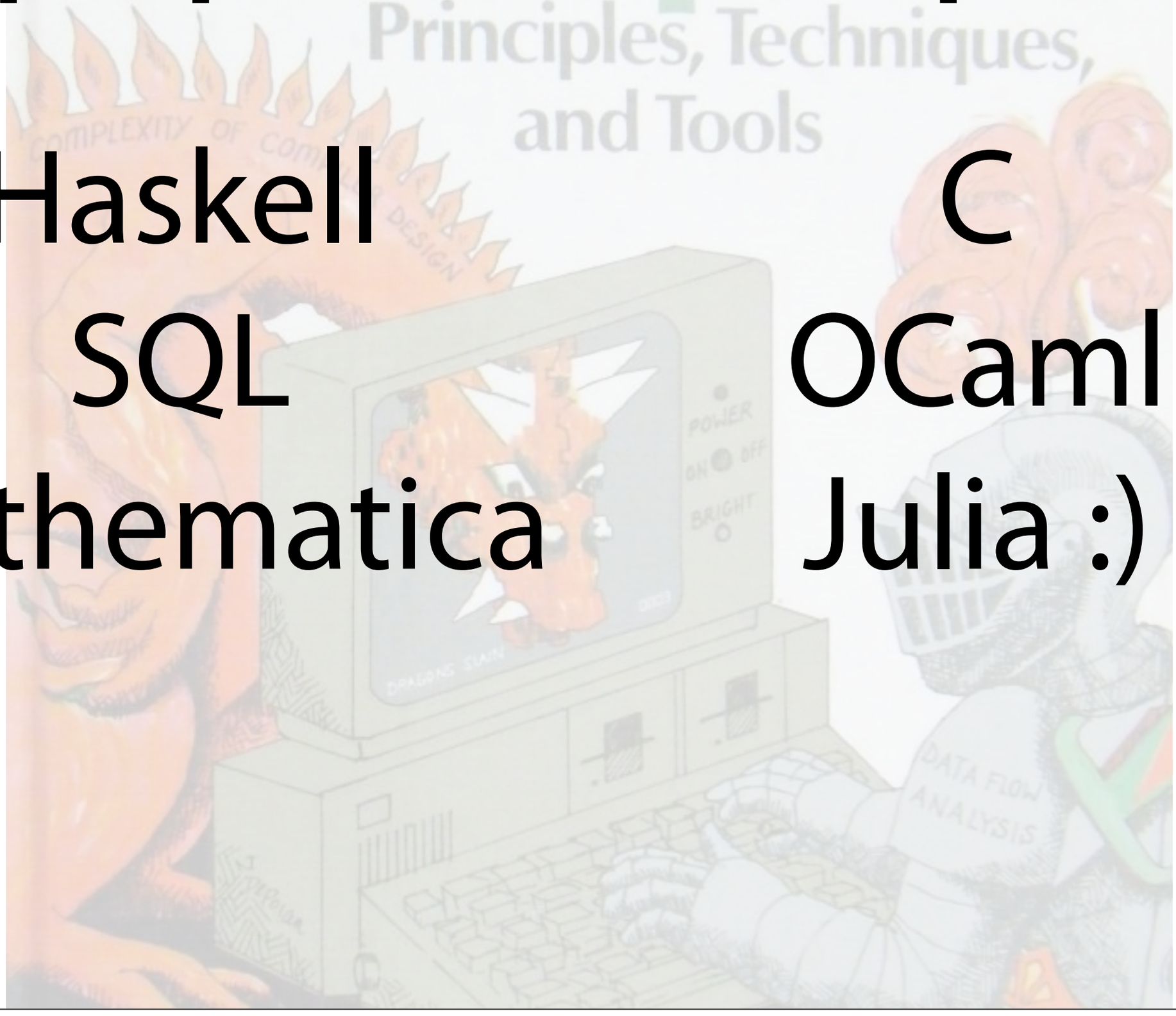


opaque **transparent**

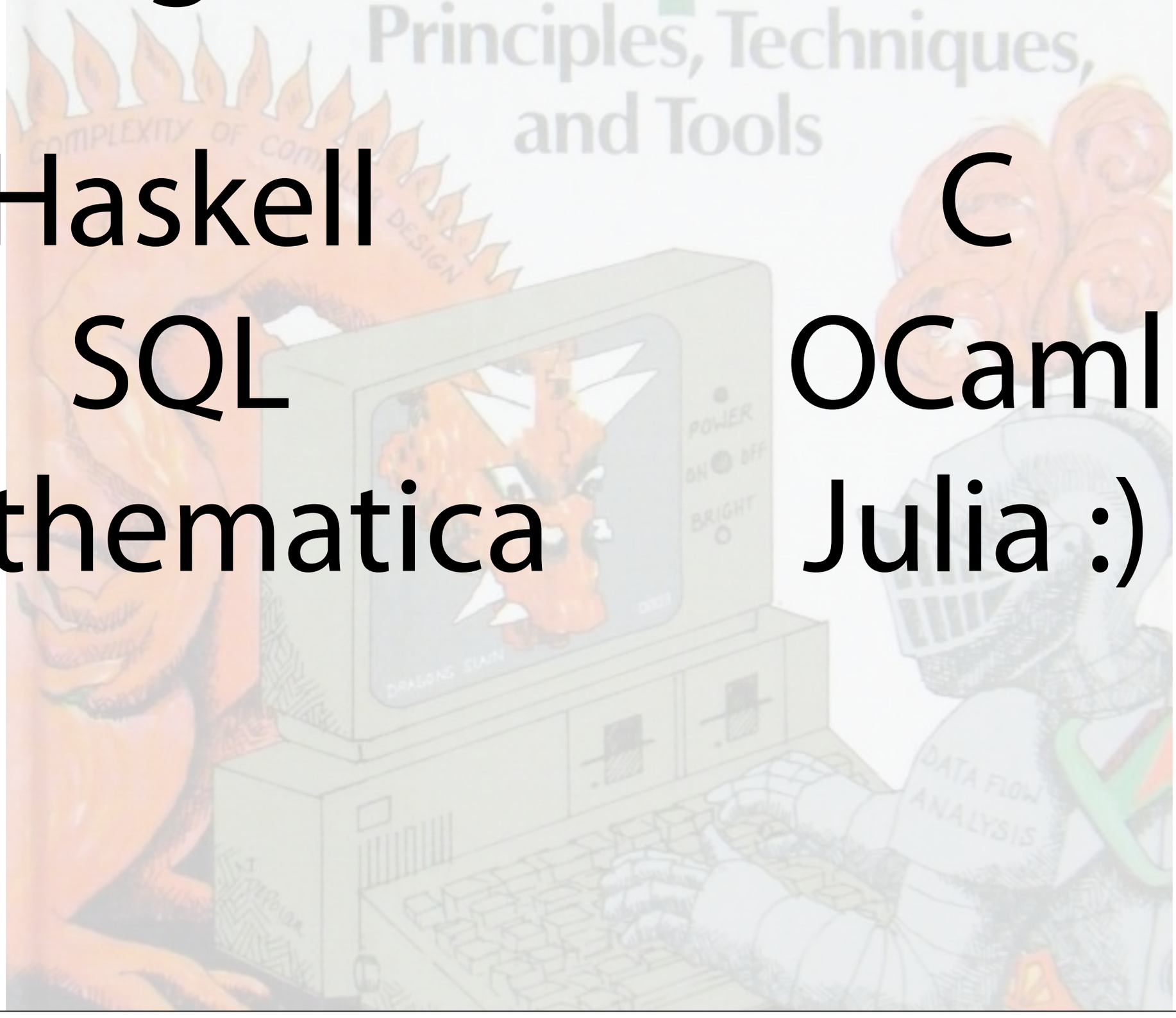
Haskell C

SQL OCaml

Mathematica Julia :)



“magic”	“mundane”
Haskell	C
SQL	OCaml
Mathematica	Julia :)



