Métodos de otimização vetorial e seus algoritmos no *MatLab*

Otimização - 2015.2

Universidade Federal do Rio de Janeiro

Felipe Matheus Fernandes Oliveira
felipemfo@poli.ufrj.br

Henrique Passos de Mattos
henriquemattos@poli.ufrj.br

Jean Américo Tomé
jamerico@poli.ufrj.br

Philipe Miranda de Moura
philipemoura@poli.ufrj.br

18 de março de 2016

Conteúdo

1 Introdução

Nos dias de hoje, várias indústrias, empresas, ciências exatas e da engenharia se deparam com diversos problemas que possuem infinitas soluções para suas variáveis. Como, nestes problemas, frequentemente queremos encontrar uma combinação de variáveis de maneira a ter o maior lucro, menor prejuízo, ou, simplesmente, achar um valor ótimo para um problema qualquer, as técnicas de otimização são e se tornam cada vez mais importantes. Entre elas, as de minimização (maximização) de resultados são as mais relevantes e as que serão estudadas neste trabalho.

A grande questão é que existem diversos tipos de situações problemáticas que precisam ser levadas em conta. Podemos ter processos otimizáveis com duas variáveis que dependam entre si. Para estes, utilizamos técnicas de minimização escalar, como já vimos anteriormente. Outros processos possuem várias variáveis envolvidas que podem ser relacionadas escolhendo-se uma delas (a que se quer minimizar (maximizar)) que dependa de uma combinação (linear ou não) das outras. Para estes, são necessárias outras técnicas de otimização, chamadas de métodos de minimização vetorial, alguns dos quais serão analisados aqui. Devido ao fato de muitas vezes ser difícil minimizarmos uma função analiticamente, analisaremos aqui métodos numéricos computacionais para realizar, com alguma tolerância, as otimizações necessárias. Estes métodos são: método do gradiente, método do gradiente conjugado, métodos de Newton, métodos de Newton modificados e métodos de quase Newton.

2 Métodos de otimização vetorial

Existem diversos métodos de otimização vetorial. Vamos explicar algumas características dos que serão analisados neste trabalho.

Primeiramente, todos são métodos de pontos convergentes, ou seja, começa-se com um ponto qualquer no $\mathbb{R}^n$ e, a cada iteração, aproxima-se este ponto do mínimo da função vetorial.

As iterações acontecem no sentido de fazer com que andemos uma distância α_k numa direção d^k com relação ao ponto anterior. Assim, $x^{k+1} = x^k + \alpha_k d^k$. A diferença entre os métodos é a maneira com a qual eles escolhem a direção e a quantidade de avanço para cada iteração.

A última característica dos métodos analisados é o critério de parada. Ele se baseia em duas ideias. A primeira é a proximidade que o gradiente de um ponto iterativo está de se anular. Quando o gradiente de um ponto estiver tão próximo de zero quanto a tolerância permita, podemos parar o método. A outra ideia básica é a proximidade de pontos da sequência iterativa. Quando um ponto estiver muito próximo do anterior, significa que estamos perto da solução e podemos interromper o método.

Explicadas algumas características comuns dos métodos, vamos à análise de suas diferenças e de suas motivações específicas.

2.1 Método do gradiente (ou descida máxima)

A ideia básica deste método é: a direção de cada iteração deve ser a de maior descida da função. Como sabemos que o gradiente de uma função em um ponto indica a direção de maior subida da função a partir daquele ponto, neste método, $d^k = - \bigtriangledown f(x^k)$.

Tendo a direção de avanço, d^k, para as iterações, precisamos saber como é calculada a quantidade de avanço, α_k, para este método. Como possuímos x^k e d^k, o próximo ponto iterativo fica em função do parâmetro desconhecido: $x^{k+1} = x^k + \alpha_k d^k = x(\alpha_k)$. Deste modo, o valor da função no próximo ponto é $f(x^k + \alpha_k d^k) = f(\alpha_k)$. Queremos encontrar o avanço α que causa o maior decréscimo na função f quando andamos na direção d^k a partir de x^k, logo, podemos minimizar, de maneira escalar (métodos que já conhecemos), a função $f(\alpha_k)$. Desta maneira, encontramos a direção e a quantidade do avanço a cada iteração.

