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"##356".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.

Simulation

Nous allons créer un modèle qui utilise la métaphore des colonies de fourmis pour résoudre un problème d'optimisation.

```
using CairoMakie
using Statistics
```

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

Nous allons tenter de résoudre le problème suivant: si on dispose d'une liste de points, dans l'espace, quel est le trajet qui permet de visiter l'ensemble de ces points et de revenir au départ, en faisant le trajet le plus court possible?

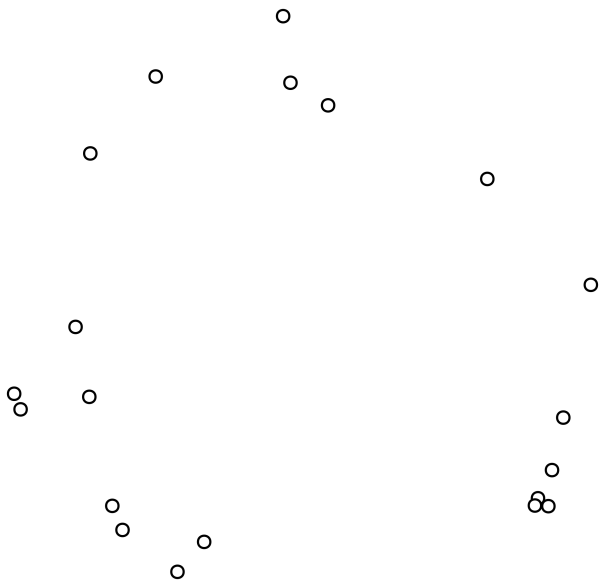
Pour commencer, nous allons générer des points qui sont disposés uniformément sur un cercle, avec une distance au centre entre 5 et 10 unités.

```
function disposition_points(n)
    angles = rand(n) .* 2π
    radii = sqrt.(rand(length(angles)) .* 5 .+ 5)
    x = cos.(angles) .* radii
    y = sin.(angles) .* radii
    stops = permutedims(hcat(x, y))
    return stops
end
```

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

```
points = 20
xy = disposition_points(points);
```

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
hidespines!(ax)
hidedecorations!(ax)
f
```



L'objectif est donc de trouver un chemin qui parte d'un point, visite tous les points, et retourne au point de départ. Le *challenge* est de trouver le chemin qui accumule la plus petite distance totale.

Mesure de la distance entre les points

La distance entre les points est constante, et nous allons donc la calculer une seule fois. Puisque nous avons un nombre de points n, il faut préparer une matrice de taille $n \times n$.

```
D = zeros(Float64, points, points);
```

Nous allons donc stocker la distance entre les points dans cette matrice:

```
function distance_entre_points!(D, xy)
    for i in axes(xy, 2)
        for j in axes(xy, 2)
            D[i, j] = sqrt(sum((xy[:, i] .- xy[:, j]) .^ 2.0))
        end
    end
    return D
end
```

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

NB: vous devrez regarder la documentation de `axes`.

On va ensuite appliquer cette fonction.

```
distance_entre_points!(D, xy);
```

Notez qu'on n'a pas besoin de stocker le résultat de cette opération: la matrice `D` est modifiée directement!

Représenter un chemin

Un chemin est une séquence de points, tous uniques, qui représente un circuit complet dans la liste des points. On prend le premier point au hasard, puis on choisit ensuite chaque point parmis la liste de ceux qui n'ont pas encore été visités:

