Rough notes on:

Conjugate gradient algorithm

by Francisco Lobo

These notes were initiated with the purpose of gaining a deeper understanding of the internal algorithmic structure underlying the implementation of the Usadel formalism made for our paper "Fluxoid solitons in superconducting tapered tubes and bottlenecks". I primarily consulted the corresponding Wikipedia articles and some lecture videos, namely "https://www.youtube.com/watch?v=AguykhI5aTA."

A. Linear conjugate gradient algorithm

1. *Minimization problem*

Consider we wish to minimize, an n dimensional function f. This amounts to solving

$$\nabla f(\boldsymbol{x}) = 0. \qquad [1]$$

Let us assume for now this f is linear in $\boldsymbol{x} = (x^1, x^2, ..., x^n)$ such that it can be expressed generically as

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b} \qquad [2]$$

with $\boldsymbol{A}$ a nxn symmetric (i.e. $A = A^T$) and positive-defined (i.e $\boldsymbol{x}^T A\boldsymbol{x} > 0$) matrix and $\boldsymbol{b}$ a nx1 vector. If this is our case, then our problem can be expressed with the linear problem

$$\nabla f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = 0 \qquad [3]$$

See Appendix C 1] for more details. Let us denote $\boldsymbol{x}_*$ as the solution, uniquely defined by $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$.

One can attribute this linear problem to many physically system, the one that comes to mind first to most being Hook's law $\boldsymbol{k}\boldsymbol{x} = \boldsymbol{F}$, with $\boldsymbol{k}$ being the Hooks/stiffness tensor, $\boldsymbol{x}$ the displacement/response and $\boldsymbol{F}$ the forcing/driving term. Moreover, within this classical mechanics overview, see that the fact that we assumed $\boldsymbol{A}$ to symmetric and positive-defined is also physically intuitive since $\boldsymbol{A}$, representing stiffnesses, is inherently positive quantities. The energy stored in this system, corresponds to the function f (first term corresponding to the elastic potential energy stored in the springs and the second the work done by external forces), is then always positive definite and it is zero only when the field or displacement itself is zero. The system cannot have multiple equilibrium states (those who minimize the energy) for the same driving thus being uniquely defined. In all cases, one finds that $\boldsymbol{A}$ encodes the medium's response properties such as stiffness, conductivity, permittivity, etc..., $\boldsymbol{b}$ represents the applied load or source, and f the a potential functional whose minimization gives the equilibrium configuration of that linear system. Just to name a few more, in the context of heat transfer we can write it as the heat diffusion equation $[-\nabla\cdot(k\nabla)]\,T = q$ with the linear operator being the discrete Laplacian operator with k thermal conductivity, T temperature and q the heat source, or, for example, in the context of electrostatics it corresponds Poisson's equation $-\nabla^2\phi = \rho$ with ϕ the potential field and ρ charge density.

2. *Gradient descent algorithm*

As a first attempt to solve this linear problem as a minimization problem, let us first understand the logic behind the simplest of this algorithms, the gradient descent algorithm. We start by considering an initial guess $\boldsymbol{x}_0$ for the minimum. We then compute the negative gradient of the function at this point, i.e $\nabla f(\boldsymbol{x}_0) = \boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b}$, since it will point in the direction of decreasing values of f. We then take this section of $\boldsymbol{x}$ along the gradient line, and find that curve minimum at $\boldsymbol{x}_1$ using, for example, Newtons method (assuming originally it was a 3D surface function). From here on what we repeat this process noting that the gradient direction $\nabla f(\boldsymbol{x}_1)$ is, by construction, orthogonal to the previous one, $\nabla f(\boldsymbol{x}_1)$ (or else we would have been able to move a little more along the previous gradient direction). This zigzag iterative process will eventually lead us to the minimum of our surface. The conjugate gradient algorithm comes next, to avoid these unnecessary zigzag pattern. In simple terms, what this does, is that it allows us, in practice, to find the minimum instead in just n different directions since progress made in one direction does not affect progress made in another direction.