O método do gradiente possui dois critérios de parada. O primeiro é a análise do gradiente em cada ponto iterativo. Se este for muito próximo de 0 (de acordo com a tolerância), então paramos o método. O segundo critério é baseado na proximidade entre dois pontos consecutivos. Se os valores de função de dois pontos consecutivos estiverem muito próximos (em uma análise absoluta unida com uma relativa, ou seja, duas tolerâncias diferentes), o método é interrompido.

Analisando mais alguns aspectos do método do gradiente, poderemos ver que, minimizando a função $f(\alpha_k)$ em cada iteração, fazemos com que duas direções de descida consecutivas sejam perpendiculares entre si. Desta forma, o método do gradiente segue uma trajetória de zig-zag com ângulos retos.

2.2 Método do gradiente conjugado

Como visto anteriormente, o método do gradiente segue uma trajetória de zig-zag com ângulos retos. Este fato pode fazer com que, até para algumas funções mais simples, o método convirja muito lentamente, ou mesmo, não convirja. Para melhorar isto, foi desenvolvido o método do gradiente conjugado.

Com o intuito de corrigir um dos grandes problemas do método de descida máxima (que é a perpendicularidade de descidas consecutivas), o MGC tenta suavizar as mudanças de direção baseado em um desvio com relação a direção anterior: $d^{k+1} = - \bigtriangledown f(x^{k+1}) + \beta_k d^k$. β_k seria o fator de suavização, ou fator de inércia, e é escolhido de maneira a fazer com que duas direções de descida consecutivas sejam conjugadas. Abaixo, estarão representados os dois βs utilizados em métodos do gradiente conjugado. Quando utiliza-se o primeiro, diz-se que estamos usando o MGC na forma de Polak-Rebière e, no caso do segundo, forma de Fletcher-Reeves.

$$\beta_{k_{PR}} = \frac{\|g^{k+1}\| - \{g^{k+1}\}^T g^k}{\|g^k\|}$$

$$\beta_{k_{FR}} = \frac{\|g^{k+1}\|}{\|g^k\|}$$

Como a primeira direção de descida no MGC, d^0, obviamente, não tem anterior, ela deve ser igual a direção de máxima descida, assim como fazemos no método da descida máxima.

O MGC foi pensado para funções quadráticas. Para este tipo de funções, a fórmula de β_k pode ser usada tranquilamente, assim como uma fórmula definida para α_k, que vem da minimização analítica de $f(\alpha_k)$, e também o método converge depois de feitas n iterações, onde n é a ordem do problema. Para estendermos o método a qualquer tipo de função, devem ser assumidos critérios de parada, a fórmula para β_k pode continuar sendo usada, mas α_k deve ser determinado por métodos de minimização escalar. Além disso, a cada n iterações, a direção de descida d^n é igual à direção de descida máxima em x^n.

2.3 Métodos de Newton

Como uma função vetorial quadrática (assim como uma escalar) é muito fácil de ser minimizada (existe uma fórmula fácil para encontrarmos o mínimo delas), o método de Newton se baseia em encontrar aproximações quadráticas de Taylor para a função objetivo que estamos tratando em torno do ponto iterativo x^k e encontrar o mínimo desta quadrática, tonando-o o seguinte ponto, x^{k+1}. Aplicando esta ideia, veremos que a diferença entre dois pontos iterativos consecutivos deve seguir a seguinte fórmula:

$$x^{k+1} - x^k = -[\bigtriangledown^2 f(x^k)]^{-1} \bigtriangledown f(x^k) = d^k$$

Onde d^k é a direção de avanço de um ponto a outro

Por considerarem as Hessianas e os gradientes nos cálculos das direções de avanço (diferentemente dos outros que vimos até agora, que só consideraram os gradientes), os métodos de Newton são chamados de métodos de segunda ordem.