“declarative” “procedural”

Haskell

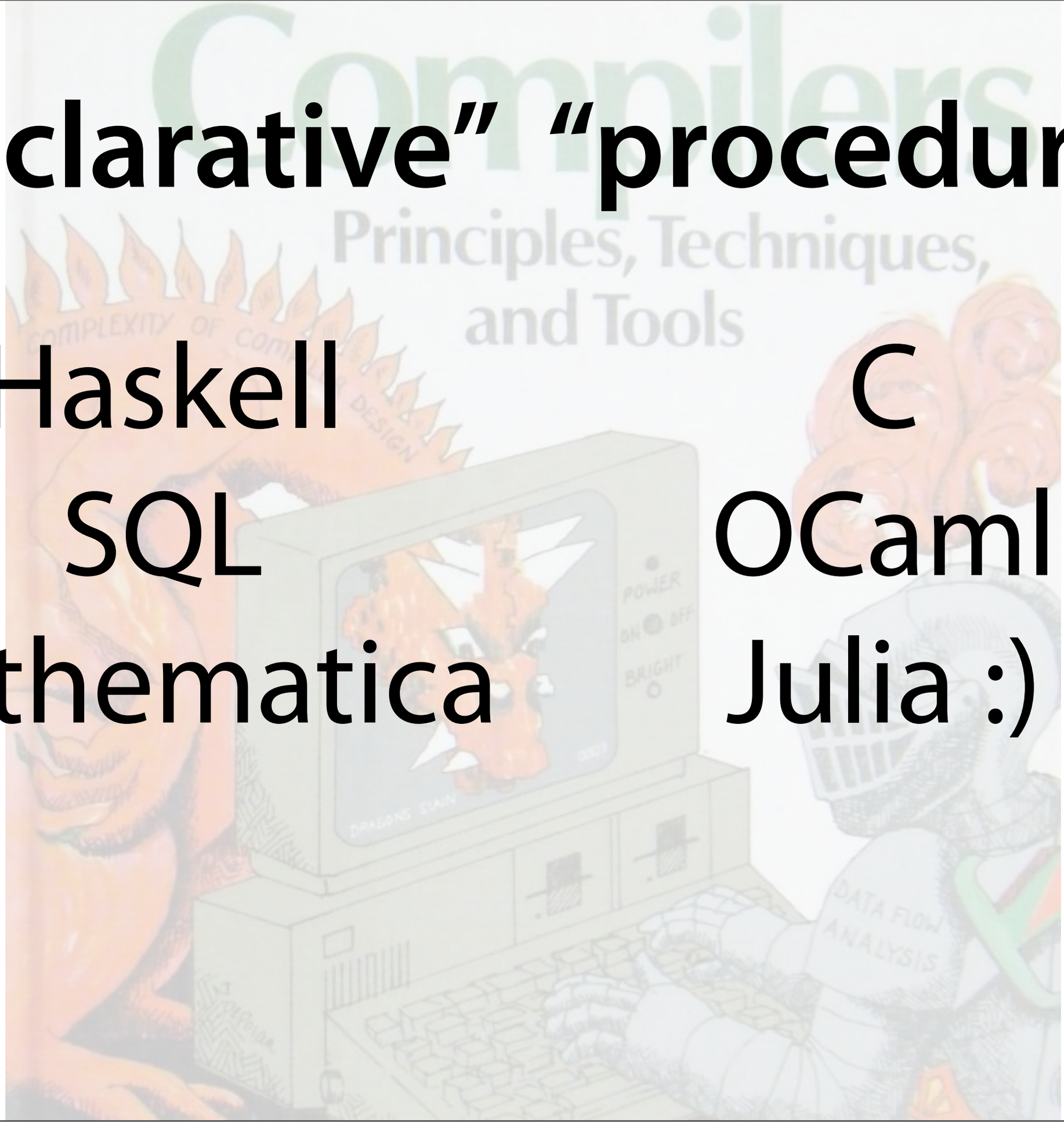
SQL

Mathematica

C

OCaml

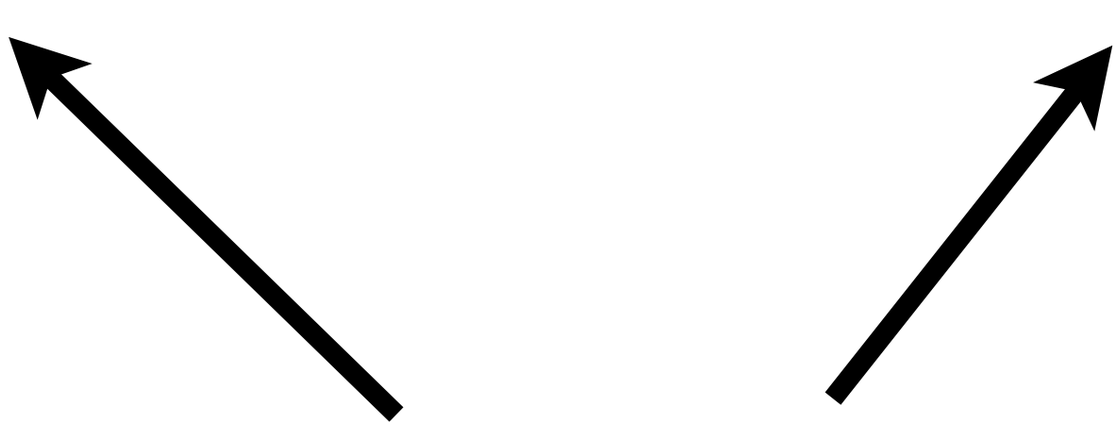
Julia :)



```
result = A .* B + C .* D + A
```

