Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function

2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

MCS 471 Lecture 11
Numerical Analysis
Jan Verschelde, 16 September 2022

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function

2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

a fixed point formula

We want to solve $A\mathbf{x} = \mathbf{b}$ for $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$, for ***very large*** n.

Consider $A = L + D + U$, where

- $L = [\ell_{i,j}], \ell_{i,j} = a_{i,j}, i > j, \ell_{i,j} = 0, i \leq j$. L is lower triangular.
- $D = [d_{i,j}], d_{i,i} = a_{i,i} \neq 0, d_{i,j} = 0, i \neq j$. D is diagonal.
- $U = [u_{i,j}], u_{i,j} = a_{i,j}, i < j, u_{i,j} = 0, i \geq j$. U is upper triangular.

Then we rewrite $A\mathbf{x} = \mathbf{b}$ as

$$
\begin{aligned}
A\mathbf{x} = \mathbf{b} \quad & \Leftrightarrow \quad (L + D + U)\mathbf{x} = \mathbf{b} \\
& \Leftrightarrow \quad D\mathbf{x} = \mathbf{b} - L\mathbf{x} - U\mathbf{x} \\
& \Leftrightarrow \quad D\mathbf{x} = D\mathbf{x} + \mathbf{b} - L\mathbf{x} - U\mathbf{x} - D\mathbf{x} \\
& \Leftrightarrow \quad D\mathbf{x} = D\mathbf{x} + \mathbf{b} - A\mathbf{x} \\
& \Leftrightarrow \quad \mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x}).
\end{aligned}
$$

The fixed point formula $\mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x})$ is well defined if $a_{i,i} \neq 0$.

the Jacobi iterative method

The fixed point formula $\mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x})$ leads to

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \underbrace{D^{-1}\left(\mathbf{b} - A\mathbf{x}^{(k)}\right)}_{\Delta\mathbf{x}}, \quad k = 0, 1, \ldots$$

Writing the formula as an algorithm:

Input: A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, ϵ, N.
Output: $\mathbf{x}^{(k)}$, k is the number of iterations done.

for k from 1 to N do
    $\Delta x := D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$
    $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \Delta\mathbf{x}$
    exit when $(||\Delta\mathbf{x}|| \leq \epsilon)$
end for.

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function

2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

cost and convergence

Counting the number of operations in

for k from 1 to N do
  $\Delta x := D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$;
  $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \Delta\mathbf{x}$;
  exit when $(||\Delta\mathbf{x}|| \leq \epsilon)$;
end for.

we have a cost of $O(Nn^2)$, $O(n^2)$ for $A\mathbf{x}^{(k)}$, if A is dense.

Theorem

The Jacobi method converges for strictly row-wise or column-wise diagonally dominant matrices, i.e.: if

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}| \quad \text{or} \quad |a_{i,i}| > \sum_{j \neq i} |a_{j,i}|, \quad i = 1, 2, \ldots, n.$$

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function
2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

design of a Julia function

```
using Printf
Base.show(io::IO, f::Float64) = @printf(io, "%.3e", f)
using LinearAlgebra
"""
    jacobi(mat::Array{Float64},rhs::Array{Float64},
           sol::Array{Float64},
           maxit::Int=100,tol::Float64=1.0e-8)

Runs the method of Jacobi on the linear system with coefficient
matrix mat, with right hand side vector  rhs, a start solution sol
Running stops if the maximum number of iterations in maxit is
reached, or if the norm of the correction is less than tol.

Returns (solution, numit, nrmdx, fail), the computed solution,
the number of iterations numit, an estimate for the forward error
normdx, and fail is true if the given tolerance was not reached.
"""
function jacobi(mat::Array{Float64},rhs::Array{Float64},
                sol::Array{Float64},
                maxit::Int=100,tol::Float64=1.0e-8)
```

the Julia function `jacobi`

```
function jacobi(mat::Array{Float64},rhs::Array{Float64},
                sol::Array{Float64},
                maxit::Int=100,tol::Float64=1.0e-8)
    nbrows, nbcols = size(mat)
    result = deepcopy(sol)
    numit = 0; nrmdx = 1
    while numit < maxit
       numit = numit + 1
       deltax = rhs - mat*result
       for i=1:nbrows
           deltax[i] = deltax[i]/mat[i,i]
       end
       result = result + deltax
       nrmdx = norm(deltax)
       strdx = @sprintf("%.2e", nrmdx)
       println("||dx|| = $strdx")
       if norm(deltax) <= tol
           return (result, numit, nrmdx, false)
       end
    end
    return (result, numit, nrmdx, true)
end
```

the main program in `runjacobi.jl`

```
import Random # to fix the seed of the random numbers
include("jacobi.jl")
"""
Prompts the user for a dimension and then
generates a random matrix to test the Jacobi method.
"""
function main()
    print("Give the dimension : ")
    line = readline(stdin)
    dim = parse(Int, line)
    Random.seed!(123);
    mat = rand(dim, dim)
    # make the matrix diagonally dominant
    for i=1:dim
        mat[i, i] = 100*mat[i,i]
    end
    sol = ones(dim, 1)
    noise = (1.0e-4)*rand(dim, 1)
    rhs = mat*sol
    wrk = sol + noise
```