Apesar da rapidez destes métodos (por utilizarem as Hessianas), eles contêm um grande problema: a convergência. A base do método parte do pressuposto que a quadrática aproximada no ponto iterativo terá Hessiana positiva definida e, por isso, a direção de avanço fará com que o próximo ponto fique mais perto do mínimo da função, mas nem sempre é assim. Caso a Hessiana em um ponto iterativo seja negativa, o método não convergirá. Caso ela seja positiva, mas variar com relação às variáveis da função objetivo, o método também não converge.

2.4 Métodos de Newton modificados

Assim como o gradiente conjugado vem para corrigir problemas de convergência do método da descida máxima, os métodos de Newton modificados vêm com o mesmo propósito em relação aos métodos puros.

Uma das ideias destes novos métodos para corrigir o problema de divergência de funções com Hessianas variáveis é utilizar a mesma direção dos anteriores ($d^k = -[\bigtriangledown^2 f(x^k)]^{-1} \bigtriangledown f(x^k)$), mas com uma quantidade de avanço (α_k) diferente de 1 e igual ao valor que minimiza $f(x^k + \alpha_k d^k) = f(\alpha_k)$ (minimização escalar).

Para a correção do problema de funções com Hessianas negativas no ponto iterativo tratado, usa-se uma pequena manipulação matricial:

$$F^k = G^k + \gamma I_n$$

Onde G^k é a Hessiana da função objetivo no ponto x^k, I_n é a matriz identidade de ordem n (ordem do problema), γ é um número real que faça com que F^k tenha autovalores positivos e F^k é a substituição para a Hessiana da função objetivo na fórmula da direção de avanço (como F^k tem autovalores positivos, ou seja, matriz positiva definida, podemos garantir que d^k será uma direção de descida). Na escolha de γ, também devemos considerar que os autovalores de F^k não devem estar muito próximos de 0.

Como os cálculos da Hessiana e dos autovalores de uma matriz podem demandar muito tempo de computação, é aconselhável o uso deste método apenas quando eles forem fáceis de serem calculados.

2.5 Métodos de quase Newton

Assim como todos os outros métodos anteriores, estes vêm com o intuito de melhorar algo, e a sua finalidade é aprimorar o cálculo da direção de avanço nos métodos de Newton modificados para não se ter a necessidade do cálculo da Hessiana nem de seus autovalores.

Como $d^k = -[F^k]^{-1} g^k$ e F^k depende da Hessiana de f, podemos procurar maneiras de determinarmos $H^k = [F^k]^{-1}$ sem termos que calcular a derivada segunda de f. Uma solução para um desenvolvimento deste problema foi apresentada por Davidon, Fletcher e Powell (DFP) e diz que:

$$H^{k+1} = H^k - \frac{H^k \gamma^k [\gamma^k]^T H^k}{[\gamma^k]^T H^k \gamma^k} + \frac{\delta^k [\delta^k]^T}{[\delta^k]^T \gamma}$$

Onde $\gamma^k = g^{k+1} - g^k$ e $\delta^k = x^{k+1} - x^k$ e considera-se $H^0 = I$.

Uma característica importante desta fórmula é o fato de que se H^k for positiva definida, H^{k+1} também será. Baseado nisto, como $H^0 > 0$, $H^k > 0$ para qualquer k e, assim, todas as direções de avanço serão descidas na função e não haverá a precoupação de existirem direções positivas.

