Numerical Algorithms (MU4IN910)

Practical 5

Assia MASTOR

March 8, 2024

Exercise 3

1- Méthode de Newton pour les fonctions non linéaires:

```matlab
%Méthode de Newton pour les fonctions non lineaires
function [root, iterations] = newtons_method(f, df, x0, tol, max_iter)
    iterations = 0;
    while iterations < max_iter
        fx = f(x0);
        if abs(fx) < tol
            root = x0;
            return;
        end
        dfx = df(x0);
        if dfx == 0
            error('La dérivée est nulle. Impossible de continuer.');
        end
        x1 = x0 - fx / dfx;
        if abs(x1 - x0) < tol
            root = x1;
            return;
        end
        x0 = x1;
        iterations = iterations + 1;
    end
    error('Nombre maximum d\ iterations atteint. Pas de convergence.');
end
```

2- Une solution approximative pour la racine positive de l'équation $x^3 = \cos(x)$ est 0.86547. De plus, l'étude de la vitesse de convergence indique que seulement 5 itérations ont été nécessaires pour atteindre cette solution.

```matlab
f = @(x) x^3 - cos(x);

% La dérivée de la fonction
df = @(x) 3*x^2 + sin(x);


x0 = 0.5; % valeur initiale
tol = 1e-6; % Tolérance pour la convergence
max_iter = 1000; % Nombre max d'itérations

% On trouve la racine avec la méthode de Newton
[root, iterations] = newtons_method(f, df, x0, tol, max_iter);
disp(['Racine : ', num2str(root)]);
disp(['Itérations : ', num2str(iterations)]);
```

```
>> tp5
Racine : 0.86547
Itérations : 5
>>
```

Exercise 4

1-

```matlab
% Méthode p-adique de Newton pour trouver des racines modulo p^k
function root = newton_p_adic(f, f_prime, x0, p, k, max_iter, tolerance)

    % Initialisation
    root = x0;

    for iter = 1:max_iter
        % Évaluation de f et f_prime à la racine actuelle
        f_val = mod(polyval(f, root), p^k);
        f_prime_val = mod(polyval(f_prime, root), p^k);

        % Calcul de l'inverse modulaire de f_prime_val
        f_prime_inv = mod_inverse(f_prime_val, p^k);

        % Mise à jour de la racine
        root = mod(root - f_val * f_prime_inv, p^k);

        if abs(f_val) < tolerance
            return;
        end
    end

    error("La méthode de Newton n'a pas convergé dans le nombre maximal d'itérations.");
end
```

```matlab
% Calcul de l'inverse modulaire de a modulo m en utilisant l'algorithme d'Euclide étendu
function inv = mod_inverse(a, m)
    [g, x, ~] = extended_gcd(a, m);
    if g ~= 1
        error('L\'inverse modulaire n\'existe pas');
    end
    inv = mod(x, m);
end

% Algorithme d'Euclide étendu pour calculer le PGCD de a et b
function [g, x, y] = extended_gcd(a, b)
    if b == 0
        g = a;
        x = 1;
        y = 0;
    else
        [g, x1, y1] = extended_gcd(b, mod(a, b));
        x = y1;
        y = x1 - floor(a/b) * y1;
    end
end
```

2-

```matlab
f = [1 0 -2];  % coefficients en ordre décroissant des puissances

%Dérivée f'
f_prime = polyder(f);

x0 = 3;  % estimation initiale
p = 7;
k_values = [2, 3, 4];
max_iter = 100;         % nombre max d'itérations
tolerance = 1e-6;       % tolérance pour la convergence

% Calculer les racines modulo p^k pour chaque k
for i = 1:length(k_values)
    k = k_values(i);
    root = newton_p_adic(f, f_prime, x0, p, k, max_iter, tolerance);
    disp(['Racine modulo p^', num2str(k), ': ', num2str(root)]);
end
```

En exécutant le code, nous obtenons les résultats suivants :

```
Racine modulo p^2: 10
Racine modulo p^3: 108
Racine modulo p^4: 2166
```

1. Pour montrer que l'itération de Newton vérifie $X_{n+1} = 2X_n - X_n A X_n$, commençons par l'itération de Newton :

À partir de $f(X) = A - X^{-1}$, nous avons :

$$f'(X) = \frac{d}{dX}(A - X^{-1}) = X^{-2}$$

L'itération de Newton est donnée par :

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}$$

En substituant $f(X)$ et $f'(X)$, nous obtenons :

$$X_{n+1} = X_n - \frac{A - X_n^{-1}}{X_n^{-2}} = X_n - X_n^2(A - X_n^{-1}) = 2X_n - X_n A X_n$$

Cela démontre que l'itération de Newton vérifie $X_{n+1} = 2X_n - X_n A X_n$.

2-

```
function [X, errors] = newtons_method_inverse(A, max_iter, tolerance)
    n = size(A, 1);
    X = A.' / trace(A.' * A);
    errors = zeros(max_iter, 1);

    for iter = 1:max_iter
        X_new = 2 * X - X * A * X;
        errors(iter) = norm(eye(n) - X * A, 'fro');

        if errors(iter) < tolerance
            break;
        end

        X = X_new;
    end
end
```