MATH3511/6111: Scientific Computing

12. Conjugate Gradient Method

Lindon Roberts

Semester 1, 2022

Office: Hanna Neumann Building #145, Room 4.87
Email: lindon.roberts@anu.edu.au
Based on lecture notes written by S. Roberts, L. Stals, Q. Jin, M. Hegland, K. Duru.

Australian National University

Conjugate Gradient Method

In this section we will look at a completely different type of iterative method for solving linear systems: the conjugate gradient (CG) method.

Unlike splitting methods (Jacobi, Gauss-Seidel, SOR), we do not need to know the entries of A. Instead, we only interact with A through calculating matrix-vector products $A\boldsymbol{x}$.

CG was developed by Magnus Hestenes and Eduard Stiefel in 1952 and is one of the most important algorithms in scientific computing. It is the method of choice in many circumstances, particularly for linear systems coming from solving partial differential equations.

"*For guidance to the future [of research], we should study not Gaussian elimination and its beguiling stability properties, but the diabolically fast conjugate gradient iteration*" — L. N. Trefethen, 1997.

Linear Operators

One advantage of CG is that we don't need to have A written down as an $n \times n$ matrix.

All we need is the ability to evaluate matrix-vector products: given $\boldsymbol{x}$, compute $A\boldsymbol{x}$. This is sometimes very convenient. For example,

$$y_i = (-x_{i-1} + 2x_i - x_{i+1})/h^2, \qquad i = 1, \ldots, n,$$

is a linear transformation $\boldsymbol{x} \to \boldsymbol{y}$ (approximating second derivatives), but we can write code to calculate $\boldsymbol{y}$ without constructing the corresponding matrix.

If A is sparse and we store just the nonzero entries of A, then it is usually very quick to evaluate $A\boldsymbol{x}$ — the number of flops is only proportional the number of nonzero entries of A.

Later in the course, we will look at the Fast Fourier Transformation, which computes $\boldsymbol{y} = F\boldsymbol{x}$ for a special (but dense!) $n \times n$ matrix F in $\mathcal{O}(n \log n)$ flops — much faster than $\mathcal{O}(n^2)$ flops for standard matrix-vector multiplication.

Symmetric Positive Definite Matrices

The conjugate gradient method is used to solve $A\boldsymbol{x} = \boldsymbol{b}$ when A is symmetric positive definite.

Symmetric matrices always have real eigenvalues, and their eigenvectors form an orthonormal basis for $\mathbb{R}^n$. This gives us the eigendecomposition

$$A = Q\Lambda Q^T,$$

where Λ is a diagonal matrix with entries $\lambda_i(A)$ and Q is an orthogonal matrix ($Q^TQ = I$ or Q has orthonormal columns) whose columns are given by the corresponding eigenvectors of A.

A matrix is positive definite if $\boldsymbol{x}^TA\boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$. If A is symmetric, this means that all its eigenvalues are strictly positive, $\lambda_i(A) > 0$.

Note: all positive definite matrices are invertible.

Symmetric Positive Definite Matrices

An important property of symmetric positive definite matrices is that they define an inner product on $\mathbb{R}^n$ defined by

$$(\boldsymbol{x}, \boldsymbol{y})_A = \boldsymbol{x}^T A \boldsymbol{y},$$

and also a vector norm (sometimes called the "energy norm" or A-norm)

$$\|\boldsymbol{x}\|_A = \sqrt{\boldsymbol{x}^T A \boldsymbol{x}}.$$

Since $\boldsymbol{x}^T A \boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$, the quantity inside the square root is never negative.

An Optimisation Problem

The first important idea for the conjugate gradient method is to think about solving $A\boldsymbol{x} = \boldsymbol{b}$ as an optimisation problem. Suppose the true solution is $\boldsymbol{x}$.

We want to find $\boldsymbol{x}$. Alternatively, we want to find a vector $\boldsymbol{y}$ such that

$$\|\boldsymbol{y} - \boldsymbol{x}\|$$

is as small as possible. In an iterative scheme, we might produce a sequence $\boldsymbol{x}^k$ such that $\|\boldsymbol{x}^k - \boldsymbol{x}\| \to 0$.

It turns out that it is very hepful to look at this in the squared A-norm.

Goal

Find a vector $\boldsymbol{y}$ such that $\frac{1}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_A^2$ is as small as possible.