Outra solução do mesmo problema foi encontrada por Broyden, Fletcher, Goldfarb e Shanno (BFGS) e diz que:

$$H^{k+1} = H^k - \frac{\delta^k [\gamma^k]^T H^k + H^k \gamma^k [\delta^k]^T}{[\delta^k]^T \gamma^k} + (1 + \frac{[\gamma^k]^T H^k \gamma^k}{[\delta^k]^T \gamma^k}) \frac{\delta^k [\delta^k]^T}{[\delta^k]^T \gamma^k}$$

3 Algoritmos e interface no *MatLab*

Explicados os funcionamentos dos métodos mais relevantes de minimização vetorial, mostraremos aqui como eles foram implementados no software *MatLab* e ainda temos um extra: a explicação do funcionamento de uma interface gráfica realizada no trabalho para unir a utilização de todos os métodos escalares e vetoriais estudados até o momento.

3.1 Métodos

As maiores restrições para a utilização destes métodos são:

- As funções objetivo para os métodos do gradiente, gradiente conjugado e de Newton devem ser unimodais e com Hessiana positiva definida em todo o seu domínio. Caso contrário, os métodos citados podem divergir ou os do gradiente podem convergir para um mínimo ou máximo local.
- Qualquer função objetivo utilizada deve ser suave. Se não for, os métodos podem acusar algum erro durante seus funcionamentos.

Além destas restrições, vale ressaltar, com relação aos códigos, que o gradiente conjugado foi implementado para a forma de Polak-Rebière (PR) ou Fletcher-Reeves (FR) e o quase Newton, para DFP ou BFGS.

Feitas as devidas análises e observações, seguem abaixo, finalmente, os códigos implementados.

3.1.1 Descida máxima

```
function [ ponto_min, valor_min, numI, historico ] = Gradiente( expr, x0, toln, tolg, lim_it )
%x0 -> ponto inicial inserir em array com ponto e virgula Ex: [x;y]
%toln -> tolerancia numerica (distancia entre pontos)
%tolg -> tolerancia do gradiente (modulo do gradiente)
xk = x0; % ponto inicial
numI = 0;
funcao = sym(expr); % funcao ja simbolica
pontos = symvar(funcao); % vetor com as variaveis ja simbolicas da funcao
grad = gradient(funcao); % gradiente com as variaveis simbolicas
%valorfuncao = subs(funcao,pontos,transpose(xk)); % definindo o valor da funcao
gk = subs(grad,pontos,transpose(xk)); % gradiente com os pontos substituidos
historico = [xk; double(subs(funcao, pontos,transpose(xk)))];
if (norm(gk)) <= tolg
    ponto_min = xk; %xk = x0
    valor_min = subs(funcao,pontos,transpose(xk)); %xk = x0
    return
end
historico = [historico [xk; double(subs(funcao, pontos,transpose(xk)))]];

while (norm(gk)) > tolg;


    if numI == lim_it
        break;
    end
    numI = numI + 1
    dk = -subs(grad,pontos,transpose(xk))/norm(subs(grad,pontos,transpose(xk)));
%vetor unitario da direcao de descida
    syms a;
    pontosT = transpose(pontos); %deixando os vetores nas mesmas dimensoes
    funcao_de_a = subs(funcao,pontosT,xk+a*dk); %funcao de a simbolica
    funstring = char(funcao_de_a); % funcao de a em forma de string
    ak = double(aurea4(funstring, 0.001, [-50,50])); %calculando o minimo
    xk1 = double(xk + ak*dk);

    ponto_min = xk1;
```

```
    valor_min = double(subs(funcao,pontos,transpose(xk1)));
    gk1 = double(subs(grad,pontos,transpose(xk1)));

    if (norm(xk1-xk)) <= toln
        break;
    end

    xk = xk1;
    gk = gk1;

    historico = [historico [xk; double(subs(funcao, pontos,transpose(xk)))]];
end
disp(historico);

disp(numI);

disp(valor_min);
```