two unnecessary memory allocations

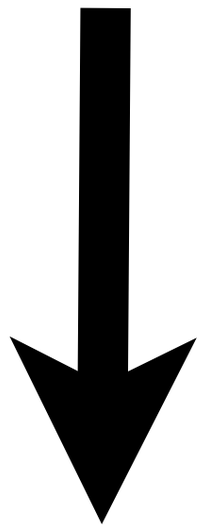
```
result = A .* B + C .* D + A
```



A is traversed twice

```
n = length(a)
r = zeros(n)
for i = 1:n
    r[i] = A[i] * B[i] + C[i] * D[i] + A[i]
end
```

```
r = A .* B + C .* D + A
```



"why can't the compiler
do this for me?"

```
n = length(a)
r = zeros(n)
for i = 1:n
    r[i] = A[i] * B[i] + C[i] * D[i] + A[i]
end
```

simplicity

simplicity

(see https://groups.google.com/d/msg/julia-users/BosGhmT9ei4/ujyLziIA_G4J for JMW on this)

simplicity

vs.

easiness

simplicity

vs.

easiness

(Cartesian.jl, Devectorize.jl, etc.)

THE GOOD

THE BAD

concise, readable syntax
(for array ops, etc.)

concise, readable syntax
(for array ops, etc.)

start from
bad code and optimize

THE GOOD

concise, readable syntax
(for array ops, etc.)

start from
bad code and optimize

THE BAD

optimization can
often be significant work

THE GOOD

concise, readable syntax
(for array ops, etc.)

start from
bad code and optimize

compiler is **simple**

THE BAD

optimization can
often be significant work

THE GOOD

concise, readable syntax
(for array ops, etc.)

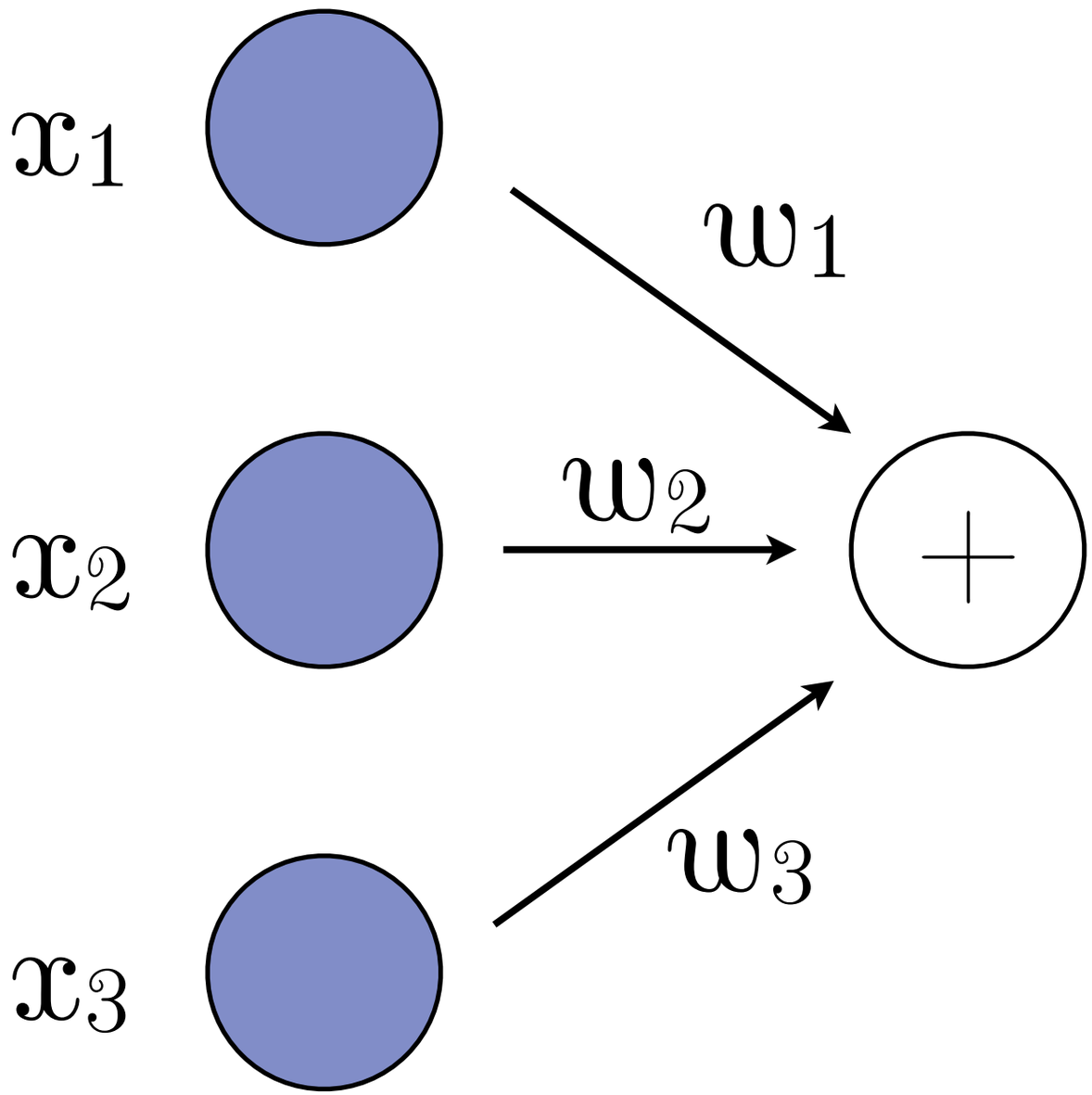
start from
bad code and optimize

compiler is **simple**

THE BAD

optimization can
often be significant work

sometimes getting
performance is not **easy**




```
if result > b
   classify 1
else
   classify -1
end
```

```
yhat = dot(w,data[:,i]) >= 0 ? 1 : -1
if yhat == -1 && labels[i] == 1
    w = w + data[:,i]
    nmistakes += 1
elseif yhat == 1 && labels[i] == -1
    w = w - data[:,i]
    nmistakes += 1
end
```

```
yhat = dot(w,data[:,i]) >= 0 ? 1 : -1
if yhat == -1 && labels[i] == 1
    w = w + data[:,i]
    nmistakes += 1
elseif yhat == 1 && labels[i] == -1
    w = w - data[:,i]
    nmistakes += 1
end
```

```
yhat = dot(w,data[:,i]) >= 0 ? 1 : -1
if yhat == -1 && labels[i] == 1
    w = w + data[:,i]
    nmistakes += 1
elseif yhat == 1 && labels[i] == -1
    w = w - data[:,i]
    nmistakes += 1
end
```

