# Otimização-não-linear-final-Ericles-9790687

27 de Dezembro de 2020

## 1 Método dos gradientes conjugados

### 1.1 Introdução

O método dos gradientes conjugados (daqui para frente abreviado como CG, da sigla em inglês) é o método iterativo mais popular para resolver sistemas grandes de equações lineares. Esse método é particularmente efetivo para resolver sistemas

$$Ax = b$$

onde $x$ é um vetor desconhecido (a incógnita), $b$ é conhecido e $A$ é uma matriz positiva-definida. Esse método é mais adequado para quando $A$ for esparsa: quando $A$ é densa, provavelmente um método mais adequado seria alguma fatoração (como a $LU$); mas quando $A$ é esparsa, suas fatorações geralmente são densas, impondo um custo de memória que pode ser inaceitável.

Apesar de CG ser usado para resolver sistemas de equações lineares, um melhor entendimento sobre o método vem ao considerarmos um problema de otimização quadrática

$$min \quad f(x) = \frac{1}{2}(x,x)_A - (b,x) + c$$

onde $(x,y)_A := (x,Ay)$ e $(x,y)$ é o produto interno usual. Temos que a derivada de $f$ em $x$, $D(f)(x)$, é $\frac{1}{2}(A' + A)x - b$, que, quando $A$ é simétrica, é $Ax - b$. O mínimo (ou máximo) $x^*$ da função é obtido onde sua derivada se anula, do que segue que resolvemos para$Ax^* - b = 0$, rendendo o sistema linear inicial. Se $A$, além de simétrica, é positiva-definida, $x^*$ é um mínimo. Se $A$ não for simétrica, CG vai buscar solução para $\frac{1}{2}(A' + A)x = b$ (note que $A' + A$ é simétrica).

Se $A$ é positiva-definida, sua forma bilinear correspondente $(x, A'x)$ (ou melhor, o seu gráfico) corresponde a um parabolóide, do que é intuitivo que $x^*$ é único.

### 1.2 Ideia geral do método

A ideia básica desse método (assim como a de outros métodos) é

1. Começando de um $x_0$;
2. Escolha uma direção $d_i$;
3. Escolha quanto se quer "andar"na direção $d_i$, determinando o tamanho de passo $\alpha$;
4. Faça $x_i := x_{i-1} + \alpha d_i$.

Assim, podemos definir os vetores

- Resíduo: $r_i := b - Ax_i$;
- Erro: $e_i := x_i - x$ (note que $r_i = Ae_i$)

Note que $r_i = -D(f)(x_i)$, que é a direção em $x_i$ que aponta onde a função está diminuindo mais rapidamente - que, o que é importante, é ortogonal à curva de nível em $x_i$.

O método de máxima descida, no qual CG se baseia, faz $d_i = r_{i-1}$, e busca um passo $\alpha$ que minimize a função nessa direção. Deve ser intuitivo que essa minimização ocorre quando $\alpha$ for tal que faça $r_{i-1} \perp r_i$ (de modo que $x_{i-1} + \alpha d_i$ tangencie uma curva de nível de $f$). Uma conta rápida mostra que $\alpha = \frac{(r_{i-1}, r_{i-1})}{(r_{i-1}, r_{i-1})_A}$.

O método de máxima descida no geral leva mais de uma iteração para encontrar a solução. Na verdade, máxima descida converge em apenas uma iteração apenas se o chute inicial $x_0$ estiver na direção de um dos auto-vetores de $A$, que coincide com um dos eixos do elipsóide (o que raramente ocorre, a não ser que $A$ seja múltipla da identidade, fazendo com que as curvas de nível de sua forma bilinear sejam esferas, e seus auto-valores sejam todos iguais).

### 1.2.1 Gradientes conjugados

A ideia do método de gradientes conjugados é buscar pela solução iterativamente por $n$ direções de busca $\{d_0, \ldots, d_n\}$, de modo que se possa garantir que $x_i$ seja a melhor solução para o problema em $\mathcal{D}_i = span(\{d_0, \ldots, d_i\})$.

