# UW–Madison Math/CS 714

# Methods of Computational Mathematics I

# Iterative methods I

Instructor: Yue Sun (yue.sun@wisc.edu)

September 23, 2025

# Motivation: solving elliptic problems efficiently

The elliptic problem example codes in the previous section used dense linear algebra for solving the linear system.

For the discretized problem $Au = f$, the matrix $A$ is directly assembled in memory. For an $m \times m$ grid, the matrix $A \in \mathbb{R}^{m^2 \times m^2}$ and has $m^4$ total entries.

$A$ is sparse, meaning most elements are zero. While NumPy can solve matrix systems highly efficiently, the time taken rises very quickly, and NumPy does not exploit the sparsity.

# Testing the speed of the Poisson solver

The program *poisson_time.py* measures the time taken to solve the *poisson2.py* test code. It uses two different measures of time.

Wall-clock time measures the time as perceived by the computer user (*i.e.* by looking at the clock on the wall).

Processor time measures the time that a program spends being processed on a CPU.

# Measures of time

Almost all modern computers (and even smartphones) have multi-core CPUs. When a program runs on multiple cores, processor time accrues across all of the cores.

Basic Python runs on a single core, but libraries like NumPy often use multiple cores.

Thus, if a program takes one second on $n$ cores, the processor time may be approximately $n$ seconds.

# Measures of time

Both measures of time are useful, and highlight different aspects of a calculation.

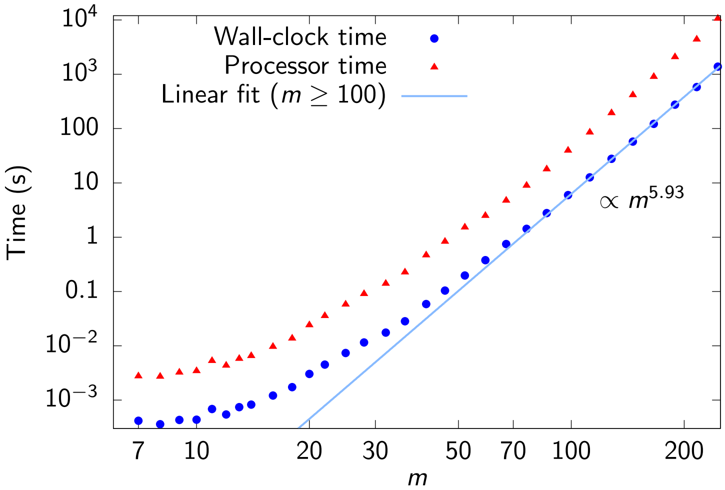
Wall-clock time may be closest to the user's experience, but processor time gives a better indication of the computational resources taken by a job.

Other factors of computer hardware (*e.g.* hyperthreading, Turbo Boost) can affect timing results.[1]

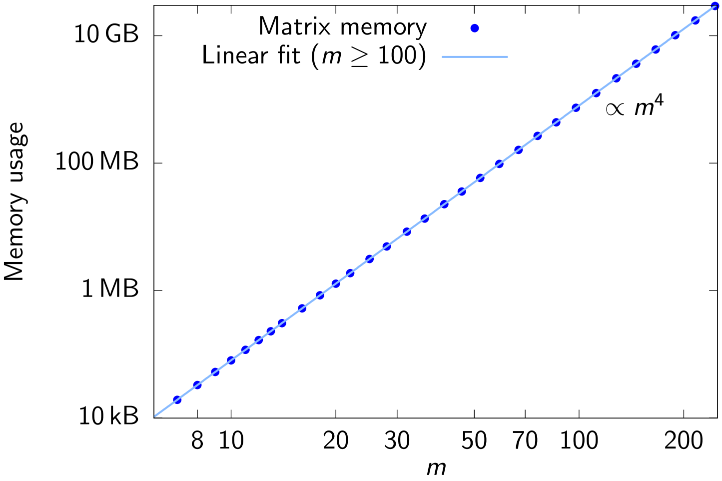
[1] See Harvard AM205 video 3.6 for more discussion and examples.

# Timing graph

*Run with 8 cores on 2020 iMac with 128 GB of memory*



# Memory graph



# Reaching computational limits

When $m$ becomes large the computation time scales like $O(m^6)$. This should be expected. The problem creates an $N \times N$ matrix of size, where $N = m^2$.

NumPy uses the LU factorization to solve the matrix. For an $N \times N$ matrix this takes $O(N^3) = O(m^6)$ time.

The required memory scales like $O(N^2) = O(m^4)$. For both memory and time, the computation scales poorly, and quickly becomes infeasible. We need a better approach to solve systems like this.

# Iterative methods for linear systems

See the notes, which introduce three methods for solving sparse linear systems iteratively:

- Jacobi method
- Gauss–Seidel method
- Successive over-relaxation (SOR) method

These methods do not require creating the matrix explicitly in memory, and for a sparse matrix require less computation than direct numerical linear algebra.

# Towards the conjugate gradient method

The conjugate gradient method is another iterative method that is widely used.