each of these is a copy!

```
yhat = dot(w,slice(data,:,i)) >= 0
    ? 1 : -1
if yhat == -1 && labels[i] == 1
    w = w + slice(data,:,i)
    nmistakes += 1
elseif yhat == 1 && labels[i] == -1
    w = w - slice(data,:,i)
    nmistakes += 1
end
```

```
yhat = dot(w,slice(data,:,i)) >= 0
    ? 1 : -1
if yhat == -1 && labels[i] == 1
    w = w + slice(data,:,i)
    nmistakes += 1
elseif yhat == 1 && labels[i] == -1
    w = w - slice(data,:,i)
    nmistakes += 1
end
```

still not fast :(

lindahua / ArrayViews.jl

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

slow :(

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

UTF8! :)

$$a \neq b$$

$$a \neq b$$

$$a \leq b$$

$$a \neq b$$

$$a \leq b$$

$$a \cdot b$$

$$a \neq b$$

$$a \leq b$$

$$a \cdot b$$

$$a \times b$$

$$a \neq b$$

$$a \leq b$$

$$a \cdot b$$

$$a \times b$$

$$a \cup b$$

(ノ °□°) ノ ⁀┻━━┻ = 3

```
# get initial results
res = pmap(σ -> crossvalidate(data, labels, σ),
           σvals)
```

```
include("mycode.jl")

data = load_data("data_source")

map(do_lots_of_work, data)
```

```
addprocs(4)

@everywhere include("mycode.jl")

data = load_data("data_source")

pmap(do_lots_of_work, data)
```

```
trials = {}
for filename in filenames
    data = readdlm(filename,Int)
    for mu in mus
        noised = noise(data,mu)
        for theta in thetas
            time, denoised = denoise(noised,theta)
            trial = {"file" => filename,
                     "θ" => theta,
                     "μ" => mu,
                     "time" => time}
            push!(trials,trial)
        end
    end
end
```

```
trials = @parallel vcat for filename in filenames
    data = readdlm(filename,Int)
    @parallel vcat for mu in mus
        noised = noise(data,mu)
        @parallel vcat for theta in thetas
            time, denoised = denoise(noised,theta)
            trial = {"file" => filename,
                     "θ" => theta,
                     "μ" => mu,
                     "time" => time}
            trial
        end
    end
end
```