### 1.2.2 Conjugação de Gram-Schmidt

Em particular, fazemos as direções de busca $d_i$ A-ortogonais (ou *conjugados*) entre si, ie. $(d_i, d_j)_A = 0$ se $i \neq j$. Podemos pensar na A-ortogonalidade como uma ortogonalidade usual sob uma mudança de coordenadas que transforma os elipsóides das curvas de nível em esferas, que é do que se deriva boa parte das propriedades que obtemos de CG.

Para obter essas direções, podemos prosseguir pelo processo de Gram-Schmidt, mas tomando o produto interno como sendo o A-produto interno $(\cdot, \cdot)_A$, descrito brevemente a seguir:

Tome um conjunto de $n$ vetores linearmente independentes $\{u_i\}$. Para obter $d_i$ bastar tomar os $u_i$, em ordem, e subtrair os componentes que não sejam A-ortogonais com os $d_i$ anteriores. Esse é um algoritmo incremental:

$$\begin{aligned} d_0 &= u_0 \\ \text{para i} > 0 \quad d_i &= u_i - \sum_{k=0}^{i-1} \beta_{ik} d_k \\ \beta_{ij} &= \frac{(u_i, d_j)_A}{(d_j, d_j)_A} \end{aligned}$$

Em CG, faz-se $u_i = r_i$. Como $r_{i+1} = r_i - \alpha_i A d_i$, temos que cada $r_i$ é uma combinação linear do resíduo anterior e $Ad_{i-1}$. Disso, obtemos então que $\mathcal{D}_i$, o "espaço de busca"até o passo i, é formado pela união de $\mathcal{D}_i$ e $A\mathcal{D}_i$, do que segue que

$$\mathcal{D}_i = span\{d_0, Ad_0, A^2 d_0, \ldots, A^{i-1} d_0\} = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{i-1} r_0\}.$$

Esses espaços são chamados espaços de Krylov (mais em geral, $Krylov_r(A, b) := \{A^j b | j < r\}$).

Disso tiramos a propriedade de que, como $r_{i+1}$ é ortogonal a $\mathcal{D}_{i+1}$ (o que segue do fato de o resíduo em qualquer ponto ser ortogonal á superfície elipsoidal naquele ponto, e de o hiperplano $x_0 + \mathcal{D}_i$ também o ser), $r_{i+1}$ é A-ortogonal a $\mathcal{D}_i \subset D_{i+1}$. Assim, $r_{i+1}$ é A-ortogonal a todas as direções de busca anteriores, exceto $d_i$, o que facilita grandemente o processo de Gram-Schmidt descrito acima.

Mais especificamente, obtemos que

$$(r_i, d_j)_A = \begin{cases} \frac{1}{\alpha_i}(r_i, r_i), & i = j \\ -\frac{1}{\alpha_{i-1}}(r_i, r_i), & i = j + 1 \\ 0, & \text{caso contrário} \end{cases}$$

o que simplifica grandemente a expressão de $\beta_{ij}$, levando a complexidade (tanto em tempo quanto em espaço) de CG de $O(n^2)$ para $O(m)$, onde m é a quantidade de elementos não nulos de $A$. Mais especificamente, obtemos

$$\beta_i := \beta_{i,i-1} = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

### 1.2.3 Precondicionador

Dada uma matriz $A$ cujos maior e menor autovalores são, respectivamente, $\lambda_1$ e $\lambda_n$, definimos seu *número de condição*, ou simplesmente seu *condicionante* como $\kappa(A) := \frac{\lambda_1}{\lambda_n}$. O condicionante de uma matriz tem um papel importante na velocidade de convergência de diversos métodos e, em particular, para CG. Em geral valores grandes para $\kappa$ são ruins, fazendo com que o método possa levar a uma convergência inaceitavelmente lenta.