An Optimisation Problem

We can compute:

$$\frac{1}{2}\|\boldsymbol{y}-\boldsymbol{x}\|_A^2 = \frac{1}{2}(\boldsymbol{y}-\boldsymbol{x})^T A(\boldsymbol{y}-\boldsymbol{x}) = \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{y} - \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{x} - \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{y} + \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x}.$$

Since $\boldsymbol{x}$ is the true solution (i.e. $A\boldsymbol{x}=\boldsymbol{b}$) and A is symmetric (i.e. $\boldsymbol{x}^T A = \boldsymbol{x}^T A^T = \boldsymbol{b}^T$), we get

$$\frac{1}{2}\|\boldsymbol{y}-\boldsymbol{x}\|_A^2 = \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{y} - \boldsymbol{b}^T\boldsymbol{y} + \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x}.$$

The last term doesn't depend on $\boldsymbol{y}$, so we can ignore it (the best choice of $\boldsymbol{y}$ is the same with or without a constant).

Theorem

If A is symmetric positive definite, the minimiser $\boldsymbol{y}$ of the optimisation problem

$$\min_{\boldsymbol{y}\in\mathbb{R}^n} f(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{y} - \boldsymbol{b}^T\boldsymbol{y},$$

is the solution to $A\boldsymbol{x}=\boldsymbol{b}$.

Optimisation

How do we solve optimisation problems of the form

$$\min_{\boldsymbol{y}\in\mathbb{R}^n} f(\boldsymbol{y}),$$

where f is some differentiable function (in our case, f is a quadratic form)?

Recall the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the vector of first derivatives:

$$\nabla f(\boldsymbol{y}) = \begin{bmatrix} \partial f(\boldsymbol{y})/\partial y_1 \\ \vdots \\ \partial f(\boldsymbol{y})/\partial y_n \end{bmatrix}.$$

Suppose we currently have an estimate $\boldsymbol{y}^k$ of the minimiser, and we start moving in a direction $\boldsymbol{d}$. We can then look at the function values we get, depending on a 'step size' α. Using a multivariate version of Taylor's theorem we get

$$f(\boldsymbol{y}^k + \alpha\boldsymbol{d}) = f(\boldsymbol{y}^k) + \alpha\nabla f(\boldsymbol{y}^k)^T\boldsymbol{d} + \mathcal{O}(\alpha^2).$$

Steepest Descent

For a small step size $\alpha > 0$, we get

$$f(\boldsymbol{y}^k + \alpha \boldsymbol{d}) \approx f(\boldsymbol{y}^k) + \alpha \nabla f(\boldsymbol{y}^k)^T \boldsymbol{d},$$

and so f will be decreased provided $\nabla f(\boldsymbol{y}^k)^T \boldsymbol{d} < 0$ (we say $\boldsymbol{d}$ is a descent direction).

In fact, for small $\alpha > 0$, f will be decreased fastest in the direction $\boldsymbol{d} = -\nabla f(\boldsymbol{y}^k)$. Then we have $\nabla f(\boldsymbol{y}^k)^T \boldsymbol{d} = -\|\nabla f(\boldsymbol{y}^k)\|_2^2 < 0$. This is called the direction of steepest descent.

What step size α should we use? Ideally, we search along the direction $\boldsymbol{d}$ to find the point that gives the smallest value of f:

$$\min_{\alpha > 0} f(\boldsymbol{y}^k + \alpha \boldsymbol{d}).$$

We can find this α by solving:

$$\frac{d}{d\alpha} f(\boldsymbol{y}^k + \alpha \boldsymbol{d}) = \nabla f(\boldsymbol{y}^k + \alpha \boldsymbol{d})^T \boldsymbol{d} = 0.$$

Steepest Descent

These choices of step direction $\boldsymbol{d}$ and size α give us an optimisation algorithm:

$$\boxed{\boldsymbol{y}^{k+1} = \boldsymbol{y}^k + \alpha_k \boldsymbol{d}^k, \qquad \text{where } \boldsymbol{d}^k = -\nabla f(\boldsymbol{y}^k) \text{ and } \alpha_k \text{ satisfies } \quad \nabla f(\boldsymbol{y}^k + \alpha_k \boldsymbol{d}^k)^T \boldsymbol{d}^k = 0}$$

For general functions f, usually finding this α_k is too difficult, so instead we just search for a "good" α_k — this is called a linesearch method.

Question

What does this look like for our function, $f(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{y} - \boldsymbol{b}^T\boldsymbol{y}$?