lambdas

lambdas

unicode identifiers

THE GOOD

THE BAD

lambdas

unicode identifiers

simple **and** easy
parallel programming

THE GOOD

lambdas

unicode identifiers

simple **and** easy
parallel programming

THE BAD

. . . that aren't
optimized

THE GOOD

lambdas

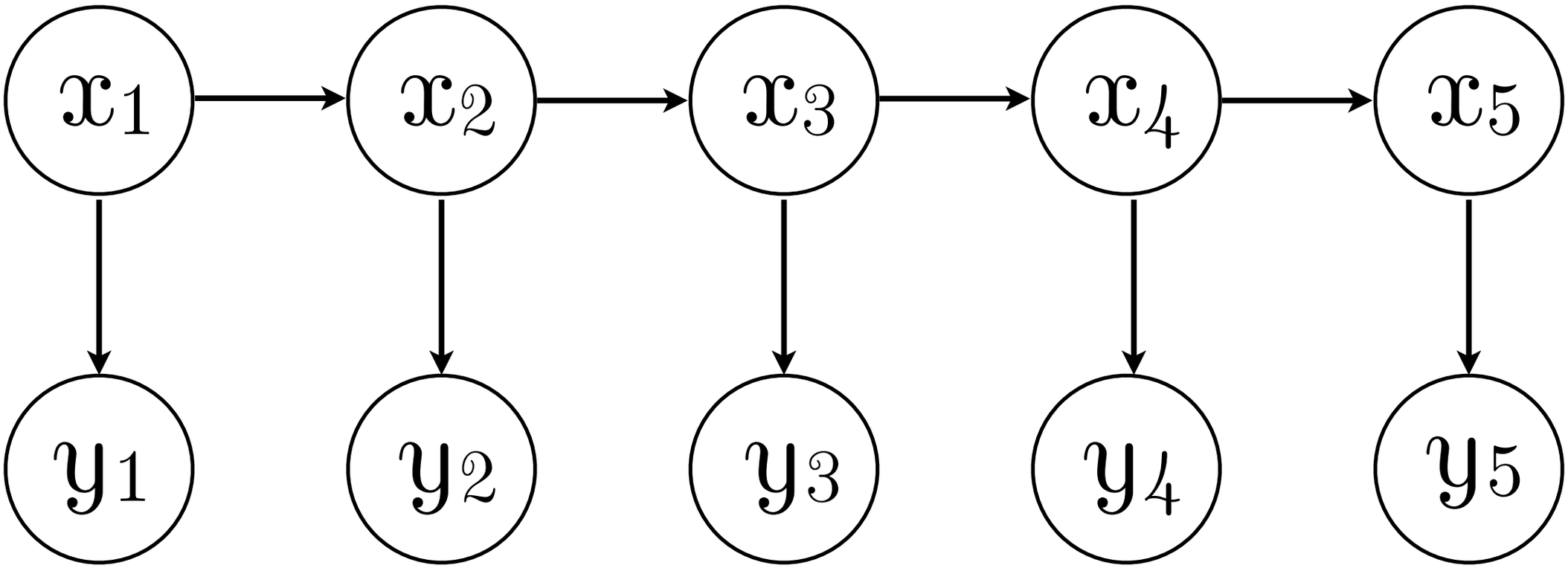
unicode identifiers

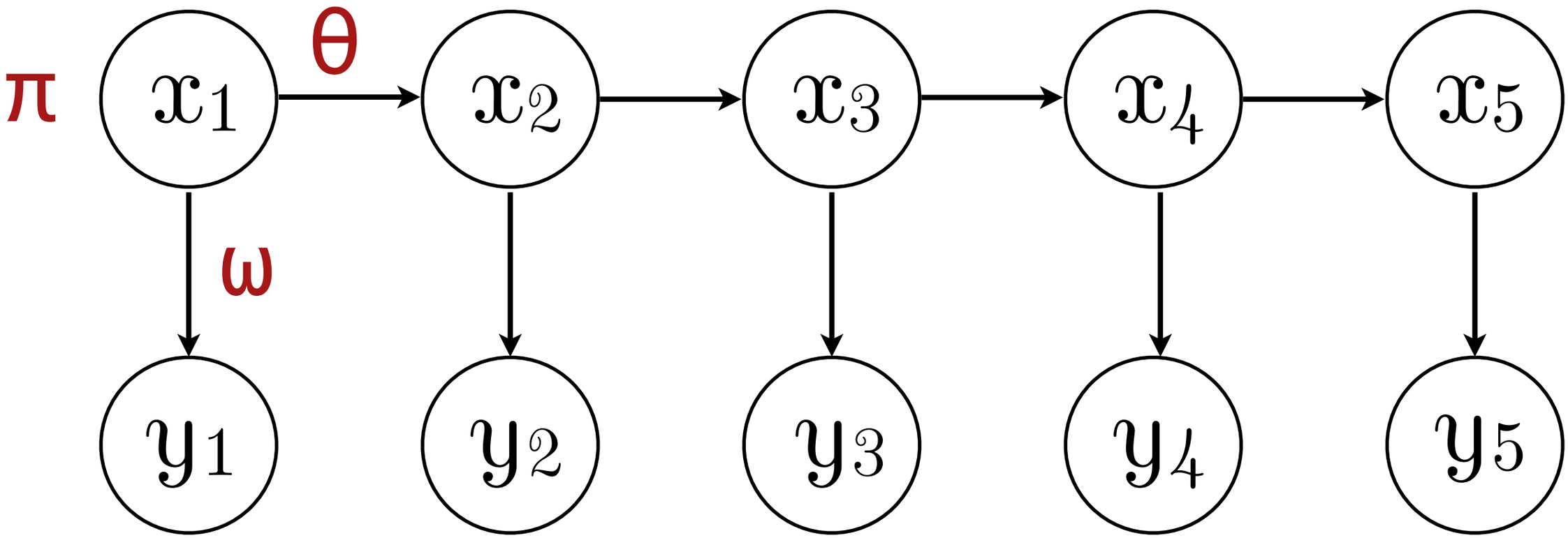
simple **and** easy
parallel programming

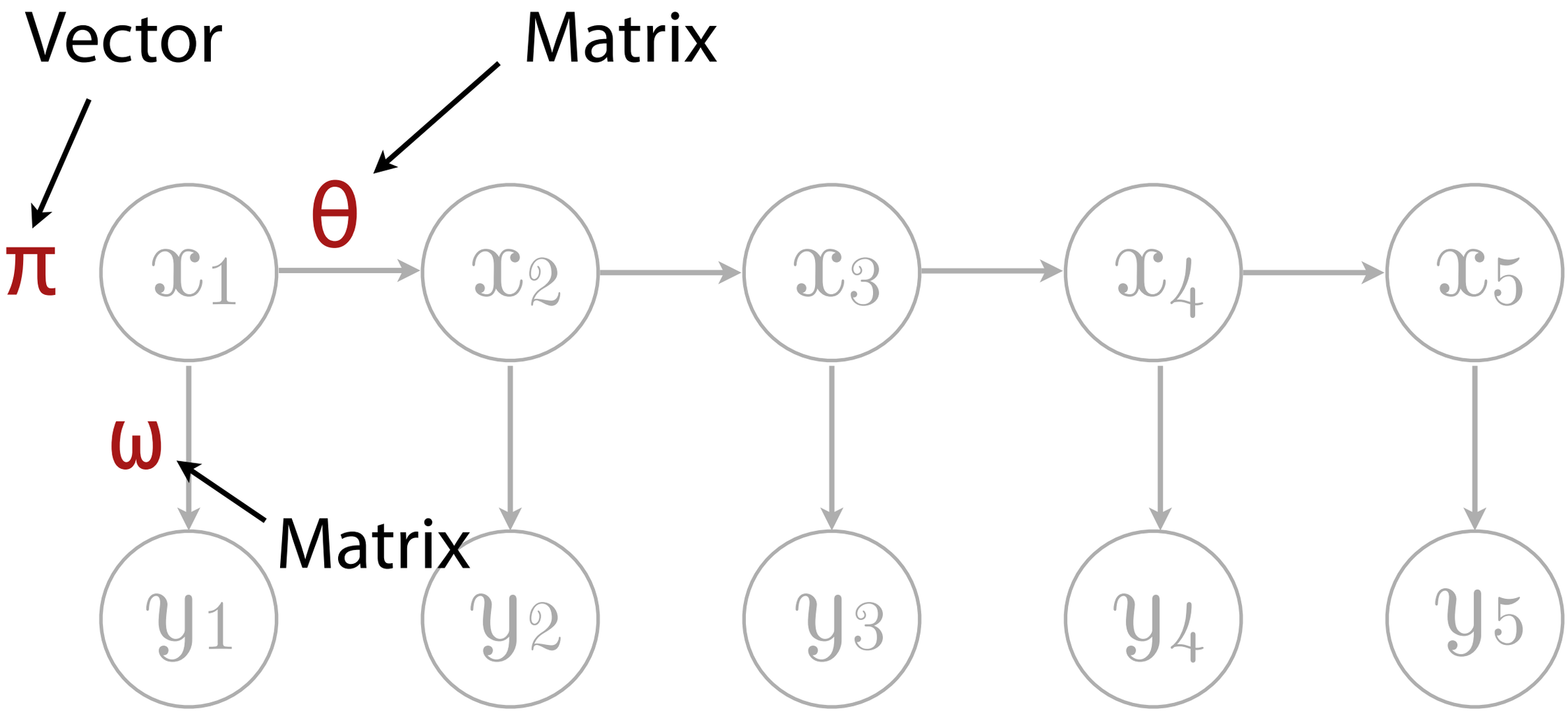
THE BAD

. . . that aren't
optimized

array copying gotchas


x_1
x_2
x_3
x_4
x_5
y_1
y_2
y_3
y_4
y_5


π
θ
ω
x_1
x_2
x_3
x_4
x_5
y_1
y_2
y_3
y_4
y_5



How do we learn these?