It can be applied to symmetric positive definite matrices $A$ where all the eigenvalues are positive. Such matrices frequently occur when discretizing PDEs.

If a matrix $A$ is negative definite, so all its eigenvalues are negative, than the conjugate gradient method can be applied to $-A$, which is SPD.

We begin by considering a simpler method that motivates the conjugate gradient method.

# Digression: symmetric positive definite (SPD)

A symmetric matrix $A \in \mathbb{R}^{m \times m}$ is positive definite (SPD) if

- It is symmetric: $A = A^{\mathsf{T}}$.
- For all nonzero vectors $x \in \mathbb{R}^m$, $x^{\mathsf{T}} A x > 0$.

# Descent methods for minimization problems

For a symmetric matrix $A \in \mathbb{R}^{m \times m}$ define the function $\phi : \mathbb{R}^m \to \mathbb{R}$ as

$$\phi(u) = \frac{1}{2} u^\mathsf{T} A u - u^\mathsf{T} f,$$

where $f \in \mathbb{R}^m$.

Suppose that $m = 2$ and

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix}, \qquad f = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

Then

$$\phi(u) = \frac{a_{11} u_1^2 + 2 a_{12} u_1 u_2 + a_{22} u_2^2}{2} - u_1 f_1 - u_2 f_2.$$

# Descent methods for minimization problems

The stationary point of $\phi(u)$ corresponds to $\nabla\phi(u) = 0$. For the $m = 2$ example, this is

$$\frac{\partial \phi}{\partial u_1} = a_{11} u_1 + a_{12} u_2 - f_1 = 0,$$
$$\frac{\partial \phi}{\partial u_2} = a_{12} u_1 + a_{22} u_2 - f_2 = 0.$$

Hence the stationary point solves the matrix equation

$$Au = f,$$

which also applies to general $m$. Call this solution $u^*$.

Thus finding the stationary point of $\phi$ is equivalent to solving the matrix equation.

# Descent methods for minimization problems

Write $u = u_* + \delta$ for $\delta \in \mathbb{R}^m$. The function can be written as

$$\begin{aligned}\phi(u_* + \delta) &= \frac{1}{2}(u_* + \delta)^\mathsf{T} A(u_* + \delta) - (u_* + \delta)^\mathsf{T} f \\ &= \frac{1}{2}u_*^\mathsf{T} A u_* + \delta^\mathsf{T} A u_* + \frac{1}{2}\delta^\mathsf{T} A\delta - u_*^\mathsf{T} f - \delta^\mathsf{T} f \\ &= \frac{1}{2}\delta^\mathsf{T} A\delta - \frac{1}{2}u_*^\mathsf{T} f.\end{aligned}$$

If $A$ is SPD, then $\delta^\mathsf{T} A\delta > 0$ for all $\delta \neq 0$.

Hence $u_*$ is a minimum of $\phi$. We could therefore find it by developing an iterative method to find this minimum.

# Steepest descent

From here on, we use subscripts to indicate the iteration number, as we will not need to reference individual vector components.

Start from an initial guess $u_0$, and construct $u_1, u_2, \ldots$ to approach the minimum.

At one estimate $u_{k-1}$, the vector $-\nabla\phi(u_{k-1})$ points in the direction of steepest descent.

Hence the next value could be chosen as

$$u_k = u_{k-1} - \alpha_{k-1}\nabla\phi(u_{k-1})$$

for some $\alpha_{k-1} \geq 0$.

# Steepest descent

Choose $\alpha_{k-1}$ as the solution of the minimization problem

$$\min_{\alpha} \phi(u_{k-1} - \alpha \nabla \phi(u_{k-1})).$$

The gradient is

$$\nabla \phi(u_{k-1}) = Au_{k-1} - f = -r_{k-1}$$

where $r_{k-1} = f - Au_{k-1}$ is the residual vector, *i.e.* the discrepancy between the LHS and the RHS of the linear system $Au = f$.

If $r_{k-1} = 0$, then $u_{k-1} = u_*$. The size of $r_{k-1}$ gives an indication of how close $u_{k-1}$ is to the solution.

# Steepest descent

For a general $u$ and $r$,

$$\phi(u+\alpha r) = \left(\frac{1}{2}u^{\mathsf{T}}Au - u^{\mathsf{T}}f\right) + \alpha(r^{\mathsf{T}}Au - r^{\mathsf{T}}f) + \frac{1}{2}\alpha^2 r^{\mathsf{T}}Ar.$$

Hence

$$\frac{d\phi(u+\alpha r)}{d\alpha} = r^{\mathsf{T}}Au - r^{\mathsf{T}}f + \alpha r^{\mathsf{T}}Ar.$$

Since $r = f - Au$, setting this to zero gives

$$\alpha = \frac{r^{\mathsf{T}}r}{r^{\mathsf{T}}Ar}$$

From here, we can write down the steepest descent algorithm.