3.1.2 Gradiente conjugado

```
function [Y1, Y2, Y3, Y4] = Gradconj_henrique (X1, X2, X3, X4, X5)
f = sym (X1);
variables=symvar(f);
dimensions = size(symvar(f));
ea = 10^-08;
er = 10^-08;
f0 = double(subs (f, variables, X2));
point = zeros([1, dimensions(2)]);
g0 = zeros([1, dimensions(2)]);
for a = 1:dimensions(2)
    b = diff(f, variables(a));
    g0(a) = double(subs (b, variables, X2));
    point(a) = double(X2(a));
end
fs = [point, f0];
if norm(g0)<X3
    Y1 = X2;
    Y2 = f0;
    Y3 = 0;
    Y4 = fs;
    return
end
d0 = -g0;
syms alpa;
f_alpha = subs(f, variables, X2 + alpa*d0);
f_alpha_str = char (f_alpha);
Yaurea = aurea4 (f_alpha_str, 10^-08, [-100, 100], 0, 50);
alpha_r = Yaurea(1);
x1 = X2 + alpha_r*d0;
f1 = double(subs (f, variables, x1));
g1 = zeros([1, dimensions(2)]);
for a = 1:dimensions(2)
    b = diff (f, variables(a));
    g1(a) = double(subs (b, variables, x1));
    point(a) = double(x1(a));
end
fs = [fs, point, f1];
k=1;
while norm (g1)>X3 && abs(f1-f0)>ea + er*abs(f0) && k~=X4
    if rem (k, dimensions(2)) == 0
        d1 = -g1;
    else
        if X5 == 0
            beta = (norm(g1) - g1*transpose(g0))/(norm(g0));
            d1 = -g1 + beta*d0;
        else
            beta = (norm(g1))/(norm(g0));
```

```
            d1 = -g1 + beta*d0;
        end
    end
    syms alpa;
    f_alpha = subs(f, variables, x1 + alpa*d1);
    f_alpha_str = char (f_alpha);
    Yaurea = aurea4 (f_alpha_str, 10^-08, [-100, 100], 0, 50);
    alpha_r = Yaurea(1);
    x2 = x1 + alpha_r*d1;
    f2 = double(subs (f, variables, x2));
    g2 = zeros([1, dimensions(2)]);
    for a = 1:dimensions(2)
        b = diff (f, variables(a));
        g2(a) = double(subs (b, variables, x2));
        point(a) = double(x2(a));
    end
    fs = [fs, point, f2];
    k = k + 1;
    x1 = x2;
    f0 = f1;
    f1 = f2;
    g0 = g1;
    g1 = g2;
    d0 = d1;
end
Y1 = x1;
Y2 = f1;
Y3 = k;
Y4 = fs;
end
```

3.1.3 Newton

```
function [ pontoMinimo, f_minimo, iteracoes, historico ] = NewtonPH( funcao, pontoInicial, tol, maxIt, enab
tic;
ponto=transpose(pontoInicial);
numI=0;
xyz=symvar(funcao);
dim=size(xyz);
dim=dim(1);
disp(dim);
funcao = sym(funcao);
grad=gradient(funcao);
hes = jacobian(gradient(funcao));
gradPonto = double(subs(grad, xyz, ponto));
historico=[ponto; subs(funcao, xyz, ponto)];
    while abs(gradPonto)>tol
        if numI>maxIt
            disp('Exceeded number of iterations');
            break
        end
        numI = numI + 1;
        hesPonto=double(subs(hes, xyz, ponto));
        ponto=double(ponto-(hesPonto^-1)*gradPonto);
        gradPonto=double(subs(grad,xyz, ponto));
        historico=[historico [ponto; subs(funcao, xyz, ponto)]];
    end
    pontoMinimo = ponto;
    f_minimo = subs(funcao, xyz, ponto);
    iteracoes = numI;
    time = toc;
    disp('Elapsed time');
    disp(time);
    hold off
end
```