How do we learn these?

$$\alpha_t(x_t) =: p(x_t, y_0, \ldots, y_t)$$

Seed: $\alpha_0(x_0) = p(y_0|x_0)\,\pi_{x_0}$

Iterate: $\alpha_t(x_t) = p(y_t|x_t) \sum_{x_{t-1}} p(x_t|x_{t-1})\,\alpha_{t-1}(x_{t-1})$

$$p(x_t|y_0, \ldots, y_t) = \frac{\alpha_t(x_t)}{\sum_{x'_t} \alpha_t(x'_t)}$$

How do we learn these?

$$\beta_t(x_t) = p(y_{t+1}, \ldots, y_T | x_t)$$

Seed: $\beta_T(x_T) = 1$

Iterate: $\beta_t(x_t) = \sum_{x_{t+1}} p(x_{t+1}|x_t)\, p(y_{t+1}|x_{t+1})\, \beta_{t+1}(x_{t+1})$

$$p(x_t|y_0, \ldots, y_T) = \frac{\alpha_t(x_t)\, \beta_t(x_t)}{\sum_{x'_t} \alpha_t(x'_t)\, \beta_t(x'_t)} =: \gamma_t(x_t)$$

How do we learn these?

$$p(x_t, x_{t+1}|y_0, \ldots, y_T) = \frac{1}{Z}\,\alpha_t(x_t)\,p(x_{t+1}|x_t)\,p(y_{t+1}|x_{t+1})\,\beta_{t+1}(x_{t+1}) =$$

$$\frac{\gamma_t(x_t)}{\beta_t(x_t)}\,p(x_{t+1}|x_t)\,p(y_{t+1}|x_{t+1})\,\beta_{t+1}(x_{t+1}) =: \xi_t(x_t, x_{t+1})$$

How do we learn these?

$$\pi_i^{(i+1)} = \gamma_0(i)$$

$$\omega_{i,j}^{(i+1)} = \frac{\sum_{t\,:\,y_t=j} \gamma_t(i)}{\sum_{t=0}^{T} \gamma_t(i)}$$

$$\theta_{i,j}^{(i+1)} = \frac{\sum_{t=0}^{T-1} \xi_t(i,j)}{\sum_{t=0}^{T-1} \gamma_t(i)},$$

it’s hard

it’s hard

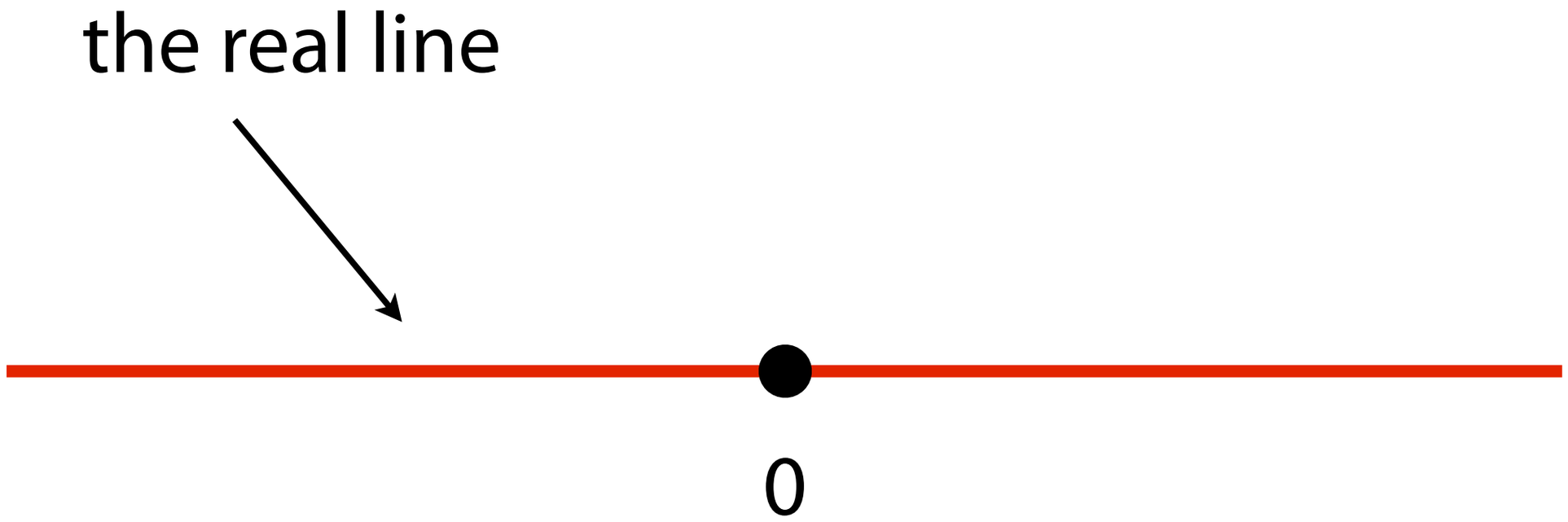
(no matter what tools you use)

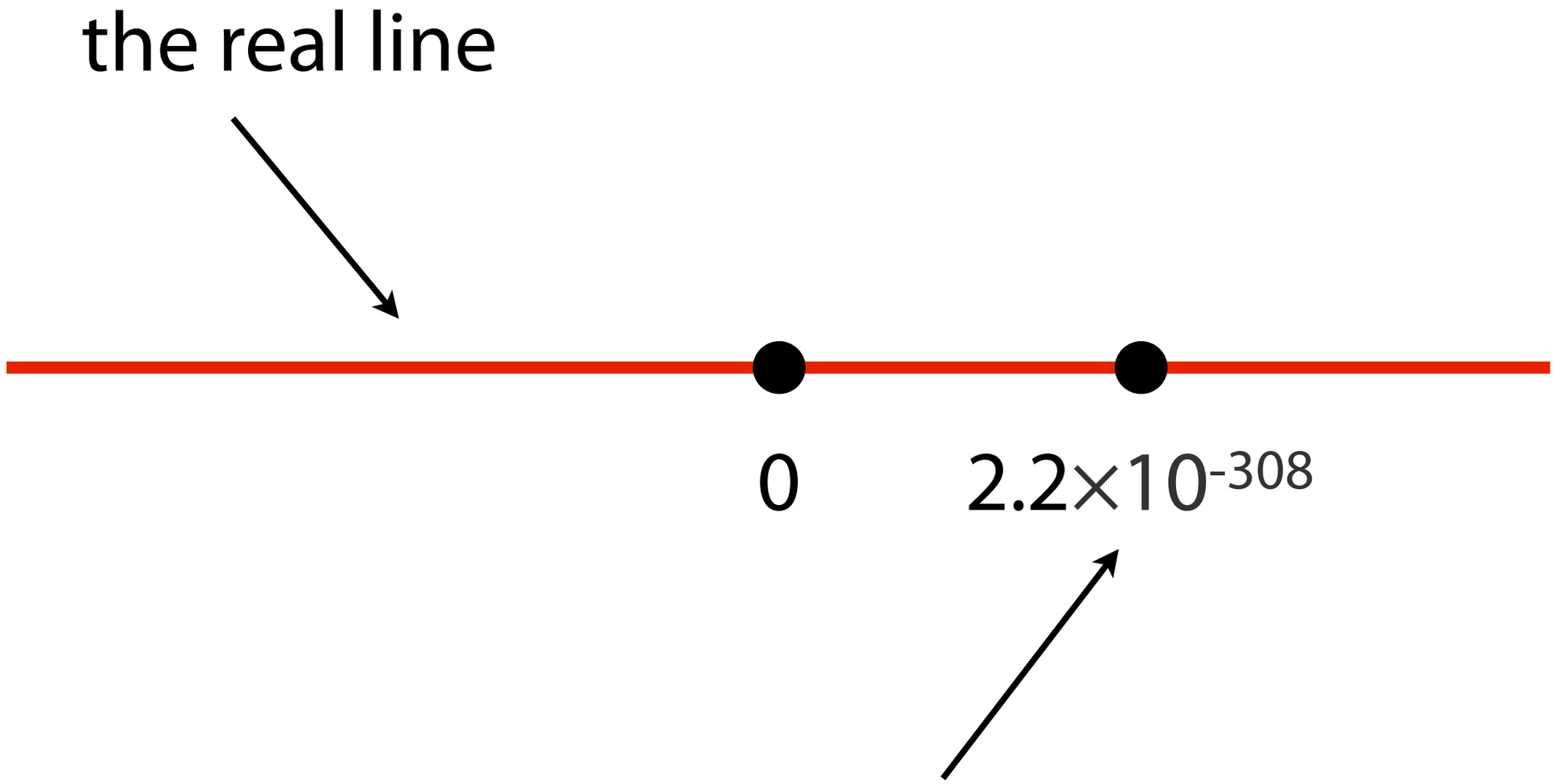
```
type HMM{T <: FloatingPoint}
    nh::Int
    no::Int
    t::Int
    π::Vector{T}
    ω::Matrix{T}
    θ::Matrix{T}
end
```