Para lidar com esse problema, uma alternativa é a pre-multiplicação do sistema por uma matriz $M$ positiva-definida que aproxima $A$, e que seja de fácil inversão. Então, pode-se resolver $Ax = b$ de forma indireta, fazendo

$$M^{-1}Ax = M^{-1}b$$

Se $\kappa(M^{-1}A) \ll \kappa(A)$, o problema acima se torna muito mais fácil do que o problema original.

O uso de precondicionadores apresenta, contudo, alguns problemas de ordem prática e teórica. Um problema é que, mesmo $A$ e $M$ sendo simétricas e definidas, $M^{-1}A$ pode não ser. Isso pode ser contornado, usando a decomposição de Cholesky, obtendo $E$ tal que $EE' = M$, e $E^{-1}AE^{-1\prime}$ será uma simétrica e positiva-definida.

Qual precondicionador usar é um problema de ordem prática, existindo algumas opções comuns, a depender do problema em mãos. Uma opção simples, mas de resultados mediocres, é o precondicionador de Jacobi, em que $M$ é uma matriz diagonal (realmente, muito fácil de inverter). Outra opção é precondicionamento de Cholesky incompleto, em que é feita uma decomposição de Cholesky incompleta da matriz A

Mas vale dizer que é, em geral, entendido que CG é sempre feito com o uso de algum precondicionador, ao menos para problemas de grandes, o que motiva a não-omissão desse tema neste documento introdutório.

## 1.3 Algoritmo

O método de gradientes conjugados pode então ser resumido no seguinte algoritmo (em que foram feitas algumas manipulações algébricas para comportar o condicionador $M$, sem a necessidade de computar $E$).

Dados $A$, $b$, um valor inicial para $x$ uma quantidade máxima de iterações $i_{max}$ e tolerância de erro $\epsilon < 1$ e um precondicionador $M$,

$$
\begin{aligned}
&i \leftarrow 0 \\
&r \leftarrow b - Ax \\
&d \leftarrow M^{-1}r \\
&\delta_{novo} \leftarrow (r, d) \\
&\delta_0 \leftarrow \delta_{novo} \\
&\text{Enquanto} \quad i < i_{max} \quad e \quad \delta_{novo} > \epsilon^2 \delta_0 \quad \text{faça} \\
&\quad q \leftarrow Ad \\
&\quad \alpha \leftarrow \frac{\delta_{novo}}{(d, q)} \\
&\quad x \leftarrow x + \alpha d \\
&\quad r \leftarrow r - \alpha q \\
&\quad s \leftarrow M^{-1}r \\
&\quad \delta_{velho} \leftarrow \delta_{novo} \\
&\quad \delta_{novo} \leftarrow (r, s) \\
&\quad \beta \leftarrow \delta_{novo}/\delta_{velho} \\
&\quad d \leftarrow s + \beta d \\
&\quad i \leftarrow i + 1
\end{aligned}
$$

Apesar de, a princípio, o método convergir em $n$ iterações ($n$ sendo a dimensão do problema), na prática isso pode não ocorrer devido a instabilidade numérica - mas, mesmo deixando instabilidade numérica de lado, para muitos problemas rodar $n$ iterações é computacionalmente impraticável. Além disso, em vários casos CG chega muito próximo à solução uma quantidade $k \ll n$ de iterações.

Por esses motivos são necessários os parâmetros $imax$ e $\epsilon$.

### 1.3.1 Implementação

Para fins de exemplo, veja como esse algoritmo pode ser implementado em Julia

[1]:
```
using LinearAlgebra # para podermos usar I, a identidade, por conveniência e␣
↪sem perda de eficiência computacional
function CG(A, b, x0, imax = size(A)[1], Minv = I,   = 1e-5)
    i  = 0
    x  = x0
    r  = b - A*x
```

```
    d  = Minv*r
    n = r'd
    0 = n
    while i < imax && n > ^2* 0
        q  = A*d
           = n/(d'q)
        x  = x + *d
        if (i + 1) ÷ 50 == 0  # por questão de estabilidade numérica,␣
↪recalculamos r a cada 50 iterações
            r = b - A*x
        else
            r  = r - *q
        end
        s  = Minv*r
        v = n       #  velho
        n = r's      #  novo
          = n/ v
        d  = s + *d
        i += 1
    end
    return x
end
```