3.1.4 Newton modificado

```
function [ pontoMinimo, f_minimo, iteracoes, historico ] = NewtonModPH( funcao, pontoInicial, tol, maxIt, e
tic;
ponto=transpose(pontoInicial);
numI=0;
xyz=symvar(funcao);
dim=size(xyz);
dim=dim(1);
disp(dim);
funcao = sym(funcao);
grad=gradient(funcao);
hes = jacobian(gradient(funcao));
gradPonto = subs(grad, xyz, ponto);
historico=[ponto; subs(funcao, xyz, ponto)];
    while abs(gradPonto)>tol
        if numI>maxIt
            disp('Exceeded number of iterations');
            break
        end
        numI = numI + 1;
        hesPonto = double(subs(hes, xyz, ponto));
        gama = double(abs(min(eig(hesPonto)))+10);
        hesPonto = double(hesPonto+eye(size(hesPonto)));
        dk = double(-(hesPonto^-1)*gradPonto);
        alpha = sym('alpha');
        newFunction = subs(funcao, xyz, ponto+alpha*dk);
        aurea4(newFunction, 0.1, [-10 10], 0, 100);
        ponto = double(ponto-(hesPonto^-1)*gradPonto);
        gradPonto = double(subs(grad,xyz, ponto));
        if enable==1
            historico=[historico [ponto; subs(funcao, xyz, ponto)]];
        end
    end
    pontoMinimo = ponto;
    f_minimo = subs(funcao, xyz, ponto);
    iteracoes = numI;
    time = toc;
    disp('numI');
    disp(numI);
    disp('Elapsed time');
    disp(time);
end
```

3.1.5 Quase Newton

```
function [ ponto_min, valor_min, numI, historico] = quase_newton( expr, x0, toln, tolg, lim_it, tipo)
    %x0 -> ponto inicial iniciar como [x,y]
    %toln -> tolerancia numerica (distancia entre pontos)
    %tolg -> tolerancia do gradiente (modulo do gradiente)
    %tipo -> diz se e DFP ou BFGS (para ser BFGS, digitar 2)

    switch nargin
        case 5
            tipo = 1;    end

    xk = x0; % ponto inicial
    funcao = sym(expr); % funcao ja simbolica
    pontos = symvar(funcao); % vetor com as variaveis ja simbolicas da funcao
    %valorfuncao = subs(funcao,pontos,xk); % definindo o valor da funcao
    numI = 0;
    grad = gradient(funcao); % gradiente com as variaveis simbolicas
    gk = subs(grad,pontos,xk); % gradiente com os pontos substituidos
    tam = size(x0); %retorna vetor de dimensao 2 com numero de linha e coluna
    dim = tam(2); %seleciona o numero de dimensoes da expressao
```

```
Hk = eye(dim); % condicao inicial de Hessiana numerica (ser identidade)
historico = [transpose(xk); double(subs(funcao, pontos, xk))];

if (norm(gk)) <= tolg
    ponto_min = xk; %xk = x0
    valor_min = subs(funcao,pontos,xk); %xk = x0
    return
end

historico = [historico [transpose(xk); double(subs(funcao, pontos, xk))]];

while (norm(subs(grad,pontos,xk))) > tolg;

        numI = numI + 1
        if numI == lim_it
            break;
        end

        %definindo variaveis que serao calculadas no loop
        gk = subs(grad,pontos,xk);
        Fk = inv(Hk);

        %otmizacao escalar
        dk = -mtimes(Hk,gk);
        syms a;
        funcao_de_a = subs(funcao,pontos,xk+a*transpose(dk));
        funstring = char(funcao_de_a);
        ak = double(aurea4(funstring, 0.01, [-50,50]));

        %expressao final
        xk1 = double(xk - ak*transpose(mtimes(inv(Fk),gk)));

        %definicao das saidas
        ponto_min = xk1;
        valor_min = double(subs(funcao,pontos,xk1));
        gk1 = double(subs(grad,pontos,xk1));

        %calculo do novo Hk por DFP
        %definicao das variaveis
        gama = double(gk1 - gk);
        delta = double(xk1 - xk);
        deltaT = transpose(delta); % Delta esta escrito em forma de pontos, enquanto gama em vetor, tra
        %conferir a tolerancia numerica (toln)
        if norm(xk1-xk) < toln
            break;
        end

        if (tipo == 2) %caso BFGS
            %termo0 = Hk
            %termo1 = (delta*gamaT*H+H*gama*deltaT)/deltaT*gama
            %e assim sucessivamente pelos proximos termos
            termo1 = double((mtimes(deltaT,transpose(gama))))*Hk+Hk*double(mtimes(gama,transpose(deltaT
            termo2 = double(mtimes((mtimes(transpose(gama),Hk)),gama))/double(mtimes(transpose(deltaT),
            termo3 = double(mtimes(deltaT,transpose(deltaT)))/double(mtimes(transpose(deltaT),gama));

            %escrevendo a expressao, detalhe termos 2 e 3 sao escalares
            %finalmente:
            Hk1 = Hk - termo1 +(1 + termo2)*termo3;

        else %caso DFP
            fracao1 = mtimes(mtimes(Hk,gama),mtimes(transpose(gama),Hk))/double(mtimes(mtimes(transpose
            fracao2 = double(mtimes(deltaT,transpose(deltaT)))/double(mtimes(transpose(deltaT),gama));
            Hk1 = Hk - fracao1 + fracao2;
        end

        %atribuindo por fim os novos valores de iteracao
        Hk = Hk1;
        xk = xk1;
```

```
            historico = [historico [transpose(xk); double(subs(funcao, pontos, xk))]];


    end

            historico = transpose(historico)
            disp(historico);

            disp(numI);

            disp(valor_min);
```