We can compute $\boxed{\nabla f(\boldsymbol{y}) = A\boldsymbol{y} - \boldsymbol{b}}$, and so $\boldsymbol{d}^k$ is the residual $\boldsymbol{d}^k = \boldsymbol{r}^k = \boldsymbol{b} - A\boldsymbol{y}^k$. Then,

$$0 = \nabla f(\boldsymbol{y}^k + \alpha_k \boldsymbol{d}^k)^T \boldsymbol{d}^k = [A(\boldsymbol{y}^k + \alpha \boldsymbol{d}^k) - \boldsymbol{b}]^T \boldsymbol{d}^k = (A\boldsymbol{y}^k - \boldsymbol{b})^T \boldsymbol{d}^k + \alpha (\boldsymbol{d}^k)^T A \boldsymbol{d}^k.$$

That is, we get

$$\alpha = \frac{(\boldsymbol{r}^k)^T \boldsymbol{d}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k}.$$

Steepest Descent

All together, we get

$$\boldsymbol{y}^{k+1} = \boldsymbol{y}^k + \alpha_k \boldsymbol{r}^k, \qquad \text{where } \boldsymbol{r}^k = \boldsymbol{b} - A\boldsymbol{y}^k \text{ and } \alpha_k = \frac{(\boldsymbol{r}^k)^T \boldsymbol{r}^k}{(\boldsymbol{r}^k)^T A \boldsymbol{r}^k}.$$

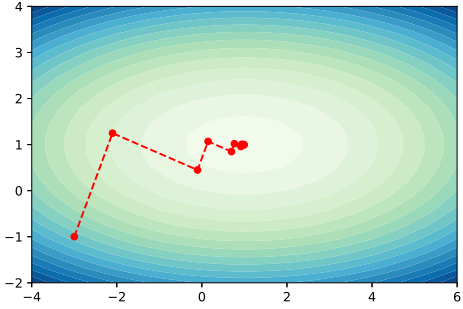
```
import numpy as np

def steepest_descent(A, b, y0, tolr):
    # Solve Ay=b using the steepest descent optimisation algorithm
    y = y0.copy()  # set initial iterate
    r = b - A @ y  # set initial residual

    while np.linalg.norm(r) > tolr:  # continue until residual is small
        alpha = (r.T @ r) / (r.T @ A @ r)  # calculate step size
        y = y + alpha*r  # update iterate
        r = b - A @ y    # calculate new residual

    return y
```

Steepest Descent

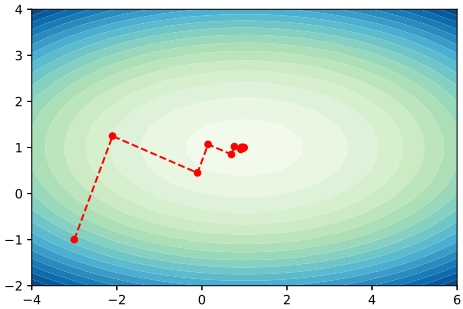
Let's use this method to solve the simple symmetric positive definite linear system

$$\begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

starting from $\boldsymbol{y}^0 = \begin{bmatrix} -3 & -1 \end{bmatrix}^T$.



Steepest Descent



Steepest descent tends to have zig-zagging behaviour, and we take many small steps in the same directions.

What if we could combine the steps in the same direction, and just take fewer, larger steps? We would converge more quickly. (this is a bit like acceleration in SOR)

Conjugate Gradients

To reduce zig-zagging behaviour, we don't want to take steps in the same direction multiple times. However, we can't use just any directions, because knowing the correct α_k is very difficult.

So, we make sure each step direction is orthogonal ("conjugate") to all previous step directions. This is the second key idea of the CG method.

It turns out that this is easier to do if we mean orthogonal in the A-inner product rather than the usual dot product. That is, we choose our $\boldsymbol{d}^k$ such that

$$(\boldsymbol{d}^k, \boldsymbol{d}^i)_A = (\boldsymbol{d}^k)^T A \boldsymbol{d}^i = 0, \qquad \forall i = 1, \ldots, k-1.$$

Question

How can we generate descent directions which are all A-conjugate?

Conjugate Gradients

Question

How can we generate descent directions which are all A-conjugate?