3. *A-orthogonal basis*

For two vectors to be conjugate, or in another terms A-orthogonal, they must satisfy

$$\langle \boldsymbol{p}_i^T, \boldsymbol{p}_k \rangle_{\boldsymbol{A}} = \boldsymbol{p}_i^T\boldsymbol{A}\boldsymbol{p}_k = 0, \qquad [4]$$

such that a set of n of this vectors, $\{\boldsymbol{p}_1, \boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ will form a base of $\mathbb{R}^n$. In this basis, we can represent the

any $\boldsymbol{x}$ as

$$\boldsymbol{x} = \sum_{i=1} \alpha_i \boldsymbol{p}_i \quad [5]$$

with the α_i proportions found via

$$\boldsymbol{p}_k^T (\boldsymbol{A}\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_i \quad [6]$$

such that, and using the fact that $\boldsymbol{p}_k^T$ and $\boldsymbol{p}_i$ are conjugate, it reads

$$\alpha_k = \frac{\boldsymbol{p}_k^T \boldsymbol{b}}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k} \quad [7]$$

4. Iterative algorithm

One could, at this point, find the various $\boldsymbol{p}$ directions and their respective α by solving for its eigenvalues however this is not optimal. Instead, we can generate them iteratively. Give some initial guess $\boldsymbol{x}_0$, the first most optimal direction to move is still the negative gradient at that point, i.e $\boldsymbol{p}_0 = -\nabla f(\boldsymbol{x}_0) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$. We can then calculate the corresponding α_0 through Eq.(7) and get the corresponding next guess as $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0$. At this point, the gradient descent algorithm would indicate us to move, for the next steps, always in the negative gradient directions, i.e

$$\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k = -\nabla f(\boldsymbol{x}_k) \quad [8]$$

however, in the conjugate gradient algorithm we insist that the directions $\boldsymbol{p}_k$ must be conjugate to each other. A way to enforce this is by requiring that the next search direction be built out of the current residual and all previous search directions. The conjugation constraint is an orthonormal-type constraint and hence the algorithm can be viewed as an example of Gram-Schmidt orthonormalization. This gives the following expression

$$\boldsymbol{p}_k = \boldsymbol{r}_k - \sum_{i<k} \frac{\boldsymbol{r}_k^T \boldsymbol{A} \boldsymbol{p}_i}{\boldsymbol{p}_i^T \boldsymbol{A} \boldsymbol{p}_k} \boldsymbol{p}_i \quad [9]$$

Check Appendix C 2 for more details. Also, we note that in the context of the CG algorithm, these $\boldsymbol{r}_k$ directions are often called residuals of the kth step, since if we consider $\boldsymbol{x}_k$ as an approximate solution of $\boldsymbol{x}_*$ then $\nabla f(\boldsymbol{x}_k)$ is not exactly equal to zero but close. Following this $\boldsymbol{p}_k$direction, the next optimal location is given by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k \text{ with } \alpha_k = \frac{\boldsymbol{p}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k} \quad [10]$$

Check Appendix C 3 for more details.

5. Linear conjugate gradient algorithm

The above subsections gives the most straightforward explanation of the conjugate gradient method. Seemingly, the algorithm requires storage of all previous search directions and residual vectors, as well as many matrix–vector multiplications, and thus can be computationally expensive. However, a closer analysis of the algorithm shows that $\mathbf{r}_i$ is orthogonal to $\mathbf{r}_j$, i.e.

$$\mathbf{r}_i^\mathsf{T} \mathbf{r}_j = 0, \quad \text{for } i \neq j.$$

Similarly, $\mathbf{p}_i$ is $\mathbf{A}$-orthogonal to $\mathbf{p}_j$, i.e.

$$\mathbf{p}_i^\mathsf{T} \mathbf{A} \mathbf{p}_j = 0, \quad \text{for } i \neq j.$$

This can be regarded as, as the algorithm progresses, $\mathbf{p}_i$ and $\mathbf{r}_i$ span the same Krylov subspace, where the residuals $\{\mathbf{r}_i\}$ form an orthogonal basis with respect to the standard inner product, and the directions $\{\mathbf{p}_i\}$ form an orthogonal basis with respect to the inner product induced by $\mathbf{A}$. Therefore, $\mathbf{x}_k$ can be regarded as the projection of $\mathbf{x}$ onto the Krylov subspace. That is, if the conjugate gradient (CG) method starts with $\mathbf{x}_0 = 0$, then

$$\begin{aligned}\mathbf{x}_k = &\text{argmin}_{\mathbf{y} \in \mathbb{R}^n} \left\{ (\mathbf{x} - \mathbf{y})^\top \mathbf{A} (\mathbf{x} - \mathbf{y}) : \right. \\ &\left. \mathbf{y} \in \text{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^{k-1}\mathbf{b}\} \right\}.\end{aligned}$$

The broadstroke of the actual algorithm is detailed in Fig.(1).

