Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function
2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

MCS 471 Lecture 13
Numerical Analysis
Jan Verschelde, 21 September 2022

Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function

2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

an optimization problem

Let A be a positive definite matrix: $\forall \mathbf{x} : \mathbf{x}^T A \mathbf{x} > 0$ and $A^T = A$.
The optimum of

$$q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b} \text{ is at } A\mathbf{x} - \mathbf{b} = \mathbf{0}.$$

For the exact solution $\mathbf{x}$: $A\mathbf{x} = \mathbf{b}$ and an approximation $\mathbf{x}_k$,
let the error be $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}$.

$$\begin{aligned} ||\mathbf{e}_k||_A^2 = \mathbf{e}_k^T A \mathbf{e}_k &= (\mathbf{x}_k - \mathbf{x})^T A (\mathbf{x}_k - \mathbf{x}) \\ &= \mathbf{x}_k^T A \mathbf{x}_k - 2\mathbf{x}_k^T A \mathbf{x} + \mathbf{x}^T A \mathbf{x} \\ &= \mathbf{x}_k^T A \mathbf{x}_k - 2\mathbf{x}_k^T \mathbf{b} + c \\ &= 2q(\mathbf{x}_k) + c \end{aligned}$$

$\Rightarrow$ minimizing $q(\mathbf{x})$ is the same as minimizing the error.

the gradient and the steepest descent method

Consider the minimization of maximization of a function

$$f(x_1, x_2, \ldots, x_n)$$

in n variables, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$.

The minima and maxima occur where the gradient

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

vanishes. The solutions of $\nabla f = 0$ are critical points.

The steepest descent method is an iterative method:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma_k \nabla f(\mathbf{x}^{(k)}), \quad k = 0, 1, \ldots,$$

where γ_k is the step size.

the conjugate gradient method

The CG method is similar to the steepest descent method.

$\mathbf{x}_0 = \mathbf{0}$; $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$; $\mathbf{p}_0 := \mathbf{r}_0$	
for $k = 1, 2 \ldots, N$ do	*no more than N iterations*
if $\|\mathbf{r}_{k-1}\| \leq \epsilon$ then stop	*stop criterion*
$\alpha_k := \dfrac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^T A \mathbf{p}_{k-1}}$	*step length*
$\mathbf{x}_k := \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_{k-1}$	*update solution*
$\mathbf{r}_k := \mathbf{r}_{k-1} - \alpha_k A \mathbf{p}_{k-1}$	*residual*
$\beta_k := \dfrac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}$	*improvement of step*
$\mathbf{p}_k := \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$	*compute search direction*

Exercise 1: For an n-by-n matrix A, write the number of arithmetical operations in one step as an expression of n. Justify this number.

an informal description

The method updates three vectors $\mathbf{r}_k$, $\mathbf{x}_k$, $\mathbf{p}_k$ in each step k.

- $\mathbf{r}_0 := \mathbf{b} - A\mathbf{x}_0$ is the residual for $\mathbf{x}_0$
- $\mathbf{r}_k := \mathbf{r}_{k-1} - \alpha_k A\mathbf{p}_{k-1}$ is the update of $\mathbf{r}_{k-1}$ to $\mathbf{r}_k$.

We obtain $\mathbf{x}_k$ as $\mathbf{x}_k := \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_{k-1}$,
the vector $\mathbf{p}_{k-1}$ is the direction to update $\mathbf{x}_{k-1}$ to $\mathbf{x}_k$.

Observe

$$\begin{aligned} A\mathbf{x}_k + \mathbf{r}_k &= A(\mathbf{x}_{k-1} + \alpha_k \mathbf{p}_{k-1}) + \mathbf{r}_{k-1} - \alpha_k A\mathbf{p}_{k-1} \\ &= A\mathbf{x}_{k-1} + \mathbf{r}_{k-1}. \end{aligned}$$

For $k = 1$: $A\mathbf{x}_1 + \mathbf{r}_1 = A\mathbf{x}_0 + \mathbf{r}_0 = A\mathbf{x}_0 + \mathbf{b} - A\mathbf{x}_0 = \mathbf{b}$, and $\mathbf{r}_1 = \mathbf{b} - A\mathbf{x}_1$.
So by induction, $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$, for all k.