# Steepest descent algorithm

Choose initial guess $u_0$ and tolerance $\epsilon > 0$
**for** $k = 1, 2, 3, \ldots$ **do**
    $r_{k-1} = f - Au_{k-1}$
    If $\|r_{k-1}\| < \epsilon$, then stop
    $\alpha_{k-1} = (r_{k-1}^{\mathsf{T}} r_{k-1})/(r_{k-1}^{\mathsf{T}} A r_{k-1})$
    $u_k = u_{k-1} + \alpha_{k-1} r_{k-1}$
**end for**

This algorithm requires computing two matrix–vector multiplications per iteration, shown in blue.

# Improvement to steepest descent algorithm

At each step we are computing $Ar_{k-1}$ to find $\alpha_{k-1}$. In addition, we are computing the residual

$$r_{k-1} = f - Au_{k-1}.$$

Note however that

$$\begin{aligned} r_k &= f - Au_k = f - A(u_{k-1} + \alpha_{k-1} r_{k-1}) \\ &= r_{k-1} - \alpha_{k-1} Ar_{k-1}. \end{aligned}$$

Since we already need to compute $Ar_{k-1}$, we can reuse it to accelerate the computation of $r_k$, without calculating $Au_{k-1}$ separately.

# Steepest descent algorithm (improved)

Choose initial guess $u_0$ and tolerance $\epsilon > 0$
$r_0 = f - Au_0$
**for** $k = 1, 2, 3, \ldots$ **do**
  $w_{k-1} = Ar_{k-1}$
  $\alpha_{k-1} = (r_{k-1}^{T} r_{k-1})/(r_{k-1}^{T} w_{k-1})$
  $u_k = u_{k-1} + \alpha_{k-1} r_{k-1}$
  $r_k = r_{k-1} - \alpha_{k-1} w_{k-1}$
  If $\|r_k\| < \epsilon$, then stop
**end for**

# Steepest descent example

The program *s_descent.py* implements the steepest descent algorithm using

$$A = \begin{pmatrix} 3 & 0.8 \\ 0.8 & 1.2 \end{pmatrix}, \qquad f = \begin{pmatrix} 4 \\ 6 \end{pmatrix}.$$

This has solution

$$u_* = \begin{pmatrix} 0 \\ 5 \end{pmatrix}.$$

The program reaches a tolerance of $10^{-10}$ in 43 iterations. It takes a zig-zag path to reach the solution.

The level sets of $\phi(u)$ are ellipses. The steepest descent directions are not ideal for finding the minimum.

# Steepest descent example

If the program is modified to run on

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, \qquad f = \begin{pmatrix} 4 \\ 6 \end{pmatrix},$$

then it finds the solution

$$u_* = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

in a single iteration.

This demonstrates how the choice of direction can have a large effect on the efficiency.

# Geometrical analysis

For an elliptical contour of $\phi(u)$, define $v_1$ and $v_2$ on the major and minor axes.

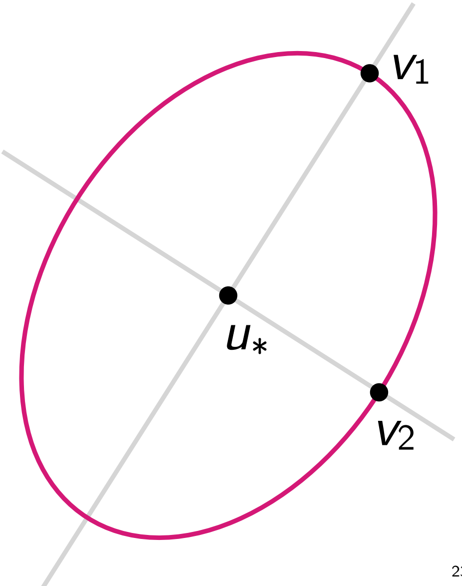
Then $\nabla\phi(v_j)$ lies in the direction of $u_*$, *i.e.*

$$\nabla\phi(v_j) = Av_j - f = \lambda_j(v_j - u_*).$$

Since $Au_* = f$, it follows that

$$A(v_j - u_*) = \lambda_j(v_j - u_*)$$

and hence $v_j - u_*$ is an eigenvector of $A$ with eigenvalue $\lambda_j$.



# Geometrical analysis

Consider the $v_1$ and $v_2$ defined on $\phi(u) = 1$. Then

$$\frac{1}{2} v_j^{\mathsf{T}} A v_j - v_j^{\mathsf{T}} A u_* = 1.$$

From the previous relationship

$$\begin{aligned} \|v_j - u_*\|_2^2 &= (v_j - u_*)^{\mathsf{T}} (v_j - u_*) \\ &= \frac{(v_j - u_*)^{\mathsf{T}} A (v_j - u_*)}{\lambda_j} = \frac{2 + u_*^{\mathsf{T}} A u_*}{\lambda_j}. \end{aligned}$$

Hence the ratio of the length of the major and minor axes is

$$\frac{\|v_1 - u_*\|_2}{\|v_2 - u_*\|_2} = \sqrt{\frac{\lambda_2}{\lambda_1}} = \sqrt{\kappa_2(A)},$$

where $\kappa_2(A)$ is the condition number in the 2-norm.