Projeto 3 de Cálculo 2 Honors
Arquivo principal

Prof. Lucas Pedroso

2º semestre de 2024

Última atualização: 08/11/24 (confira no final do arquivo a lista com as modificações desde a publicação)

Este arquivo trata das regras gerais para o Projeto 3.

1 Funcionamento do projeto

O projeto funcionará no esquema de jogos. Para zerar o jogo, o aluno deve cumprir o seguinte:

- implementar o método do gradiente para minimização de funções no $\mathbb{R}^2$, com derivadas fornecidas pelo usuário;
- criar 3 exemplos de funções a serem minimizadas;
- colocar um output visual das curvas de nível da função junto com o percurso do método.

Zerar o jogo sem erros vale 100 pontos no projeto (lembrando que são 3 projetos, essa nota será somada com as dos outros dois projetos e dividida por 3). Além de terminar o jogo, haverá os achievements. Serão seis achievements (quantidade pode ser modificada nos próximos dias). Cada achievement completo e correto valerá 1 ponto na média. Ou seja, se alguém fizer os 6 e tiver, digamos, 50 de média, ficará com 56. Os achievements não são obrigatórios e recomendo que só faça quem estiver a fim de aprender (não quero que ninguém faça forçado). Os achievements são os seguintes:

- implementar tudo (exceto output gráfico obviamente) para problemas gerais do $\mathbb{R}^n$ em vez de se limitar ao $\mathbb{R}^2$ (nível de dificuldade: fácil)
- implementar o método de Newton além do gradiente. Este método é mais caro computacionalmente que o do gradiente mas leva bem menos iterações pra convergir. Será ensinado em sala, exceto se não der tempo, nesse caso marcarei um horário extra com os interessados (nível de dificuldade: médio)

- dar a possibilidade de os gradientes e Hessianas serem calculados por diferenças finitas centradas (nível de dificuldade: médio)
- para os métodos que não sejam o do gradiente, implementar salvaguardas para garantir que a direção seja de descida (nível de dificuldade: fácil)
- fazer busca linear em vez de usar passo constante (nível de dificuldade: fácil)
- implementar o método BFGS, que costuma ser mais rápido que Newton e gradiente. Os interessados terão que fazer uma aula extra (rápida) comigo pra eu falar sobre o método (nível de dificuldade: honors)

1.1 Método do gradiente

Considere o problema de minimizar $f : \mathbb{R}^n \to \mathbb{R}$. O pseudocódigo abaixo explica o funcionamento do método do gradiente, também conhecido como gradiente descendente.

Algorithm 1 Método do gradiente

Data: $f : \mathbb{R}^n \to \mathbb{R}$, $x^0 \in \mathbb{R}^n$, $\alpha, \epsilon > 0$, $K \in \mathbb{N}$.
Initialization: $x \leftarrow x^0$ e $k \leftarrow 0$.
  while $\|\nabla f(x)\| > \epsilon$ e $k < K$ **do**
    $k \leftarrow k + 1$.
    $x \leftarrow x - \alpha \nabla f(x)$.
  end while
  return x, k

Os dados de entrada são os seguintes:

- f, a função a ser minimizada;
- x^0, o chute inicial para a solução;
- α, o tamanho do passo (learning rate). Deve ser um valor pequeno que muda a cada problema. Exemplo: $\alpha = 0.1$.
- ϵ, a tolerância. O algoritmo é interrompido se o gradiente fica menor que a tolerância (lembre-se que nosso objetivo seria $\|\nabla f(x)\| = 0$). Exemplo de tolerância: $\epsilon = 10^{-5}$.
- K, o número máximo de iterações. Impede que o algoritmo entre em um loop infinito se algo estiver dando errado. Deve ser um valor grande. Exemplo: $K = 10000$.

A saída será a solução aproximada x e o número de iterações perfomadas k (útil para termos noção do desempenho). Poderíamos também retornar o tempo de execução (para compararmos um método com o outro para o mesmo problema),

$\|\nabla f(x)\|$ (para, caso o número de iterações máximas tenha sido atingido, conseguirmos avaliar se estávamos longe da solução), etc.

1.2 Output gráfico

Para problemas em $\mathbb{R}^2$, será obrigatória a possibilidade de um output gráfico nos seguintes moldes: uma figura contendo as curvas de nível de f juntamente com a trajetória do algoritmo, ou seja, com os iterandos x conectados. Em breve coloco exemplo aqui de como deve ficar.

1.3 Exemplos

Além dos exemplos que o professor fornecerá para os testes, os alunos devem testar o algoritmo em 3 exemplos de função no $\mathbb{R}^2$, que podem ser inventados ou colhidos da literatura, sendo uma quadrática e duas não quadráticas.