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
  $\alpha_k := \dfrac{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}{\mathbf{p}_k^\mathsf{T} \mathbf{A} \mathbf{p}_k}$
  $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
  $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$
  if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
  $\beta_k := \dfrac{\mathbf{r}_{k+1}^\mathsf{T} \mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}$
  $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
  $k := k + 1$
end repeat
return $\mathbf{x}_{k+1}$ as the result

Figure 1. Conjugate gradient algorithm

Note that, in practice, for the CG algorithm, we do not need to store the full matrix $\boldsymbol{A}$ explicitly ($\boldsymbol{b}$, on the other hand, being the "applied force" term, is generally know). For large systems $\boldsymbol{A}$ can be enormous and thus storing $\boldsymbol{A}$ as a dense matrix would be computationally wasteful. Instead, we often use sparse matrix representations (meaning that zeros are not stored explicitly) and use matrix-free implementations, where $\boldsymbol{A}\boldsymbol{x}$ is computed by applying

an operator (like a differential stencil) directly to $\boldsymbol{x}$. So, for example, suppose that $\boldsymbol{A}$ represents the discretized Laplace operator $-\nabla^2$ in 2D, instead of constructing $\boldsymbol{A}$ explicitly, we can compute $\boldsymbol{Ax}$ by the finite differences $(\boldsymbol{Ax})_{i,j} = 4x_{i,j} - x_{i+1,j} - x_{i-1,j} - x_{i,j+1} - x_{i,j-1}$. This fits perfectly within the CG algorithm because in practice we only compute $\boldsymbol{Ap}_k$, not $\boldsymbol{A}$ directly.

Note that, in exact arithmetic, CG converges in at most n steps although, in practice, were we allow for rounding errors, it often converges much faster. Concretely, it converges roughly in a number of iterations proportional to the square root of the condition number of A with the condition number of A measuring how sensitive the solution is to small changes in $\boldsymbol{b}$ or rather, to rounding errors in computation. This condition number is formally defined as the ration between the largest eigenvalue of $\boldsymbol{A}$, $\lambda_{\max}(\boldsymbol{A})$, and its smallest $\lambda_{\min}(\boldsymbol{A})$, i.e. $\kappa(\boldsymbol{A}) = \lambda_{\max}(\boldsymbol{A})/\lambda_{\min}(\boldsymbol{A})$. If κ is close to 1, the matrix is said to be well-conditioned and small errors in data or computation lead to small errors in the result. If κ is large, the matrix is ill-conditioned and even small perturbations can cause large changes in the solution meaning slower convergence and higher numerical instability.

Moreover, to accelerate convergence, a preconditioner M given as an approximation of A^{-1} can be introduced, leading to the so-called preconditioned conjugate gradient (PCG) method.

B. Nonlinear conjugate gradient algorithm

As we saw previously, in the *linear* CG algorithm, one seeks to solve a system of equations of the form $A\mathbf{x} = \mathbf{b}$ which is equivalent to minimizing the quadratic functional $f(\boldsymbol{x})$. Because $f(\mathbf{x})$ is quadratic, its level sets are ellipsoids, and the gradient $\nabla f = A\mathbf{x} - \mathbf{b}$ is a linear function of $\mathbf{x}$. The elegance of the CG method lies in its ability to construct, at each step, a search direction that is *conjugate* with respect to A, ensuring that successive minimization directions do not interfere with one another. This property guarantees convergence to the exact minimum in at most n steps for an n-dimensional problem. However, when extending this reasoning to *nonlinear* systems or general optimization problems where $f(\mathbf{x})$ is not quadratic, the geometry of the objective function becomes substantially more complex. The curvature of the energy landscape varies from point to point, and thus there is no single, fixed matrix A defining the local shape of $f(\mathbf{x})$. The *nonlinear CG method* generalizes the same geometric principle: it seeks to move along locally conjugate directions that adapt to the current curvature of the landscape (which now can vary), while relying solely on gradient information, without requiring explicit computation of the Hessian matrix. The algorithm continually updates an approximate conjugacy, yielding much faster convergence than simple gradient descent, particularly when the objective function exhibits elongated contours or strong anisotropy.

