BIO2045 – Séance 4

Optimisation par colonies de fourmis

Contenu

Concepts principaux

Commentaires

Dans les séances précédentes, nous avons vu qu'utiliser des noms de variables explicites permet de rendre le code plus lisible, et donc plus facile a modifier.

Dans certains cas, il est important d'expliciter certaines parties du code en utilisant des commentaires. Ces commentaires servent une fonction différente: ils permettent de clarifier *ce que le code doit faire*.

Par exemple, le code suivant n'est pas une bonne utilisation des commentaires:

```
import Random
Random.seed!(2045)
```

```
Random.TaskLocalRNG()
```

```
# On génère trois valeurs aléatoires
a = rand(3)
```

```
# Et on les multiplie par 2π et ensuite on enlève π
b = map(x -> (x * 2π) - π, a)
```

```
3-element Vector{Float64}:
 -0.7777327717072757
  1.938788160738743
 -2.665731441064061
```

Le même code avec des commentaires utiles serait

```
# On veut générer des points qui soient disposés de manière
aléatoire sur le périmètre d'un cercle.

# On commence par gnérer des valeurs aléatoires de manière
uniforme entre 0 et 1
a = rand(3)
# On projette ensuite ces valeurs entre -π et π pour que ces
valeurs soient uniformément réparties mais deviennent des angles
b = map(x -> (x * 2π) - π, a)
```

```
3-element Vector{Float64}:
 -1.3115960127833683
  1.3325356560364812
  2.810004475302713
```

Notez que le code est le même, mais que la compréhension du code est facilité dans la deuxième version: le code lui-même indique ce qu'on fait, et les commentaires indiquent *pourquoi* on le fait.

NB: dans les prochains devoirs, nous utiliserons le package Literate pour transformer du code commenté en documents. Dans ce contexte la, les commentaires qui devront *rester* dans le code seront notés par ##.

Documentation

Les fonctions que nous allons écrire peuvent (doivent!) avoir leur propre documentation. La documentation d'une fonction se compose au minimum de trois choses:

1. le *prototype* de la fonction
2. une explication de ce que la fonction permet de faire
3. une explication des arguments de la fonction

Toutes ces informations forment le *docstring* de la fonction, et s'organisent de la manière suivante:

```
"""
    moyenne(x)

Cette fonction calcule la moyenne arithmétique d'un vecteur de
nombres `x` donné
en argument.

`x` doit être un vecteur de nombres.
"""
function moyenne(x)
    somme = sum(x)
    n = length(x)
    return somme / n
end
```

```
Main.var"##318".moyenne
```

Cette documentation est ensuite accessible *via* `?moyenne`. Plus d'information sur les *docstring* est disponible dans le manuel de Julia, dont le lien est disponible dans le plan de cours.

Passage par référence et modification des arguments

À l'inverse de langages de programmation comme R, Julia évite de dupliquer des objets quand les fonctions ont besoin de les utiliser. Cela rend le code plus efficace, mais crée des complications possibles.

Prennons la matrice suivante:

```
V = rand(1:5, 3, 6)
```

```
3×6 Matrix{Int64}:
 4  3  3  2  3  5
 4  2  2  2  4  4
 4  3  4  2  2  3
```

Et la fonction suivante:

```
function modif(X)
    X[1,1] = rand(30:50)
    return X
end
```

```
modif (generic function with 1 method)
```

Si on applique cette fonction à `V`, on obtient

```
modif(V)
```

```
3×6 Matrix{Int64}:
 45  3  3  2  3  5
  4  2  2  2  4  4
  4  3  4  2  2  3
```

Si on affiche maintenant la matrice `V`, on obtient

```
V
```

```
3×6 Matrix{Int64}:
 45  3  3  2  3  5
  4  2  2  2  4  4
  4  3  4  2  2  3
```

Regardez bien la première position de cette matrice: `V` a été *modifié* par la fonction `modif`. C'est parce que cette fonction n'a pas reçu une copie de `V`, mais bien la matrice elle-même.

Pour indiquer qu'une fonction modifie son premier argument, on utilise la notation `!`:

```
function modif!(X)
    X[1,1] = rand(50:70)
    return X
end
```

```
modif! (generic function with 1 method)
```

Cette notation est beaucoup plus explicite, et indique qu'un ou plusieurs arguments vont être modifiés par la fonction (et on sait écrire la documentation pour indiquer lesquels, et comment!).

Si on souhaite ne pas modifier la matrice `V`, il existe un *design pattern* très commun en Julia:

```
function modif(X)
    Y = copy(X)
    modif!(Y)
    return Y
end
```

```
modif (generic function with 1 method)
```

Cette nouvelle fonction va d'abord *copier* `X`, puis modifier cette copie à l'aide de la fonction qui modifie son argument, et retourner la copie modifiée.

On parle en général de `modif` comme *non-mutating*, et de `modif!` comme *mutating*.

Application d'une fonction sur une collection

La semaine dernière, nous avons introduit `map` pour appliquer une fonction sur une collection. Il existe une syntaxe plus simple pour la vectorisation (qui est décrite dans le manuel dans la section sur le *broadcasting*):

```
(x -> 2*x).(rand(3))
```

```
3-element Vector{Float64}:
 0.511541964639278
 0.9665160151825205
 1.1374075092448384
```

La notation `fonction.(argument)` va appliquer `fonction` à chaque élément de la collection `argument`. Nous allons utiliser cette notation très souvent dans les séances suivantes.

