TP0 - Poisson

M1 CSMI, S1 2023

Victor Michel-Dansac **Léopold Trémant**

1 Discrétisation du problème

Soient $L > 0$ et f une fonction réelle. On considère le problème de Poisson, d'inconnue $u \in \mathcal{C}^2([0, L], \mathbb{R})$:

$$\forall x \in]0, L[, \quad -u''(x) = f(x). \tag{1}$$

On considère dans un premier temps les conditions aux limites de Dirichlet homogène données par

$$u(0) = u(L) = 0. \tag{2}$$

Question 1.1 Proposer une discrétisation en différences finies de l'équation (1) avec les conditions aux limites (2).

Question 1.2 Écrire cette discrétisation sous forme matricielle, sous la forme $MU = F$, où $M \in \mathcal{M}_n(\mathbb{R})$, $U \in \mathbb{R}^n$ et $F \in \mathbb{R}^n$. Que remarque-t-on sur la structure de M ?

2 La factorisation LU

2.1 Propriétés de la factorisation

On considère une matrice tridiagonale $A = (a_{ij})_{1 \le i,j \le n} \in \mathcal{M}_{n(\mathbb{R})}$ dont les éléments, pour $1 \le i, j \le n$, sont donnés par

$$a_{ij} = \begin{cases} \alpha_i & \text{si } j = i - 1, \\ \beta_i & \text{si } j = i, \\ \gamma_i & \text{si } j = i + 1, \\ 0 & \text{sinon.} \end{cases} \quad \text{i.e.} \quad A = \begin{pmatrix} \beta_1 & \gamma_1 & & & (0) \\ \alpha_1 & \beta_2 & \gamma_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \alpha_{n-2} & \beta_{n-1} & \gamma_{n-1} \\ (0) & & & \alpha_{n-1} & \beta_n \end{pmatrix}.$$

Question 2.1.a Calculer la décomposition LU d'une telle matrice. Que remarque-t-on ? On pourra commencer par se placer dans un cas simplifié, par exemple pour $n = 4$, et pour A donnée par :

$$\begin{cases} \forall\, 2 \le i \le n, & \alpha_i = \gamma_{i-1} = -1, \\ \forall\, 1 \le i \le n, & \beta_i = 2. \end{cases} \tag{3}$$

Question 2.1.b Résoudre le système linéaire $Ax = b$ d'inconnue $x \in \mathbb{R}^n$ en utilisant la décomposition LU de A.

2.2 Implémentation en Julia

On peut écrire des scripts Julia (extension `.jl`) ou des notebooks Jupyter (extension `.ipynb`) pour exécuter du code. Pour afficher la documentation d'une fonction `f`, on peut toujours taper `?f` dans le REPL.

2.2.a Calcul de la factorisation

```julia
"Factorisation LU pour une matrice tridiagonale"
struct TridiagLU
    lower :: Vector{Float64}
    diag :: Vector{Float64}
    upper :: Vector{Float64}
end
```

En contraste avec le C++ ou le Python, les structures ne sont pas des classes ! Plutôt que d'utiliser des méthodes, on crée ou surcharge des fonctions dont on a besoin. Il n'est pas nécessaire de typer les fonctions, mais il est préférable de le faire.

```julia
"Initialisation de la mémoire pour une factorisation n×n"
TridiagLU(n::Int64) = TridiagLU(zeros(n-1), zeros(n), zeros(n-1))
"Dimensions de la matrice représent"
Base.size(A::TridiagLU) = (length(A.diag), length(A.diag))
```

Pour faire du code efficace, on alloue d'abord la mémoire, et *ensuite* on effectue les opérations. Pour indiquer que le contenu mémoire est modifié (sans modifier les adresses), ont met un `!` dans le nom de la fonction. On parle alors de fonction « in-place ».

```julia
function factorize!(LU::TridiagLU, A::Tridiagonal)
    n = size(A, 1)
    lower, diag, upper = zeros(n-1), zeros(n), zeros(n-1)
    LU.diag[1] = A[1, 1]
    for i = 2:n
        LU.lower[i-1] = A[i, i-1] / LU.diag[i-1]
        LU.upper[i-1] = A[i-1, i]
        LU.diag[i] = A[i, i] - LU.lower[i-1] * LU.upper[i-1]
    end
    return LU
end
```

Il n'est pas nécessaire de renvoyer l'argument modifié, mais il s'agit d'une convention (pas toujours respectée). On peut alors simplifier la syntaxe pour construire une nouvelle factorisation LU :

```julia
function TridiagLU(A::Matrix)
    LU = TridiagLU(size(A, 1)) # on initialise la mémoire
    factorize!(LU, A)          # on effectue la factorisation
end
```

Par défaut, comme `factorize!` renvoie `LU`, la fonction `Tridiag` le renverra aussi. Notez que les indices commencent à 1 dans ce langage.