[1]:
```
CG (generic function with 4 methods)
```

**Teste simples**

[2]:
```
A  = [3 2; 2 6]

b  = [2; -8]
x0 = [14; -20]

x = CG(A, b, x0)
```

[2]:
```
2-element Array{Float64,1}:
  2.0
 -2.000000000000001
```

Vemos que, ao menos para esse problema pequeno (para *sanity check*), o método funciona bem, apesar de haver a adição de um valor espúrio da ordem de $10^{-15}$:

[3]:
```
A\b
```

[3]:
```
2-element Array{Float64,1}:
  2.0
 -2.0
```

**CG não linear (NCG)** CG pode ser usado para encontrar o mínimo de quaisquer funções contínuas $f$, com algumas mudanças:

1. a fórmula para o resíduo muda;
2. é mais difícil calcular $\alpha$ - é preciso fazer, por exemplo, uma busca linear;
3. existem diferentes escolhas possíveis para $\beta$.

Não entraremos em muitos detalhes aqui, mas vale fazer uma comparação com o popular BFGS. BFGS é um método quase-Newton bem conhecido, cuja principal característica que afeta sua performance (em particular, em armazenamento) é a utilização de uma aproximação da Hessiana de $f$. Para problemas grandes, essa o armazenamento dessa aproximação pode significar um grande custo computacional, tornando sua utilização inviável, e o NCG uma alternativa válida.

Se a memória não for um problema, no entanto, BFGS, exceto para casos específicos, tende a se tornar uma melhor opção (na média, uma iteração de BFGS equivale a $n$ de NCG em questão de convergência, de modo que mesmo uma iteração de NCG podendo ser mais barata do que uma de BFGS, no geral essa diferença não compensa).

A variante L-BFGS (lê-se "*limited memory BFGS*") do BFGS é uma aproximação do mesmo que apresenta menor consumo de memória e (assintoticamente) menor tempo computacional por iteração, apesar de convergência mais lenta.

Vale lembrar que essas são considerações gerais, no entanto, sendo que cada problema prático pode exigir uma análise mais detalhada, apontando qual método seria mais efetivo.

## 1.4 Exemplo de Aplicação: Resolvendo o problema de Poisson em elementos finitos

O problema de Poisson é o de encontrar $u$ tal que

$$\begin{aligned} \nabla^2 u(x) &= -f, & x \in \Omega \\ u(x) &= u_D(x), & x \in \partial\Omega. \end{aligned}$$

Onde $u = u(x)$ é a função desconhecida, $f = f(x)$ é uma função, $\nabla^2$ é o operador de Laplace, $\Omega$ é o domínio espacial e $\partial\Omega$ é sua borda.

Apesar de simples, esse problema é muito importante, por exemplo, para aplicações em física e engenharia.

Para prosseguir pelo método de elementos finitos, precisamos reescrever o problema em sua forma variacional. Para tanto, primeiro multiplica-se a equação por uma função $v$, dita *função de teste*, e integramos no domínio

$$\int_\Omega (\nabla^2 u) v dx = \int_\Omega f v dx$$

então, aplicando integração por partes e insistindo que $v$ deve sumir na borda, obtém-se o sistema

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx \quad \forall v \in V$$

ou, de forma mais compacta (em que se colocam os problemas para serem resolvidos por métodos elementos finitos no geral)

$$a(u, v) = l(v)$$

onde a forma bilinear $a$ é $\int_\Omega \nabla u \cdot \nabla v dx$ e a forma linear $v$ é $\int_\Omega f v dx$, para $vinV$, onde $V$ é chamado "espaço de teste"(costuma-se insistir que $V$ seja um espaço Sobolev apropriado, mas por questões de brevidade vamos omitir aqui esse tipo de detalhe técnico).