2 Implementação

2.1 Primeira linha da função

```
def gd(f,x0,grad,eps = 1e-5,alpha = 0.1,itmax = 10000,fd = False,h =
                1e-7,plot = False,search = False,):
```

sendo

- `f:` função a ser minimizada. Deve ser conforme explicado abaixo;
- `x0:` chute inicial. Deve ser um numpy array de n componentes;
- `grad:` função que calcula o gradiente da função no ponto. Será explicada abaixo;
- `eps:` tolerância para o critério de parada (default: 10^{-5});
- `alpha:` o tamanho do passo (default: 0.1);
- `itmax:` o número máximo de iterações permitidas (default: 10000);
- `fd:` `True` para o gradiente ser calculado por diferenças finitas (é um achievement), `False` para ser calculado pela função `grad` (default: `False`);
- `h:` passo a ser usada nas diferenças finitas (default: 10^{-7});
- `plot:` `True` para que haja output gráfico (o default é `False` apenas para não dar erro caso $n > 2$);
- `search:` `True` para fazer busca linear (é um achievement) (default = `False`).

Sobre a função `f:` deve ter como entrada um ponto `x`, que é um numpy array com n componentes, e como saída o valor da função neste ponto. Um exemplo em que $n = 2$: a função $f(x) = \cos(x_1^2 x_2^3)$ pode ser implementada como

```
def f(x):
   return np.cos(x[0]**2*x[1]**3)
```

Sobre a função `grad:` deve ter como entrada um ponto `x`, que é um numpy array com n componentes, e como saída o vetor gradiente no mesmo formato. Para o exemplo acima, podemos ter

```
def grad(x):
   return np.array([-2*x[0]*x[1]**3*np.sin(x[0]**2*x[1]**3),
                    -3*x[0]**2*x[1]**2*np.sin(x[0]**2*x[1]**3)])
```

3 Exemplo de uso

Considere as seguintes funções

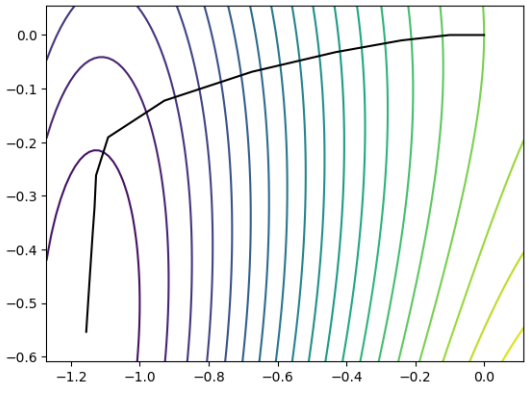
```
def f(x):
   return x[0]**4-2*x[0]**2+x[0]-x[0]*x[1]+x[1]**2
def grad(x):
   return np.array([4*x[0]**3-4*x[0]+1-x[1],-x[0]+2*x[1]])
```

Ao se rodar

```
x,k = gd(f,np.array([0,0]),grad,plot = True)
print(f"x = {x}")
print(f"k = {k}")
```

Devem aparecer na tela a solução aproximada `x`, o número de iterações utilizadas `k` e um gráfico mostrando a trajetória do método, mais ou menos conforme abaixo.

```
x = [-1.15796978, -0.57898067]
k = 60
```



4 Detalhes e comentários

- O método do gradiente dá origem a outro método, chamado método do gradiente estocástico (sgd), muito usado em aprendizagem de máquina;
- Não é necessário fazer tudo no $\mathbb{R}^n$, quem quiser pode supor que a f é sempre do $\mathbb{R}^2$ em $\mathbb{R}$. Isso facilita as coisas. Fazer em $\mathbb{R}^n$ para n genérico é um achievement;
- Para calcular a norma do vetor gradiente, pode ser usada a função `numpy.linalg.norm`. Para as curvas de nível, pode ser usada a função `matplotlib.pyplot.contour`;
- No uso do `matplotlib.pyplot.contour`, será útil a função `numpy.meshgrid`;
- Para gerar o output gráfico, há duas possibilidades: plotar o novo ponto cada vez que ele for gerado ou guardar todos os pontos em uma matriz e plotá-los todos de uma vez. Prefiro a segunda opção;
- As três funções do $\mathbb{R}^2$ que fazem parte do projeto devem ser entregues implementadas com seus respectivos gradientes.

5 Updates neste arquivo

- 07/11 v.2 - foi incluído um exemplo de saída esperada da função gd.
- 08/11 - foi retirada a exigência de que os numpy arrays sejam colunas. Façam com numpy arrays normais. Foi retirada a sugestão de que no output se plote poucos iterandos (pode plotar todos). Corrigido um erro no exemplo de uso.