2.2.b Produit factorisation-vecteur

Le produit matrice-vecteur est déjà implémenté en Julia.

```julia
A = randn(4,4) # matrice aléatoire 4x4
x = randn(4)   # vecteur aléatoire de taille 4
b = A*x
```

Pour faire une multiplication « in-place », on utilise la syntaxe

```julia
b = zeros(4) # initialisation de la mémoire
mul!(b, A, x) # produit matrice-vecteur
```

Pour rester attentif à la mémoire, nous ne surchargeons que cette dernière fonction pour notre structure :

```julia
function mul!(b::Vector{Float64}, LU::TridiagLU, x::Vector{Float64})
    n, = size(LU)
    b[1] = LU.diag[1] * x[1] + LU.upper[1] * x[2]
    for i in 2 : n-1
        #TODO
    end
    b[end] = LU.lower[end] * x[end-1] + LU.diag[end] * x[end]
    return b
end
```

2.2.c Résolution de système

```julia
function _solve_l!(y, LU, b)
    #TODO
    return y
end

function _solve_u!(x, LU, y)
    n = length(x)
    x[n] = y[n] / LU.diag[n]
    for i in reverse(1:n-1)
        #TODO
    end
    return x
end

"""
    solve!(x::Vector{Float64}, LU::TridiagLU, b::Vector{Float64})

Résout le système `LU*x = b` en modifiant `x` "in-place".
"""
function solve!(x::Vector{Float64}, LU::TridiagLU, b::Vector{Float64})
    _solve_l!(x, LU, b)
    _solve_u!(x, LU, x)
    return x
end
```

3 Application au problème de Poisson

3.1 Conditions aux limites de Dirichlet

Question 3.1.a Écrire une fonction `poisson_matrix` qui prend en entrée la taille `n` du système et la taille `L` du domaine, et qui construit la factorisation LU de la matrice de Poisson associée.

Question 3.1.b Écrire une fonction `solve_dirichlet` qui prend en entrée une taille de domaine `L` et un vecteur `F` et qui calcule la solution `U`.

Question 3.1.c On considère une solution fabriquée $u : x \mapsto \sin\left(p\pi \frac{x}{L}\right)e^{\lambda x}$ pour $p \in \mathbb{N}^*$ et $\lambda \in \mathbb{R}$. Quel est le terme source qui génère cette solution ? Écrire des fonctions Julia associées `u_exact` et `f_source`, qui prennent en entrée `x`, `p`, `λ` (taper `\lambda` + Tab) et `L`.

Pour $L = 2$, $n = 100$, $p = 1$ et $\lambda = 0.5$, on peut calculer les vecteurs `F_source` et `U_exact` avec

```
n = 100
p, λ, L = 1, 0.5, 2.0
x = range(0.0, L, n+2)
U_ref = u_exact.(x, p, λ, L)
F = f_source.(x, p, λ, L)
```

Le point en décorateur indique qu'on *vectorise* la fonction, i.e. qu'on l'applique terme-à-terme à `x`.

Question 3.1.d Définir une fonction `test_dirichlet` qui prend en entrée `n`, `p`, `λ` et `L` et qui renvoie `x`, `U` et `U_ref`.

On peut tracer un exemple avec la syntaxe

```
x, U, U_ref = test_dirichlet(n, p, λ, L)
plot(x, U, label = "sol exacte")
plot!(x, U, label = "sol num")
```

Question 3.1.e Tracer l'évolution de l'erreur

$$e(n) = \sup_{1 \leq i \leq n} |u_i - u(x_i)|. \tag{4}$$

en fonction de n (en échelle logarithmique) pour différentes valeurs de $p \in \mathbb{N}$ et $\lambda \in \mathbb{R}$.

3.2 Conditions aux limites de Neumann

Effectuer le même travail avec des conditions aux limites de Neumann homogènes.