To do this, we choose to make $\boldsymbol{d}^k$ a specific linear combination of $\boldsymbol{r}^k$ and $\boldsymbol{d}^{k-1}$:

$$\boxed{\boldsymbol{d}^k = \boldsymbol{r}^k + \beta_k \boldsymbol{d}^{k-1}}$$

and calculate β_k from the condition $(\boldsymbol{d}^k, \boldsymbol{d}^{k-1})_A = 0$:

$$0 = (\boldsymbol{d}^k)^T A \boldsymbol{d}^{k-1} = (\boldsymbol{r}^k)^T A \boldsymbol{d}^{k-1} + \beta_k (\boldsymbol{d}^{k-1})^T A \boldsymbol{d}^{k-1},$$

and so we get

$$\boxed{\beta_k = -\frac{(\boldsymbol{r}^k)^T A \boldsymbol{d}^{k-1}}{(\boldsymbol{d}^{k-1})^T A \boldsymbol{d}^{k-1}}}$$

Conjugate Gradients

We have made the choice

$$\boxed{\boldsymbol{d}^k = \boldsymbol{r}^k + \beta_k \boldsymbol{d}^{k-1}} \qquad \text{where} \qquad \boxed{\beta_k = -\frac{(\boldsymbol{r}^k)^T A \boldsymbol{d}^{k-1}}{(\boldsymbol{d}^{k-1})^T A \boldsymbol{d}^{k-1}}}$$

From the previous slide, we know that we get $(\boldsymbol{d}^k, \boldsymbol{d}^{k-1})_A = 0$.

What about the other orthogonality conditions we wanted: $(\boldsymbol{d}^k, \boldsymbol{d}^i)_A = 0$ for $i < k-1$?

Theorem

The direction $\boldsymbol{d}^k$ chosen as above is A-orthogonal to all previous directions:

$$(\boldsymbol{d}^k, \boldsymbol{d}^i)_A = 0, \qquad \forall i = 1, \ldots, k-1.$$

This is the key insight that makes CG so effective: we can very easily generate A-orthogonal directions.

Conjugate Gradients

Writing this out in full (using $\boldsymbol{x}^k$ instead of $\boldsymbol{y}^k$) we have:

Algorithm 1 Conjugate Gradient method for solving $A\boldsymbol{x} = \boldsymbol{b}$.

Input: Symmetric positive definite matrix A, right-hand side $\boldsymbol{b}$, starting guess $\boldsymbol{x}^0$.

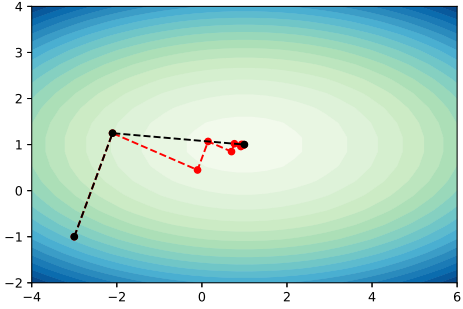
1: Define $\boldsymbol{r}^0 = \boldsymbol{b} - A\boldsymbol{x}^0$ and set $\boldsymbol{d}^0 = \boldsymbol{r}^0$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Set step length $\alpha_k = \frac{(\boldsymbol{r}^k)^T \boldsymbol{d}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k}$.
4: Calculate new iterate $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \alpha_k \boldsymbol{d}^k$.
5: Calculate new residual $\boldsymbol{r}^{k+1} = \boldsymbol{b} - A\boldsymbol{x}^{k+1}$.
6: If $\|\boldsymbol{r}^{k+1}\|$ is small enough, terminate and return $\boldsymbol{x}^{k+1}$.
7: Set $\beta_{k+1} = -\frac{(\boldsymbol{r}^{k+1})^T A \boldsymbol{d}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k}$.
8: Calculate new search direction $\boldsymbol{d}^{k+1} = \boldsymbol{r}^{k+1} + \beta_{k+1} \boldsymbol{d}^k$.
9: **end for**

Conjugate Gradients: Example

Let's use CG to solve the simple linear system from before.

$$\begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

starting from $\boldsymbol{y}^0 = \begin{bmatrix} -3 & -1 \end{bmatrix}^T$. CG shown in black, steepest descent in red.



Conjugate Gradients

With some manipulation, we can slightly simplify the CG method.

It turns out that

$$\alpha_k = \frac{(\boldsymbol{r}^k)^T \boldsymbol{d}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k} = \frac{(\boldsymbol{r}^k)^T \boldsymbol{r}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k},$$

and

$$\beta_k = -\frac{(\boldsymbol{r}^{k+1})^T A \boldsymbol{d}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k} = \frac{(\boldsymbol{r}^{k+1})^T \boldsymbol{r}^{k+1}}{(\boldsymbol{r}^k)^T \boldsymbol{r}^k}.$$

Also, we have

$$\boldsymbol{r}^{k+1} = \boldsymbol{b} - A\boldsymbol{x}^{k+1} = \boldsymbol{b} - A(\boldsymbol{x}^k + \alpha_k \boldsymbol{d}^k) = \boldsymbol{r}^k - \alpha_k A \boldsymbol{d}^k.$$

Making these substitutions, we get the 'classic' version of CG:

Algorithm 2 Conjugate Gradient method for solving $A\boldsymbol{x} = \boldsymbol{b}$.