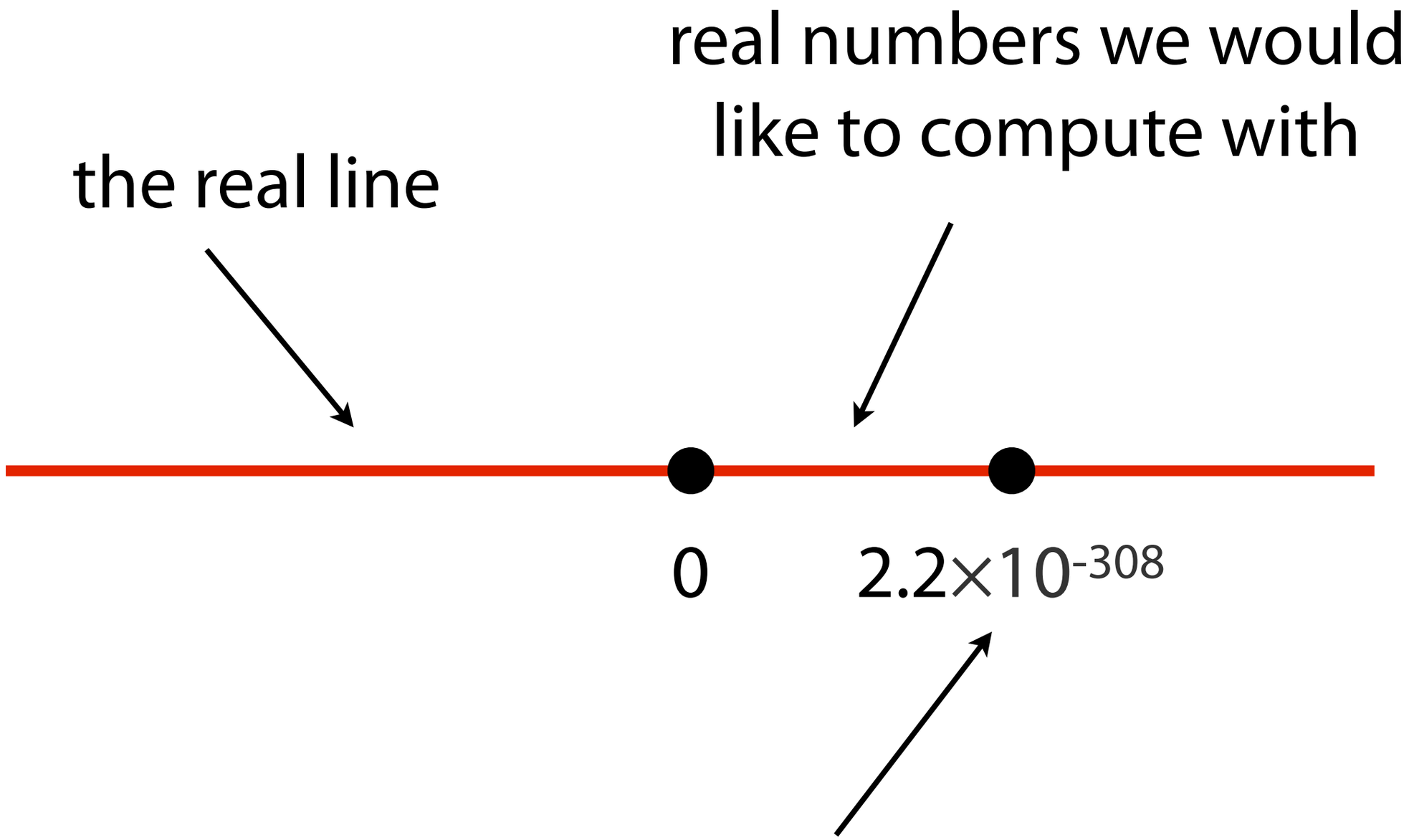
```
type HMM{T <: FloatingPoint}
    nh::Int
    no::Int
    t::Int
    π::Vector{T}
    ω::Matrix{T}
    θ::Matrix{T}
end
```