```
function chemin_aleatoire(xy)
    pool = collect(axes(xy, 2))
    chemin = []
    for i in axes(xy, 2)
        push!(chemin, popat!(pool, rand(eachindex(pool))))
    end
    push!(chemin, first(chemin))
    return chemin
end
```

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

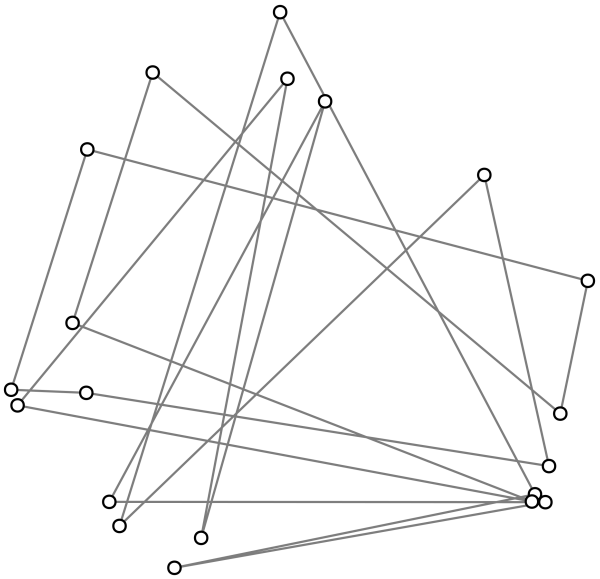
NB: vous devrez regarder la documentation de `collect` et de `pop!` / `popat!`

```
C = chemin_aleatoire(xy);
```

On peut maintenant visualiser ce chemin

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
for i in 2:length(C)
    linesegments!(ax, xy[1, [C[i-1], C[i]]], xy[2, [C[i-1],
C[i]]], color=:grey50)
end
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
```

```
hidespines!(ax)
hidedecorations!(ax)
f
```



Ce chemin n'est pas optimal – il est assez long. Quelle est sa longueur?

```
function distance_chemin(C, D)
    d = 0.0
    if last(C) != first(C)
        push!(C, first(C))
    end
    for i in 2:length(C)
        d += D[C[i-1], C[i]]
    end
    return d
end
```

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

La distance de notre chemin est la suivante:

```
distance_chemin(C, D)
```

```
80.37111579828556
```

Améliorer la distance du chemin

Pour améliorer la distance du chemin, nous allons laisser une colonie de fourmis virtuelles se déplacer entre les points. Les fourmis se déplacent en prennant en compte deux informations.

D'abord, une fourmi a plus de chances d'aller visiter un site qui est proche du site sur lequel elle se trouve – c'est une optimisation à l'échelle "micro", qui favorise les chemins localements courts.

Ensuite, les fourmis vont favoriser les chemins qui ont déjà été visités par d'autres fourmis. Pour rendre ce mécanisme possible, les fourmis vont ajouter des phéromones à chaque fois qu'elles se déplacent entre deux points.

Il faut donc stocker ces phéromones dans une matrice. Elle représente une autre vision de la distance entre les deux sites, et on peut donc la générer en utilisant les informations sur la taille de la matrice de distance:

```
P = zeros(Float64, size(D)...);
```

Le processus de décision d'une fourmi est donc résumé de la manière suivante: la probabilité qu'un site soit choisi pour le prochain mouvement est proportionnel à P^{α}, et inversement proportionnel à D^{β}, avec α et β qui sont des paramètres que l'on peut varier pour rendre la distance ou les phéromones plus importants.

Exemple avec une seule fourmi

Imaginons que la fourmi soit initialement sur le site 1, on commence par identifier les sites qui sont disponibles pour une visite:

```
sites_disponibles = filter(x -> x != 1, 1:points);
```

On choisit les valeurs pour α et β:

```
α, β = 0.2, 3.7
```

```
(0.2, 3.7)
```

Et on mesure les distances avec tous ces sites:

```
D[1, sites_disponibles] .^ β;
```

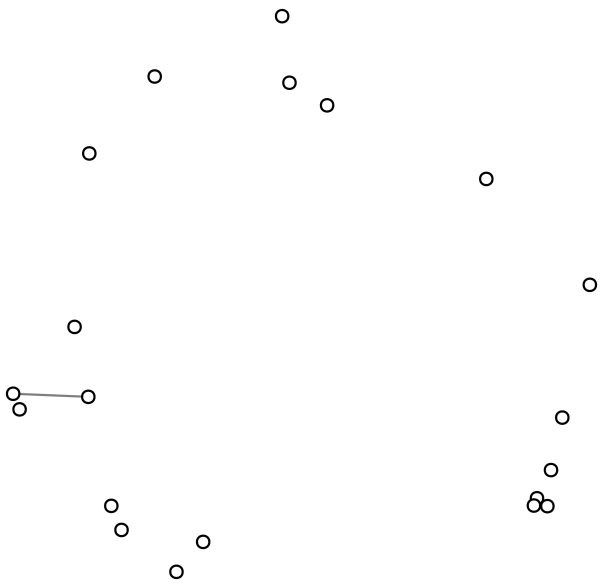
On peut choisir un de ces sites au hasard en prennant en compte les poids:

```
next_site = StatsBase.sample(sites_disponibles, Weights(1.0 ./
(D[1, sites_disponibles] .^ β)))
```

```
16
```