A key component distinguishing the nonlinear CG method from its linear counterpart is the line search. Because the curvature varies, one cannot predict the optimal step size analytically as in the linear case. A line search is therefore conducted at each iteration to determine how far one should move along the current direction. This process ensures that the function value decreases sufficiently (the Armijo condition), and that the new gradient is not too misaligned with the previous direction (the curvature condition). Together, these form the so-called Wolfe conditions, which are central to guaranteeing both convergence and stability of the algorithm. With a linear function the minimum is reached within n iterations (apart from roundoff error), but a nonlinear function will make slower progress since subsequent search directions lose conjugacy requiring the search direction to be reset to the steepest descent direction at least every n iterations, or sooner if progress stops.

C. Appendix

1. *Proof of gradient of linear function*

In Einstein notation, differentiating with respect to $\boldsymbol{x}$ reads as

$$\nabla(\boldsymbol{x}^T\boldsymbol{b}) = \frac{\partial}{\partial x_j}\left(\sum_i x_i b_i\right) = b_j = \boldsymbol{b}\,\blacksquare$$

Analogously, one also has that

$$\begin{aligned}
\nabla(\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}) =& \frac{\partial}{\partial x_k}\left(\sum_i\sum_j x_i A_{ij} x_j\right)\\
=& \sum_{i,j} A_{ij}\frac{\partial(x_i x_j)}{\partial x_k}\\
=& \sum_{i,j} A_{ij}(\delta_{ik}x_j + x_i\delta_{jk})\\
=& \sum_j A_{kj}x_j + \sum_i A_{ik}x_i\\
=& (\boldsymbol{Ax})_k + (\boldsymbol{A}^T\boldsymbol{x})_k\\
=& (\boldsymbol{A} + \boldsymbol{A}^T)\boldsymbol{x}\\
=& 2\boldsymbol{Ax}\,\blacksquare
\end{aligned}$$

where δ_{ik} is the Kronecker delta and where, in the last step, we used the fact that $\boldsymbol{A}$ is a symmetric matrix.

2. *Proof of $\boldsymbol{p}_k$ directions expression*

3. *Proof of α_k indices expression*

The expression for α_k can be derived if one substitutes the expression for $\boldsymbol{x}_{k+1}$ into f, such that

$$f(\boldsymbol{x}_{k+1}) = f(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k)$$

and minimizes it with respect to α_k as follows

$$
\begin{aligned}
&\nabla_{\alpha_k}\left[f(\boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k)\right]=0\\
\Leftrightarrow&\nabla_{\alpha_k}\left[\frac{1}{2}\left(\boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k\right)^T\boldsymbol{A}\left(\boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k\right)-\left(\boldsymbol{x}_k+\alpha_k\boldsymbol{p}_k\right)^T\boldsymbol{b}\right]=0\\
\Leftrightarrow&\alpha_k\left(\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k\right)+\frac{1}{2}\left(\boldsymbol{x}_k^T\boldsymbol{A}\boldsymbol{p}_k\right)+\frac{1}{2}\left(\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{x}_k\right)-\boldsymbol{p}_k^T\boldsymbol{b}=0\\
\Leftrightarrow&\alpha_k\left(\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k\right)+\boldsymbol{p}_k^T\left(\boldsymbol{A}\boldsymbol{x}_k-\boldsymbol{b}\right)=0\\
\Leftrightarrow&\alpha_k=\frac{\boldsymbol{p}_k^T\boldsymbol{r}_k}{\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k}\ \blacksquare
\end{aligned}
$$

Note that, seemingly, this does not equal the same expression shown first in Eq.(7). However, see that one can relate the two expressions by expanding the residual analogously too Eq.(5) as

$$\boldsymbol{r}_k=\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}_k=\boldsymbol{b}-\boldsymbol{A}\sum_{i<k}\alpha_i\boldsymbol{p}_i$$

and therefore

$$\boldsymbol{p}_k^T\boldsymbol{r}_k=\boldsymbol{p}_k^T\boldsymbol{b}-\sum_{i<k}\alpha_i\boldsymbol{p}_k\boldsymbol{A}\boldsymbol{p}_i$$

However, due to $\boldsymbol{A}$-orthogonality, one has that $\boldsymbol{p}_k\boldsymbol{A}\boldsymbol{p}_i=0$ for all $i<k$, and thus, under $\boldsymbol{A}$-orthogonality, the two forms coincide.