the update direction is orthogonal to the residual

To derive the formula for α_k, consider

$$\begin{array}{rcl} \mathbf{x}_k & = & \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_{k-1} \\ A\mathbf{x}_k & = & A\mathbf{x}_{k-1} + \alpha_k A\mathbf{p}_{k-1} \\ \mathbf{b} - A\mathbf{x}_k & = & \mathbf{b} - Ax_{k-1} - \alpha_k A\mathbf{p}_{k-1} \\ \mathbf{r}_k & = & \mathbf{r}_{k-1} - \alpha_k A\mathbf{p}_{k-1} \end{array}$$

We want the update vector $\mathbf{p}_{k-1}$ to be orthogonal to the residual vector $\mathbf{r}_k$: $\mathbf{p}_{k-1}^T \mathbf{r}_k = 0$.

$$\begin{array}{rcl} \mathbf{p}_{k-1}^T \mathbf{r}_k = 0 & = & \mathbf{p}_{k-1}^T \mathbf{r}_{k-1} - \alpha_k \mathbf{p}_{k-1}^T A\mathbf{p}_{k-1} \\ & \Downarrow & \\ \alpha_k & = & \dfrac{\mathbf{p}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^T A\mathbf{p}_{k-1}} \end{array}$$

rewriting α_k

We can rewrite the numerator of

$$\alpha_k = \frac{\mathbf{p}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^T A \mathbf{p}_{k-1}}$$

using $\mathbf{p}_k = \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$:

$$\begin{array}{rcl} \mathbf{p}_k & = & \mathbf{r}_k + \beta_k \mathbf{p}_{k-1} \\ \mathbf{r}_k^T \mathbf{p}_k & = & \mathbf{r}_k^T \mathbf{r}_k + \beta_k \mathbf{r}_k^T \mathbf{p}_{k-1} \end{array}$$

By orthogonality of $\mathbf{p}_{k-1}$ to $\mathbf{r}_k$: $\mathbf{r}_k^T \mathbf{p}_{k-1} = \mathbf{p}_{k-1}^T \mathbf{r}_k = 0$.

Therefore $\mathbf{r}_k^T \mathbf{p}_k = \mathbf{r}_k^T \mathbf{r}_k$, for all k, also for $k-1$, and $\alpha_k = \dfrac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^T A \mathbf{p}_{k-1}}$.

ensure A-conjugacy

To derive the formula for β_k, consider $\mathbf{p}_k = \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$.

The β_k is determined for pairwise A-conjugacy: $\mathbf{p}_{k-1}^T A \mathbf{p}_k = 0$.

$$\begin{array}{rcl} \mathbf{p}_k & = & \mathbf{r}_k + \beta_k \mathbf{p}_{k-1} \\ 0 = \mathbf{p}_{k-1}^T A \mathbf{p}_\mathbf{k} & = & \mathbf{p}_{k-1}^T A \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}^T A \mathbf{p}_{k-1} \\ & \Downarrow & \\ \beta_k & = & -\dfrac{\mathbf{p}_{k-1}^T A \mathbf{r}_k}{\mathbf{p}_{k-1}^T A \mathbf{p}_{k-1}} \end{array}$$

The expression for β_k can be simplified, using orthogonality: $\mathbf{r}_k^T \mathbf{r}_j = 0$ and A-conjugacy $\mathbf{p}_k A \mathbf{p}_j = 0$, both for all $j < k$.

Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function

2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

a Julia function

```
using Printf
using LinearAlgebra

"""
    CGM(A, b, x0, maxit, tol)

Applies the Conjugate Gradient Method to solve
the linear system Ax = b starting at x0.
"""
function CGM(A::Array{Float64,2},b::Array{Float64,1},
             x0::Array{Float64,1},
             maxit::Int64=10,tol::Float64=1.0e-8,
             verbose=true)
```

loop and stop criterion

```
function CGM(A::Array{Float64,2},b::Array{Float64,1},
             x0::Array{Float64,1},
             maxit::Int64=10,tol::Float64=1.0e-8,
             verbose=true)
    sol = deepcopy(x0)
    r = b - A*sol
    p = deepcopy(r)
    if verbose
        println(" norm(r)    alpha       beta")
    end
    for i=1:maxit
        res = norm(r)
        if verbose
            sres = @sprintf("%.2e", res)
            print("$sres")
        end
        if res < tol
            if verbose
                println("  succeeded after ", i, " steps")
            end
            return (sol, res, i, false)
        end
```

computing the update

```
        alpha = (transpose(r)*r)/(transpose(p)*A*p)
        if verbose
            salpha = @sprintf("%.4e", alpha)
            print("  $salpha")
        end
        sol = sol + alpha*p
        r1 = r - alpha*A*p
        beta = (transpose(r1)*r1)/(transpose(r)*r)
        if verbose
            sbeta = @sprintf("%.4e", beta)
            println("  $sbeta")
        end
        p = r1 + beta*p
        r = r1
    end
    return (sol, norm(r), maxit, true)
end
```

running the method

```
include("conjugategradient.jl")

mat = [2.0 2.0; 2.0 5.0]
rhs = [6.0; 3.0]
sol = [0.0; 0.0]
res = CGM(mat, rhs, sol)
println(res)
```