Input: Symmetric positive definite matrix A, right-hand side $\boldsymbol{b}$, starting guess $\boldsymbol{x}^0$.

1: Define $\boldsymbol{r}^0 = \boldsymbol{b} - A\boldsymbol{x}^0$ and set $\boldsymbol{d}^0 = \boldsymbol{r}^0$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Set step length $\alpha_k = \frac{(\boldsymbol{r}^k)^T \boldsymbol{r}^k}{(\boldsymbol{d}^k)^T A \boldsymbol{d}^k}$.
4: Calculate new iterate $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \alpha_k \boldsymbol{d}^k$.
5: Calculate new residual $\boldsymbol{r}^{k+1} = \boldsymbol{r}^k - \alpha_k A \boldsymbol{d}^k$.
6: If $\|\boldsymbol{r}^{k+1}\|$ is small enough, terminate and return $\boldsymbol{x}^{k+1}$.
7: Set $\beta_{k+1} = \frac{(\boldsymbol{r}^{k+1})^T \boldsymbol{r}^{k+1}}{(\boldsymbol{r}^k)^T \boldsymbol{r}^k}$.
8: Calculate new search direction $\boldsymbol{d}^{k+1} = \boldsymbol{r}^{k+1} + \beta_{k+1} \boldsymbol{d}^k$.
9: **end for**

Written this way, we only need A to calculate $A\boldsymbol{d}^k$ at each iteration.

Conjugate Gradients: Convergence

The most important convergence result is the following.

Theorem

In exact arithmetic, CG converges to the true solution in exactly n iterations.

In practice, rounding errors mean CG generally does not *exactly* converge in n iterations. However, we often don't need to run CG for many iterations to get a small residual (often much less than n).

The rate of convergence of CG depends on the eigenvalues of A:

- If A has k distinct eigenvalues, then CG converges to the true solution in exactly k iterations.
- If the eigenvalues of A are close together, or clumped around a small number of values, then CG generally converges quickly.

More details about the convergence of CG are covered in MATH3512.

Preconditioning

CG converges faster if the eigenvalues of A are reasonably close together, or clumped.

One important topic is preconditioning: replacing a linear system $A\boldsymbol{x} = \boldsymbol{b}$ with an equivalent system, such as

$$(P^{-1}A)\boldsymbol{x} = P^{-1}\boldsymbol{b} \qquad \text{or} \qquad (AP^{-1})(P\boldsymbol{x}) = \boldsymbol{b},$$

for which methods like CG will converge faster (i.e. $P^{-1}A$ has a better distribution of eigenvalues than A), but where P^{-1} is easy to compute.

There is a lot of knowledge about designing good preconditioners for certain types of matrices A, particularly those coming from the solution of differential equations. Examples include

- Techniques based on splitting methods, like Jacobi ($P = D$) or SOR.
- Incomplete LU: find $A \approx LU$ (i.e. L, U triangular but $A \neq LU$).
- Multigrid: solve a differential equation on grids of different sizes.

Krylov Subspace Methods

CG is an example of a Krylov subspace method (Alexei Krylov, 1931). The Krylov subspace of degree k is the subspace given by

$$\mathcal{K}_k = \text{span}\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, \ldots, A^{k-1}\boldsymbol{b}\}.$$

It can be proven that $\boldsymbol{x}^k$ from CG is the vector in $\mathcal{K}_k$ which minimises the error $\|\boldsymbol{e}_k\|_A$.

In fact, our choice of directions $\boldsymbol{d}^0, \boldsymbol{d}^1, \ldots, \boldsymbol{d}^{k-1}$ form an A-orthogonal basis for $\mathcal{K}_k$.

Several other important matrix algorithms are based on calculating in Krylov subspaces:

- MINRES/SQMR and GMRES/BiCGStab/QMR for $A\boldsymbol{x} = \boldsymbol{b}$: like CG, but for symmetric (MINRES, 1975; SQMR, 1994) and general (GMRES, 1986; QMR, 1991; BiCGStab, 1994) matrices.
- Several methods for computing eigenvalues of matrices (Lanczos, 1950; Arnoldi, 1951).