runjacobi.jl continued

```
    println("A random matrix :")
    show(stdout, "text/plain", mat); println("");
    sol, numit, nrmdx, fail = jacobi(mat, rhs, wrk)
    println("The solution after ", numit, " iterations :")
    for i=1:dim
        strsol = @sprintf("%.16e", sol[i])
        println(i, " : $strsol")
    end
    print("Estimated forward error : ", nrmdx)
    if fail
        println(" failed.")
    else
        println(" succeeded.")
    end
end

main()
```

running `runjacobi.jl` at the command prompt

```
$ julia runjacobi.jl
Give the dimension : 3
A random matrix :
3×3 Array{Float64,2}:
 7.684e+01  3.955e-01  5.860e-01
 9.405e-01  3.132e+01  5.213e-02
 6.740e-01  6.626e-01  2.686e+01
||dx|| = 5.21e-05
||dx|| = 9.32e-07
||dx|| = 2.73e-08
||dx|| = 6.56e-10
The solution after 4 iterations :
1 : 1.000000000054128e+00
2 : 1.000000000079821e+00
3 : 1.000000000121563e+00
Estimated forward error : 6.559e-10 succeeded.
$
```

a first exercise

Exercise 1: Consider `runjacobi.jl`.

1. What is the largest dimension M for which `runjacobi.jl` reports success?

   Make a table with dimension $n = 2, 3, \ldots, M$
   and the number of iterations for each dimension n.

2. Adjust the statement `mat[i, i] = 100*mat[i,i]`
   (you may modify only the diagonal element of `mat`*)*
   so the matrix is always diagonally dominant,
   for any dimension of the matrix.

   Illustrate the adjustment with a run for a dimension larger than the M you found in the first part of the exercise.

a test system

For the dimension n, we consider the diagonally dominant system:

$$\begin{bmatrix} n+1 & 1 & \cdots & 1 \\ 1 & n+1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & n+1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2n \\ 2n \\ \vdots \\ 2n \end{bmatrix}.$$

The exact solution is $\mathbf{x}$: for $i = 1, 2, \ldots, n$, $x_i = 1$.

Exercise 2:
Start the Jacobi iteration method at $\mathbf{x}^{(0)} = \mathbf{0}$,
with tolerance 10^{-4}, allowing $N = 2n^2$ iterations,
for $n = 10, 20, 40$, and 80.

How many steps does the method of Jacobi take to converge?

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function
2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

fixed point formula for Gauss-Seidel relaxation

The fixed point formula for $A\mathbf{x} = \mathbf{b}$ where $A = L + D + U$,

- L is strict lower triangular, $L = [a_{i,j}]$, $i > j$, 0 otherwise
- D is diagonal, $D = [a_{i,j}]$, $i = j$, 0 otherwise
- U is strict upper triangular, $U = [a_{i,j}]$, $i < j$, 0 otherwise

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \quad & \Leftrightarrow \quad (L + D + U)\mathbf{x} = \mathbf{b} \\ & \Leftrightarrow \quad (L + D)\mathbf{x} + U\mathbf{x} = \mathbf{b} \\ & \Leftrightarrow \quad (L + D)\mathbf{x} = \mathbf{b} - U\mathbf{x} \end{aligned}$$

Observe that $L + D$ is lower triangular.
We apply forward substitution in each step.

the formulas for Gauss-Seidel relaxation

We want to solve $A\mathbf{x} = \mathbf{b}$ for $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$, for ***very large*** n.

Writing the method of Jacobi componentwise:

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

we observe that we can already use $x_j^{(k+1)}$ for $j < i$.

This leads to the following formulas

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function
2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

the Gauss-Seidel method

Writing the formulas as an algorithm:

Input: A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, ϵ, N.
Output: $\mathbf{x}^{(k)}$, k is the number of iterations done.

for k from 1 to N do
    for i from 1 to n do
        $\Delta x_i := b_i$
        for j from 1 to $i - 1$ do
            $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k+1)}$
        for j from i to n do
            $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k)}$
        $\Delta x_i := \Delta x_i / a_{i,i}$
        $x_i^{(k+1)} := x_i^{(k)} + \Delta x_i$
    exit when $(||\Delta \mathbf{x}|| \leq \epsilon)$

the Julia function gauss_seidel

```
function gauss_seidel(mat::Array{Float64},rhs::Array{Float64
                      sol::Array{Float64},
                      maxit::Int=100,tol::Float64=1.0e-8)
    nbrows, nbcols = size(mat)
    result = deepcopy(sol)
    deltax = zeros(nbrows, 1)
    numit = 0; nrmdx = 1;
    while numit < maxit
        numit = numit + 1
        for i=1:nbrows
            deltax[i] = rhs[i]
            for j=1:nbcols
                deltax[i] = deltax[i] - mat[i,j]*result[j]
            end
            deltax[i] = deltax[i]/mat[i,i]
            result[i] = result[i] + deltax[i]
        end
        # The rest is the same as in the function jacobi.
```

running `rungauss_seidel.jl` as a program

```
$ julia rungauss_seidel.jl
Give the dimension : 3
A random matrix :
3×3 Array{Float64,2}:
 7.684e+01  3.955e-01  5.860e-01
 9.405e-01  3.132e+01  5.213e-02
 6.740e-01  6.626e-01  2.686e+01
||dx|| = 5.13e-05
||dx|| = 4.50e-07
||dx|| = 2.41e-10
The solution after 3 iterations :
1 : 1.000000000001867e+00
2 : 1.000000000000031e+00
3 : 9.999999999999523e-01
Estimated forward error : 2.405e-10 succeeded.
$
```