3.2 Interface gráfica

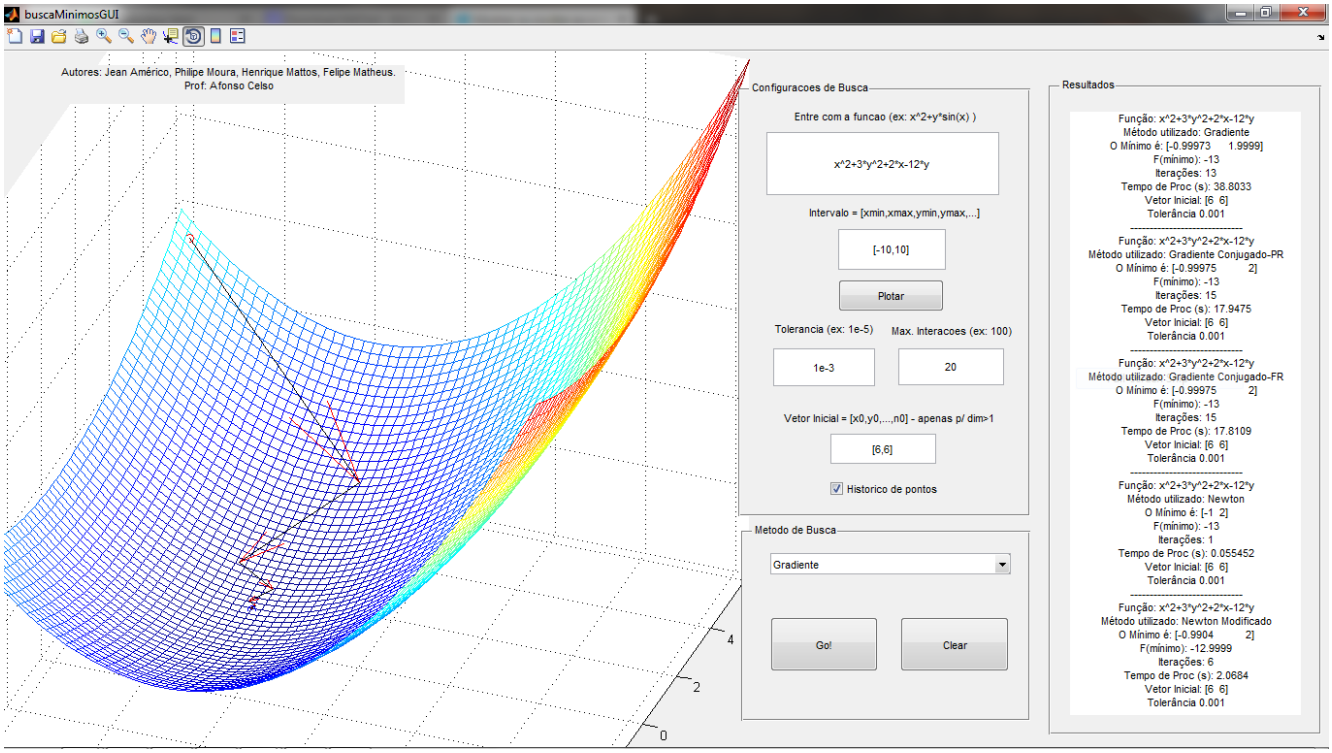

Figura 1: GUI desenvolvida pelo grupo

Como já foi falado, adaptamos a GUI desenvolvida no primeiro trabalho para uma forma mais genérica, abrangendo tanto otimização em uma dimensão quanto otimização em um número genérico de dimensões.

Para tanto, usamos um mecanismo dinâmico que, verificando a dimensão da função de entrada, altera as demais caixas de seleção. Dessa forma, o usuário final pode escrever a função desejada em qualquer dimensão sem se preocupar com as variações entre os métodos de entrada.

- **Função:** Para o caso de uma dimensão: uma expressão qualquer relacionando a variável 'x'. Para o $\mathbb{R}^n$: qualquer expressão relacionando variáveis alfabéticas. É permitido utilizar funções pré-definidas do *MatLab*, como sin(x), cos(x), exp(x), etc. Exemplos: `x^2, x^2+4*x^3+4, x^3*cos(-5*x) x^2+y^2,z^3*sin(w), x^2*z^2*exp(-3*t)`
- **Intervalo:** Vetor intervalo da forma [xmin,xmax,ymin,ymax,...,zmin,zmax]. Este intervalo define os limites do *plot*. Para o caso de não haver argumentos o suficiente, o *plot* será padrão de -2π a 2π para cada variável de funções multivariáveis ou [-10, 10] para o caso univariável. Exemplo: [-4,4] para `x^2` ou [-10,10,-4,4] para `x^2+y^2`.