A solução $u$ da EDP subjacente deve estar em um espaço de funções com derivadas contínuas, mas o espaço de Sobolev posto pela formulação variacional permite variadas descontínuas, o que tem grandes consequências práticas, como permitir a construção de uma solução a partir da "colagem"de funções polinomiais por partes.

Para resolver o problema, introduzimos uma discretização e buscaremos uma aproximação poligonal $u_N$ para $u$. O espaço $\Omega$ é discretizado em uma malha conforme (assumiremos aqui a discretização em um complexo simplicial, mas outras discretizações são possíveis). Assim, fazemos $V_N = \{v \in C^0 : v|_{s_i}$é linear e $v(\partial\Omega) = 0\}$, sendo $s_i$ os simplexos da discretização. $V_N$ tem dimensão finita, sendo $v \in V_N$ unicamente determinado por seus valores nos pontos da malha.

Dada uma base $\{\phi_i\}$ para $V_N$ (por exemplo, a base de Lagrange usual), temos $u_N = \sum_1^N U_j \phi_j$. Como todo $v \in V_N$ é combinação linear dos $\{\phi_j\}$, vemos que a formulação variacional é equivalente a

$$\int_\Omega u'_N \phi'_k dx = \int_\Omega f \phi_k dx \text{ para } k = 1, \ldots, N$$

do que segue que

$$\sum_1^N U_j \int_\Omega \phi'_j \phi'_k dx = \int_\Omega f \phi_k dx \text{ para } k = 1, \ldots, N$$

Assim, o problema se torn encontrar $U \in \mathbb{R}^N$ resolvendo o sistema

$$AU = F$$

onde $a_{ij} = \int_\Omega \phi'_j \phi'_i dx$ e $F_k = \int_\Omega f \phi_k dx$.

Tipicamente, a matriz $A$ é altamente esparsa quando se escolhe uma base apropriada (geralmente polinômios de Lagrange) e, quando se busca solução para uma malha de refinamento alto, o problema pode atingir dimensões muito altas (e especialmente para malhas em $\mathbb{R}^3$). Assim, para problemas grandes o suficiente (que são na verdade problemas de tamanho modesto para malhas em $\mathbb{R}^3$), é necessário o uso de métodos iterativos, entre os quais CG é entre os mais populares. CG é especialmente adequado para o problema de Poisson, que resulta em uma matriz simétrica e positiva-definida, e pacotes para métodos de elementos finitos populares, como o fenics oferecem a opção de usá-lo como método iterativo (com precondicionador apropriado).

## 1.5 Experimentos computacionais

A seguir são feitos experimentos computacionais quanto à adequação de CG para a resolução de sistemas esparsos, com ou sem condicionador.

Para efeitos dos experimentos, foi utilizada uma matriz de Wathen. Uma matriz de Wathen(Nx,Ny) é uma matriz de elementos finitos aleatória N por N (fazendo $N = 3NxNy + 2Nx + 2Ny + 1$), sendo a "matriz de consistência de massa"para um grade regular Nx por Ny de 8 elementos nodais em 2 dimensões espaciais. A matriz é simétrica positiva definida para qualquer valor (positivo) da densidade, que é escolhida aleatoriamente.

[4]:
```
using BenchmarkTools, MatrixDepot, IterativeSolvers, LinearAlgebra, SparseArrays

# Matriz de Wathen de dimensões 30401 x 30401
A = matrixdepot("wathen", 100)
```

```
include group.jl for user defined matrix generators
verify download of index files…
reading database
adding metadata…
adding svd data…
writing database
used remote sites are sparse.tamu.edu with MAT index and math.nist.gov with HTML
index
```