Et visualiser le site suivant:

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
linesegments!(ax, xy[1, [1, next_site]], xy[2, [1, next_site]],
color=:grey50)
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
hidespines!(ax)
hidedecorations!(ax)
f
```



Plus la valeur de β est élevé, plus la probabilité que les sites distants soient choisis diminue.

Pour le moment, on ne prend pas en compte la valeur des phéromones, parce qu'il faut identifer un mécanisme pour que les fourmis déposent ces phéromones. Avant de faire ce travail, on peut regarder quel chemin serait choisit uniquement sur la base de la distance

```
function chemin_proximity(xy, D)
    pool = collect(axes(xy, 2))
    chemin = []
    push!(chemin, popat!(pool, rand(eachindex(pool))))
    while !isempty(pool)
        next_site_index = StatsBase.sample(eachindex(pool),
Weights(1.0 ./ (D[last(chemin), pool])))
        push!(chemin, popat!(pool, next_site_index))
    end
    push!(chemin, first(chemin))
    return chemin
end
```

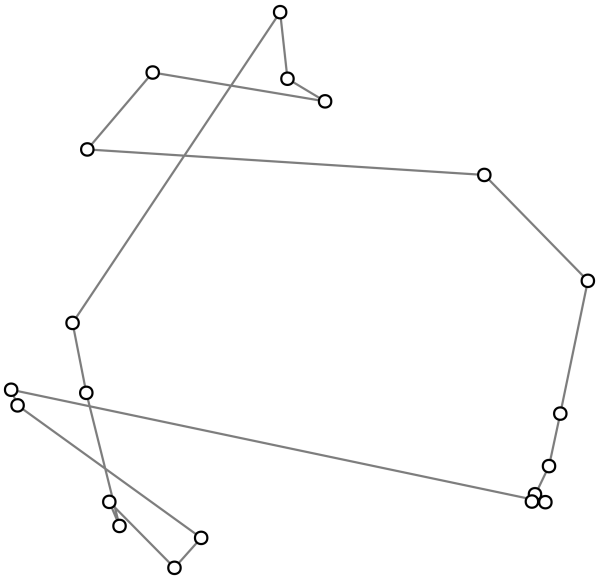
```
chemin_proximity (generic function with 1 method)
```

new test

```
C = chemin_proximity(xy, D .^ β);
```

On peut maintenant visualiser ce chemin

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
for i in 2:length(C)
    linesegments!(ax, xy[1, [C[i-1], C[i]]], xy[2, [C[i-1],
C[i]]], color=:grey50)
end
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
hidespines!(ax)
hidedecorations!(ax)
f
```



La distance de ce chemin est meilleure que la version aléatoire:

```
distance_chemin(C, D)
```

```
26.81860197674849
```

On va maintenant déposer les phéromones. Dans cette simulation, on considère que les phéromones sont directionnels. La logique pour les phéromones est la suivante. Chaque fourmi dispose d'une

quantité total de phéromone, et leur trace est donc moins intense pour un chemin plus long. Pour simplifier, on va considérer que chaque fourmi dispose d'une unité de phéromone pour chaque site.

```
function pheromones!(P, chemin, D, n)
    qte_pheromone = size(P, 1)
    score = qte_pheromone / distance_chemin(chemin, D)
    for i in 2:length(chemin)
        P[sort([chemin[i-1], chemin[i]])...] += score*n
    end
    return P
end
```