At the command prompt:

```
$ julia runcgm.jl
 norm(r)     alpha       beta
6.71e+00  2.3810e-01  3.2653e-01
3.83e+00  7.0000e-01  6.9792e-31
3.20e-15  succeeded after 3 steps
([4.0, -0.9999999999999993], \
 3.2023728339893768e-15, 3, false)
$
```

considering the convergence

Exercise 2:

Consider the statements:

```
dim = 3
rnd = rand(dim, dim)
low = LowerTriangular(rnd)
mat = low*transpose(low)
rhs = mat*ones(dim)
sol = zeros(dim)
res = CGM(mat, rhs, sol)
println(res)
```

How many steps does it take for `CGM` to converge?
Repeat the experiment for $\texttt{dim} = 4, 5, 6, 7$.
For each run report the number of steps.
Write at least one sentence to summarize your findings.

Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function
2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

Newton's method for nonlinear systems

Consider a system of two equations in two variables:

$$\left\{ \begin{array}{rcl} f(x,y) & = & 0 \\ g(x,y) & = & 0. \end{array} \right.$$

Suppose we have an approximation for a solution (x_0, y_0) and we would like to compute Δx and Δy so $x_1 = x_0 + \Delta x$ and $y_1 = y_0 + \Delta y$ satisfy the system:

$$\left\{ \begin{array}{rcccl} f(x_1,y_1) & = & f(x_0+\Delta x, y_0+\Delta y) & = & 0 \\ g(x_1,y_1) & = & g(x_0+\Delta x, y_0+\Delta y) & = & 0. \end{array} \right.$$

How to compute Δx and Δy?

Taylor series in two variables

$$
\begin{aligned}
f(x_0 + \Delta x, y_0 + \Delta y) &= f(x_0, y_0) + \frac{\partial f}{\partial x}(x_0, y_0)\Delta x + \frac{\partial f}{\partial y}(x_0, y_0)\Delta y + \cdots \\
g(x_0 + \Delta x, y_0 + \Delta y) &= g(x_0, y_0) + \frac{\partial g}{\partial x}(x_0, y_0)\Delta x + \frac{\partial g}{\partial y}(x_0, y_0)\Delta y + \cdots
\end{aligned}
$$

where

- $\frac{\partial f}{\partial x}(x_0, y_0)$ and $\frac{\partial f}{\partial y}(x_0, y_0)$ are the partial derivatives of f with respect to x and y evaluated at (x_0, y_0);
- $\frac{\partial g}{\partial x}(x_0, y_0)$ and $\frac{\partial g}{\partial y}(x_0, y_0)$ are the partial derivatives of g with respect to x and y evaluated at (x_0, y_0); and
- the $\cdots$ represent the higher order terms in the series, in $(\Delta x)^2$, $(\Delta x)(\Delta y)$, and $(\Delta y)^2$. Because Δx and Δy are already small numbers, the higher order terms are even smaller.

in matrix format

Because $f(x_0 + \Delta x, y_0 + \Delta y) = 0$ and $g(x_0 + \Delta x, y_0 + \Delta y) = 0$:

$$
\begin{aligned}
0 &= f(x_0, y_0) + \frac{\partial f}{\partial x}(x_0, y_0)\Delta x + \frac{\partial f}{\partial y}(x_0, y_0)\Delta y + \cdots \\
0 &= g(x_0, y_0) + \frac{\partial g}{\partial x}(x_0, y_0)\Delta x + \frac{\partial g}{\partial y}(x_0, y_0)\Delta y + \cdots
\end{aligned}
$$

we solve for Δx and Δy:

$$
\left[\begin{array}{cc}
\frac{\partial f}{\partial x}(x_0, y_0) & \frac{\partial f}{\partial y}(x_0, y_0) \\
\frac{\partial g}{\partial x}(x_0, y_0) & \frac{\partial g}{\partial y}(x_0, y_0)
\end{array} \right]
\left[\begin{array}{c} \Delta x \\ \Delta y \end{array} \right]
= - \left[\begin{array}{c} f(x_0, y_0) \\ g(x_0, y_0) \end{array} \right].
$$