[4]:
```
30401×30401 SparseMatrixCSC{Float64,Int64} with 471601 stored entries:
  [1    ,     1]  =  6.2388
  [2    ,     1]  =  -6.2388
  [3    ,     1]  =  2.0796
  [202  ,     1]  =  -6.2388
  [203  ,     1]  =  -8.31839
  [303  ,     1]  =  2.0796
  [304  ,     1]  =  -8.31839
  [305  ,     1]  =  3.1194
  [1    ,     2]  =  -6.2388
  [2    ,     2]  =  33.2736
  [3    ,     2]  =  -6.2388
  [202  ,     2]  =  20.796

  [30199, 30400]  =  21.3736
  [30200, 30400]  =  21.3736
  [30399, 30400]  =  -6.41209
  [30400, 30400]  =  34.1978
  [30401, 30400]  =  -6.41209
  [30097, 30401]  =  3.20604
  [30098, 30401]  =  -8.54945
  [30099, 30401]  =  2.13736
  [30199, 30401]  =  -8.54945
```

```
  [30200, 30401]  =  -6.41209
  [30399, 30401]  =  2.13736
  [30400, 30401]  =  -6.41209
  [30401, 30401]  =  6.41209
```

[6]:
```
# Nível de esparsidade
count(!iszero, A) / length(A)
```

[6]:
```
0.0005102687577359558
```

[7]:
```
b = ones(size(A, 1))
# Resolve Ax=b by CG
xcg = cg(A, b);
@benchmark cg($A, $b)
```

[7]:
```
BenchmarkTools.Trial:
  memory estimate:  951.36 KiB
  allocs estimate:  16
  --------------
  minimum time:     280.710 ms (0.00% GC)
  median time:      430.303 ms (0.00% GC)
  mean time:        438.680 ms (0.00% GC)
  maximum time:     765.398 ms (0.00% GC)
  --------------
  samples:          12
  evals/sample:     1
```

### 1.5.1 Usando precondicionador de Cholesky

[8]:
```
using Preconditioners
@time p = CholeskyPreconditioner(A, 2)
```

```
  4.076178 seconds (2.43 M allocations: 151.649 MiB, 1.26% gc time)
```

[8]:
```
CholeskyPreconditioner{Float64,SparseMatrixCSC{Float64,Int64}}([7.77401676578252
85 0.0 … 0.0 0.0; 0.0 11.520622371156024 … 0.0 0.0; … ; 0.0 0.0 …
3.048358439214821 0.0; 0.0 0.0 … 0.0 5.7811090761367625], 2)
```

Resolve Ax=b com precondicionador

[9]:
```
xpcg = cg(A, b, Pl=p)
# same answer?
norm(xcg - xpcg)
```

[9]:
```
5.306315159734449e-7
```

**CG foi, neste exemplo > vezes 10 mais lento do que CG** O que é curioso, porque parece que em outros computadores o resultado é o inverso (CG com precondicionador (PCG) é > 10 vezes mais rápido).

[10]:
```
@benchmark cg($A, $b, Pl=$p)
```

[10]:
```
BenchmarkTools.Trial:
  memory estimate:  951.36 KiB
  allocs estimate:  16
  --------------
  minimum time:     4.958 s (0.00% GC)
  median time:      5.235 s (0.00% GC)
  mean time:        5.235 s (0.00% GC)
  maximum time:     5.513 s (0.00% GC)
  --------------
  samples:          2
  evals/sample:     1
```

## 1.6 Referências

Para escrever este documento, foram usadas, primariamente, as seguintes referências:


- An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, de Jonathan Richard Shewchuk, primariamente para o desenvolvimento teórico disponível aqui
- Notas de aula de Hua-Zhou, primariamente para a implementação computacional disponível aqui
- Documentação da função Wathen para a sua utilização disponível aqui (foi utilizada a documentação da função mas não a função indicada, que foi feita para MatLab)
- Galerkin Approximations and Finite Element Methods, de Ricardo G. Durán, para alguns detalhes quanto a métodos de elementos finitos disponível aqui
- Solving PDEs in Python - The FEniCS Tutorial Volume 1, de Hans Peter Langtangen e Anders Logg, para outros detalhes quanto a métodos de elementos finitos para o problema de Poisson, disponível aqui