Compare with `runjacobi.jl`*!*

convergence

We have the same condition on convergence as the method of Jacobi:

Theorem

The Gauss-Seidel method converges for strictly row-wise or column-wise diagonally dominant matrices, i.e.: if

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}| \quad \text{or} \quad |a_{i,i}| > \sum_{j \neq i} |a_{j,i}|, \quad i = 1, 2, \ldots, n.$$

The method of Gauss-Seidel converges faster than the method of Jacobi.

comparing on the test system

Exercise 3:
Consider again the test system as in exercise 2.
Solve exercise 2 with the method of Gauss-Seidel.
Compare the convergence of the method of Gauss-Seidel with the method of Jacobi.

Iterative Methods for Linear Systems

1. the method of Jacobi
   - derivation of the formulas
   - cost and convergence of the algorithm
   - a Julia function

2. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - the algorithm
   - successive over-relaxation

successive over-relaxation

Successive Over-Relaxation (SOR) takes a weighted average of the current and the new approximation, using the *relaxation parameter* ω.
For $\omega > 1$, we have over-relaxation, under-relaxation for $\omega < 1$.

Writing $A = L + D + U$, we derive

$$\begin{aligned}
(\omega L + \omega D + \omega U)\mathbf{x} &= \omega \mathbf{b} \\
(\omega L + \omega D + D - D)\mathbf{x} &= \omega \mathbf{b} - \omega U \mathbf{x} \\
(\omega L + D)\mathbf{x} &= \omega \mathbf{b} + (1 - \omega) D \mathbf{x} - \omega U \mathbf{x} \\
\mathbf{x} &= (\omega L + D)^{-1} \left[\omega \mathbf{b} + (1 - \omega) D \mathbf{x} - \omega U \mathbf{x} \right]
\end{aligned}$$

For $\omega = 1$, we have $\mathbf{x} = (L + D)^{-1} [\mathbf{b} - U\mathbf{x}]$, which is Gauss-Seidel.

code for the Julia function `sor`

The ω is provided by the parameter `wgt`.

```
function sor(mat::Array{Float64},rhs::Array{Float64},
             sol::Array{Float64},wgt::Float64=1.1,
             maxit::Int=100,tol::Float64=1.0e-8)
    nbrows, nbcols = size(mat)
    result = deepcopy(sol)
    deltax = zeros(nbrows, 1)
    numit = 0; nrmdx = 1;
    while numit < maxit
        numit = numit + 1
        for i=1:nbrows
            deltax[i] = wgt*rhs[i]
            for j=1:nbcols
                deltax[i] = deltax[i] - mat[i,j]*wgt*result[j]
            end
            deltax[i] = deltax[i]/mat[i,i]
            result[i] = result[i] + deltax[i]
       end
       # The rest is the same as in the function jacobi.
```

running on a special matrix

```
"""
Returns a special matrix.
"""
function special_matrix()
    mat = [  3.0, -1.0,  0.0,  0.0,  0.0,  0.5,
            -1.0,  3.0, -1.0,  0.0,  0.5,  0.0,
             0.0, -1.0,  3.0, -1.0,  0.0,  0.0,
             0.0,  0.0, -1.0,  3.0, -1.0,  0.0,
             0.0,  0.5,  0.0, -1.0,  3.0, -1.0,
             0.5,  0.0,  0.0,  0.0, -1.0,  3.0 ]
    mat = reshape(mat, (6, 6))
    mat = permutedims(mat)
    return mat
end
```

Set the solution as `ones(6, 1)`.

the output of `runsor.jl`

```
||dx|| = 1.36e-04
||dx|| = 3.06e-05
||dx|| = 9.12e-06
||dx|| = 2.94e-06
||dx|| = 1.11e-06
||dx|| = 4.50e-07
||dx|| = 6.24e-08
||dx|| = 1.78e-08
||dx|| = 5.69e-09
The solution after 9 iterations :
1 : 9.9999999905817438e-01
2 : 9.9999999785724880e-01
3 : 9.9999999865023992e-01
4 : 9.9999999949241514e-01
5 : 1.0000000002526264e+00
6 : 1.0000000002774445e+00
Estimated forward error : 5.690e-09 succeeded.
```

one last exercise

Exercise 4:
For the special matrix to test successive over-relaxation,
compare with the method of Jacobi and the method of Gauss-Seidel.

Is successive over-relaxation better than Jacobi and Gauss-Seidel?