The solution $(\Delta x, \Delta y)$ of the linear system updates x_0 and y_0:

$$
x_1 := x_0 + \Delta x \quad \text{and} \quad y_1 := y_0 + \Delta y.
$$

the Jacobian matrix

Given a system of n equations in m unknowns $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, with $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ and $\mathbf{x} = (x_1, x_2, \ldots, x_m)$,

$$\mathbf{f}(\mathbf{x}) = \left\{ \begin{array}{rcl} f_1(x_1, x_2, \ldots, x_m) & = & 0 \\ f_2(x_1, x_2, \ldots, x_m) & = & 0 \\ & \vdots & \\ f_n(x_1, x_2, \ldots, x_m) & = & 0, \end{array} \right.$$

the *Jacobian matrix* of $\mathbf{f}$ is the matrix of all first order partial derivatives:

$$J_\mathbf{f} = \left[\begin{array}{cccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{array} \right].$$

a numerical example

Consider the system

$$\mathbf{f}(x,y) = \left\{ \begin{array}{rcl} e^x - y & = & 0 \\ xy - e^x & = & 0 \end{array} \right. \qquad \mathbf{f}(1,e) = 0.$$

Let us do one step with Newton's method, starting at $(0.9, 2.5)$.
The Jacobian matrix is

$$J_\mathbf{f} = \left[\begin{array}{cc} e^x & -1 \\ y - e^x & x \end{array} \right] \quad A = J_\mathbf{f}(0.9, 2.5) = \left[\begin{array}{rr} 2.5\,\mathrm{E}{+0} & -1.0\,\mathrm{E}{+0} \\ 4.0\,\mathrm{E}{-2} & 9.0\,\mathrm{E}{-1} \end{array} \right]$$

$$\mathbf{f}(0.9, 2.5) = \left[\begin{array}{c} 4.0\,\mathrm{E}{-2} \\ 2.1\,\mathrm{E}{-1} \end{array} \right] \qquad \begin{array}{l} \Delta x = 1.1\,\mathrm{E}{-1}, x = 1.0091\,\mathrm{E}{+0} \\ \Delta y = 2.3\,\mathrm{E}{-1}, y = 2.7280\,\mathrm{E}{+0} \end{array}$$

Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function

2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

computing the Jacobian matrix with SymPy

```
using SymPy
x, y = Sym("x, y")

"""
Returns all evaluable partial derivatives of the
expression in x and y, given in the strings f and g.
"""
function SymPyDerivatives(f::String, g::String)
    evaluatedf = sympify(f)
    fx = diff(evaluatedf, x)
    fy = diff(evaluatedf, y)
    evaluatedg = sympify(g)
    gx = diff(evaluatedg, x)
    gy = diff(evaluatedg, y)
    return [lambdify(fx, (x, y)) lambdify(fy, (x, y));
            lambdify(gx, (x, y)) lambdify(gy, (x, y))]
end
```

evaluating the Jacobian matrix

```
"""
Given a symbolic representation for the
Jacobian matrix, and values for its arguments,
returns the evaluated Jacobian matrix.
"""
function SymPyMatrixEvaluate(jac,
                             xval::Float64,
                             yval::Float64)
    vfx = jac[1, 1](xval, yval)
    vfy = jac[1, 2](xval, yval)
    vgx = jac[2, 1](xval, yval)
    vgy = jac[2, 2](xval, yval)
    return [vfx vfy; vgx vgy]
end
```

code for one Newton step

```
"""
    NewtonStep(fun, jac, x0, y0)

Runs one step with Newton's method,
where fun is the vector function,
jac is the matrix of all partial derivatives
and (x0, y0) is the current point.

On return is a 4-tuple, the coordinates
of the updated point, the norm of the update,
and the norm of the function value at (x0, y0).
"""
function NewtonStep(fun, jac,
                    x0::Float64, y0::Float64)
```

definition of the function

```
using LinearAlgebra

function NewtonStep(fun, jac,
                    x0::Float64, y0::Float64)
    valfun = -SymPyFun(fun, x0, y0)
    nfx = norm(valfun)
    valmat = SymPyMatrixEvaluate(jac, x0, y0)
    update = valmat\valfun
    ndx = norm(update)
    x1 = x0 + update[1]
    y1 = y0 + update[2]
    return [x1, y1, ndx, nfx]
end
```