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

On applique cette fonction pour mettre à jour les phéromones qui sont maintenant sur la surface couverte par les points:

```
pheromones!(P, C, D, 1);
```

Choix du chemin optimal

Nous allons maintenant mettre à jour la fonction complète:

```
function chemin_complet(xy, D, P)
    pool = collect(axes(xy, 2))
    chemin = []
    push!(chemin, popat!(pool, rand(eachindex(pool))))
    while !isempty(pool)
        next_site_index = StatsBase.sample(eachindex(pool),
Weights((P[last(chemin), pool] .+ 1e-2) ./ (D[last(chemin),
pool])))
        push!(chemin, popat!(pool, next_site_index))
    end
    push!(chemin, first(chemin))
    return chemin
end
```

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

Le principe de cette simulation est de faire se déplacer *beaucoup* de fourmis en même temps:

```
n_fourmis = 50
```

```
50
```

On peut maintenant générer un chemin pour chaque fourmi:

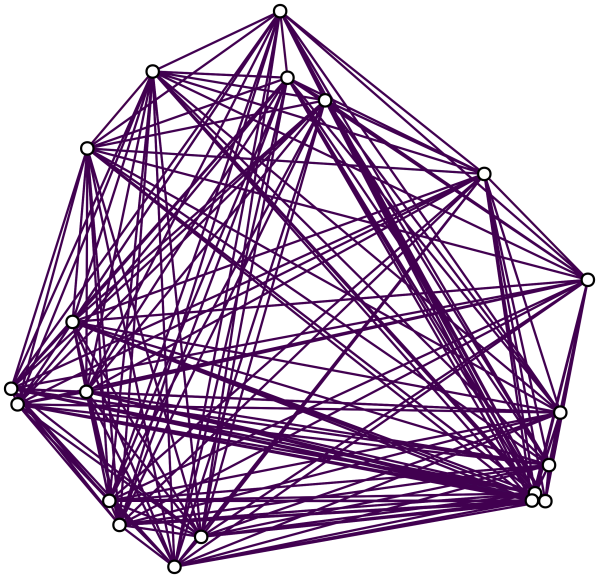
```
chemins = []
for i in 1:n_fourmis
    push!(chemins, chemin_complet(xy, D .^ β, P .^ α))
end
```

Après que toutes les fourmis ont terminé leur parcours, on va maintenant mettre à jour les phéromones:

```
for i in 1:n_fourmis
    pheromones!(P, chemins[i], D, 1/n_fourmis)
end
```

Voici le résultat sous forme de figure.

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
for i in axes(P, 1)
    for j in axes(P, 2)
        lines!(ax, [xy[1, i], xy[1, j]], [xy[2, i], xy[2, j]],
color=P[i, j], colorrange=extrema(P))
    end
end
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
hidespines!(ax)
hidedecorations!(ax)
f
```



Évaporation

Pour permettre d'optimiser ce chemin, on veut enlever un peu de phéromones à chaque génération. Cette opération est assez simple:

```
evaporation_rate = 0.9
P .*= evaporation_rate;
```

Une fois que cette évaporation est appliquée, il suffit de relancer une nouvelle génération de fourmis:

Simulation

```
P = zeros(Float64, size(D)...)
for generation in 1:20
    global chemins = []
    for i in 1:n_fourmis
        push!(chemins, chemin_complet(xy, D .^ β, P .^ α))
    end

    for i in 1:n_fourmis
        pheromones!(P, chemins[i], D, 1/n_fourmis)
    end
    P .*= evaporation_rate
end
```

Voici le résultat sous forme de figure.

```
f = Figure()
ax = Axis(f[1, 1], aspect=DataAspect())
for i in axes(P, 1)
    for j in axes(P, 2)
        lines!(ax, [xy[1, i], xy[1, j]], [xy[2, i], xy[2, j]],
color=P[i, j], alpha=P[i, j], colorrange=extrema(P),
colormap=:Greys)
    end
end
scatter!(ax, xy, color=:white, strokecolor=:black,
strokewidth=1.5, markersize=12)
hidespines!(ax)
hidedecorations!(ax)
f
```