- **Plotar:** Plota a função dentro do intervalo desejado. Esse mecanismo ajuda na escolha de um intervalo adequado antes de iniciar a busca.
- **Tolerância:** Valor numérico ou em forma de notação científica com a tolerância máxima. Exemplo: 0.0001 ou 1e-3.
- **Máximo de Iterações:** Número máximo de iterações permitidas até o programa parar de procurar. Exemplo: 20 ou 100.
- **Vetor Inicial:** Vetor com as coordenadas do ponto inicial da busca. Exemplo: [-1,2] para `x^2+y^2`. A dimensão do vetor ponto inicial deve ser igual à quantidade de variáveis da função de entrada. Para `y^3*sin(x)+z^2`, que relaciona três variáveis (x,y e z), é necessário um vetor com três pontos, tal como [-1,-10,10].
- **Histórico de Pontos:** Se marcado, mostra no gráfico o conjunto de pontos encontrados em cada iteração. Para funções multivariáveis, o histórico representa vetores ligando o ponto anterior ao próximo ponto.
- **Método de Busca:** Selecionar entre os métodos possíveis para a otimização escalar ou multivariável:
  - Fibonacci, Seção Áurea ou Interpolação Polinomial.
  - Gradiente, Gradiente Conjugado-PR, Gradiente Conjugado-FR, Newton, Newton Modificado, Quase Newton-BFGS, Quase Newton-DFP.
- **Botão Clear:** Limpa a caixa de textos 'Resultados' à direita.
- **Botão Go!:** Inicia a busca.