specification of the method

```
"""
Runs Newton's method on the function defined by fun
and Jacobi matrix jac, starting at (x0, y0),
The maximum number of iterations is given by maxit,
the tolerance on the forward error is dxtol, and
the tolerance on the backward error is fxtol.

Returns the new coordinates of the solution,
the number of steps done, and false or true,
respectively if failed or not.
"""
function Newton(fun, jac,
                x0::Float64, y0::Float64,
                maxit::Int64=5,
                dxtol::Float64=1.0e-8,
                fxtol::Float64=1.0e-8)
```

code before the loop

```
using Printf

function Newton(fun, jac,
                x0::Float64, y0::Float64,
                maxit::Int64=5,
                dxtol::Float64=1.0e-8,
                fxtol::Float64=1.0e-8)
    xsol, ysol = x0, y0
    stri = @sprintf("%3d", 0)
    sx1 = @sprintf("%.16e", xsol)
    sy1 = @sprintf("%.16e", ysol)
    print("step           x                        y
    println("          |update|    |f(x,y)|")
    println("$stri : $sx1  $sy1")
```

the loop

```
    for i=1:maxit
        xsol, ysol, ndx, nfx
             = NewtonStep(fun, jac, xsol, ysol)
        stri = @sprintf("%3d", i)
        sx1 = @sprintf("%.16e", xsol)
        sy1 = @sprintf("%.16e", ysol)
        sdx = @sprintf("%.2e", ndx)
        sfx = @sprintf("%.2e", nfx)
        println("$stri : $sx1  $sy1  $sdx   $sfx")
        if((ndx < dxtol) | (nfx < fxtol))
            return (xsol, ysol, i, false)
        end
    end
    return (xsol, ysol, maxit, true)
end
```

we observe quadratic convergence

In the output below, there are four columns:

1. the value for x,
2. the value for y,
3. the norm of the update,
4. the norm of the residual.

Observe the quadratic convergence:

```
step          x                       y                  |update|    |f(x,y)|
  0 : 9.0000000000000002e-01  2.5000000000000000e+00
  1 : 1.0091197782934511e+00  2.7279944573362789e+00  2.53e-01   2.13e-01
  2 : 1.0000325513375456e+00  2.7182573929531251e+00  1.33e-02   1.80e-02
  3 : 9.9999999970740971e-01  2.7182818262235138e+00  4.07e-05   1.16e-04
  4 : 1.0000000000000000e+00  2.7182818284590451e+00  2.25e-09   2.66e-09
```

intersecting two circles

Exercise 3:
Consider the intersection of two circles:

$$\left\{ \begin{array}{rcl} x^2 + y^2 - 1 & = & 0, \\ (x-1)^2 + y^2 - 1 & = & 0. \end{array} \right.$$

Start at $(x_0, y_0) = (0.2, 0.5)$.
How many iterations are needed to compute an intersection point so the error is less than 10^{-14}?

Do you observe quadratic convergence?

intersecting two circles

Exercise 4:
Consider the intersection of two circles:

$$\left\{ \begin{array}{rcl} x^2 + y^2 - 1 & = & 0, \\ (x-2)^2 + y^2 - 1 & = & 0. \end{array} \right.$$

Start at $(x_0, y_0) = (0.9, 0.1)$.
How many iterations are needed to compute an intersection point so the error is less than 10^{-8}?

Do you observe quadratic convergence?

Make a drawing of the two circles and use this drawing to explain why this problem is harder than the previous exercise.

Conjugate Gradient and Multivariate Newton

1. The Conjugate Gradient Method
   - linear system solving and optimization
   - a Julia function

2. Nonlinear Systems
   - derivation of the method
   - examples with Julia
   - nonlinear optimization

the Hessian

If we take the Jacobian matrix of

$$\left\{ \begin{array}{rcl} \frac{\partial f}{\partial x_1}(\mathbf{x}) & = & 0 \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) & = & 0 \\ & \vdots & \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) & = & 0 \end{array} \right.$$

then we arrive at the second partial derivatives of f:

$$\left[\begin{array}{cccc} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_2 x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{array} \right]$$

If f is continuous, then the matrix is symmetric.

If close to a minimum, then the matrix is positive definite.

six introductory lectures on numerical linear algebra

Six sentences to summarize the last six lectures:

1. Linear systems are formulated with matrices and vectors.
2. Condition numbers determine the difficulty of a linear system.
3. Row pivoting leads to a numerically stable algorithm to compute the LU factorization of a matrix.
4. Iterative methods converge for diagonally dominant matrices.
5. Symmetric positive definite matrices can be factored twice as fast.
6. Gradients, Jacobians, and Hessians occur in iterative methods to solve optimization problems and nonlinear systems.