Modèle

```
using CairoMakie
using Statistics
using StatsBase
CairoMakie.activate!(px_per_unit = 6.0)
```

LOL

```
function disposition_points(n)
    angles = rand(n) .* 2π
    radii = sqrt.(rand(length(angles)) .* 3.0 .+ 1.0)
    x = cos.(angles) .* radii
    y = sin.(angles) .* radii
    stops = permutedims(hcat(x, y))
    return stops
end
```

```
points = 100

P = zeros(Float64, points, points)
D = zeros(Float64, points, points)

xy = disposition_points(points)

for i in 1:points
    for j in 1:points
        D[i,j] = sqrt(sum((xy[:,i] .- xy[:,j]) .^2.0))
    end
end

scatter(xy)

function walk_on_graph(D, P, i)
    α = 0.9
    β = 1.5
    n_sites = size(D, 1)

    visites = zeros(Bool, n_sites)
    visites[i] = true

    chemin = [i]

    while sum(visites) != n_sites
        voisins = Int64[]
        cible = Float64[]
        for voisin in setdiff(1:n_sites, chemin)
            pheromones = max(P[last(chemin), voisin], 1e-5)
            poids = (pheromones^α)/D[last(chemin), voisin]^β
            push!(cible, poids)
            push!(voisins, voisin)
        end

        next_site = sample(voisins, Weights(cible))
        push!(chemin, next_site)
        visites[next_site] = true
    end
    return chemin
end

"""
    chemin_distance(chemin, D)

Distance totale du chemin, incluant le retour au point initial

- `chemin`: vecteur de positions qui indique l'ordre de visite
des sites
- `D`: matrice de distance entre les sites
"""
function chemin_distance(chemin, D)
```

```
ERROR: ParseError:
```

```
# Error @ /home/tpoisot/Documents/SimulerLeVivant/
output/04_fourmis.md:60:36
"""
function chemin_distance(chemin, D)
#                                  └ ── premature end of input
```

d représente la distance pour revenir au début du cycle

on veut créer un circuit, et le chemin s'arrête au dernier site visité

le chemin indique l'ordre de visite des sites

D est la matrice de distance entre tous les points

```
    d = D[chemin[end], chemin[begin]]
```

```
ERROR: UndefVarError: `chemin` not defined in `Main.var"##318"`
Suggestion: check for spelling errors or missing imports.
```

on va calculer la distance totale du reste de chemin on a deja la distance du retour vers le premier site

```
    for i in 2:length(chemin)
```

```
ERROR: ParseError:
# Error @ /home/tpoisot/Documents/SimulerLeVivant/
output/04_fourmis.md:1:30
    for i in 2:length(chemin)
#                            └ ── premature end of input
```

on lit dans D la distance entre le site visité à la position i et le site visité juste avant (position i-1)

on ajouter cette distance à d

```
        d += D[chemin[i-1], chemin[i]]
    end
```

```
ERROR: UndefVarError: `d` not defined in `Main.var"##318"`
```

```
Suggestion: add an appropriate import or assignment. This global
was declared but not assigned.
```

d contient la distance totale du cycle distance du chemin + distance du retour au premier point on renvoie d

```
    return d
end

function pheromones!(P, chemin, D)
    Q = size(P, 1)
    score = Q / chemin_distance(chemin, D)
    P[chemin[end], chemin[1]] += score
    for i in 2:length(chemin)
        P[chemin[i-1], chemin[i]] += score
    end
    return P
end

track = zeros(Float64, 120)
n_fourmis = 50
evaporation_rate = 0.99

@showprogress for i in 1:length(track)

    chemins = [walk_on_graph(D, P, rand(1:points)) for _ in
1:n_fourmis]
```

```
ERROR: UndefVarError: `d` not defined in `Main.var"##318"`
Suggestion: add an appropriate import or assignment. This global
was declared but not assigned.
```

Remove the chemins with more than the median chemin_distance

```
    distances = [chemin_distance(chemin, D) for chemin in
chemins]
    median_distance = median(distances)
    chemins = chemins[findall(distances .<= median_distance)]

    for chemin in chemins
        pheromones!(P, chemin, D)
    end
    P ./= length(chemins)
    #P .+= rand(size(P)).*0.02 .- 0.01
    P .*= evaporation_rate

    track[i] = minimum(distances)

end
```

```
ERROR: UndefVarError: `chemins` not defined in `Main.var"##318"`
Suggestion: check for spelling errors or missing imports.
```

Plotting the points and lines colored by the value of P

```
fig = Figure(size=(600, 680))
gl = fig[1,1] = GridLayout()
ax = Axis(gl[1, 1], aspect=1)
ax2 = Axis(gl[2,1], yscale=sqrt)
lines!(ax2, track, color=:purple, linewidth=2)
hidedecorations!(ax2)

scatter!(ax, xy[1, :], xy[2, :], color=:black)
hidespines!(ax)
hidedecorations!(ax)

rowsize!(gl, 2, Relative(0.2))
fig
```

```
ERROR: UndefVarError: `track` not defined in `Main.var"##318"`
Suggestion: check for spelling errors or missing imports.
```