underflow

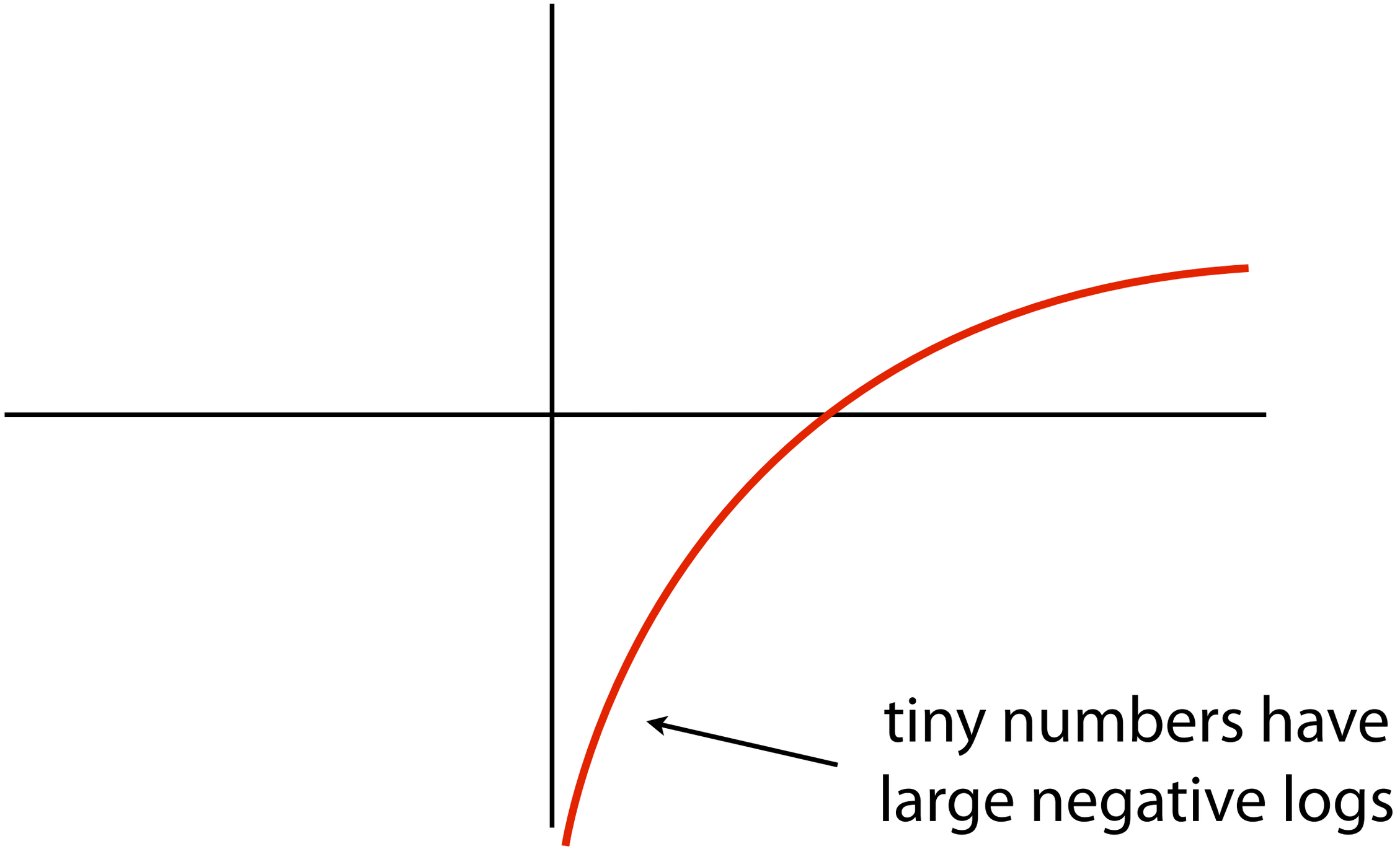
IEEE 754


the real line
0


the real line
0
2.2×10-308
smallest number representable
by 64-bit floats


real numbers we would like to compute with
the real line
0
2.2×10-308
smallest number representable by 64-bit floats

use logs



use logs

$$\log(xy) = \log x + \log y$$

use logs

$$\log(xy) = \log x + \log y$$

$$\log(x + y) =$$

$$\log x + \log(1 + e^{\log y - \log x})$$

```
immutable LogFloat <: FloatingPoint
    n::Float64
end
```

```
Base.convert(::Type{LogFloat},
             f::Float64) = LogFloat(log(f))
```

```
Base.convert(::Type{LogFloat},
             f::Float64) = LogFloat(log(f))

Base.convert(::Type{Float64},
             lf::LogFloat) = exp(lf.n)
```

```
Base.convert(::Type{LogFloat},
             f::Float64) = LogFloat(log(f))

Base.convert(::Type{Float64},
             lf::LogFloat) = exp(lf.n)

Base.promote_rule(::Type{LogFloat},
                  ::Type{Float64}) = LogFloat
```

```
# a LogFloat(n) represents e^n
function Base.show(io::IO,lf::LogFloat)
    n = lf.n
    print(io, "e^$n")
end
```

```
function (*)(lf1::LogFloat,lf2::LogFloat)
    return LogFloat(lf1.n + lf2.n)
end
```

```
function (*)(lf1::LogFloat,lf2::LogFloat)
    return LogFloat(lf1.n + lf2.n)
end

function (/)(lf1::LogFloat,lf2::LogFloat)
    return LogFloat(lf1.n - lf2.n)
end
```

```
function (*)(lf1::LogFloat,lf2::LogFloat)
    return LogFloat(lf1.n + lf2.n)
end

function (/)(lf1::LogFloat,lf2::LogFloat)
    return LogFloat(lf1.n - lf2.n)
end

function (+)(lf1::LogFloat, lf2::LogFloat)
    if lf1.n > lf2.n
        x = lf1.n
        y = lf2.n
    else
        x = lf2.n
        y = lf1.n
    end
    LogFloat(x + log1p(exp(y-x)))
end
```

```
function (<)(lf1::LogFloat,lf2::LogFloat)
    return lf1.n < lf2.n
end
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

$$\frac{\alpha_t(x_t)\,\beta_t(x_t)}{\sum_{x'_t}\alpha_t(x'_t)\,\beta_t(x'_t)} =: \gamma_t(x_t)$$

```
# gamma[i,t] = state i ,time t
gamma(α,β) = broadcast(/, α .* β, sum(α .* β,1))
```

```
# gamma[i,t] = state i ,time t
function gamma(α,β)
    ret = α .* β
    # sums[t] is sum at time t over all states
    sums = sum(ret,1)
    for i in 1:size(ret,1)
        for j in 1:size(ret,2)
            ret[i,j] = ret[i,j] / sums[j]
        end
    end
    return ret
end
```

```
hmm = HMM{LogFloat}(20,27,1000)

train!(text, hmm)
```

```
correct(hmm, corrupted_text)

sample(hmm, n)
```

type system /
numeric promotions

THE GOOD

type system /
numeric promotions

easy to map from
math onto code
(readability)

THE BAD

THE GOOD

type system /
numeric promotions

easy to map from
math onto code
(readability)

THE BAD

no debugger

THE GOOD

THE BAD

THE GOOD > THE BAD

questions?

@porterjamesj

github.com/porterjamesj/254

porterjamesj@gmail.com