Aplicação de Métodos Estocásticos para a Otimização Do Problema Apresentado

CEFET-RJ - Unidade de Nova Friburgo

Disciplina: Problemas Inversos em Python

Professora: Josiele da Silva Teixeira

Aluno: Raul Martins Furtado Fernandes (raul.fernandes@aluno.cefet-rj.br)

RESUMO. O objetivo deste trabalho foi a implementação computacional de dois métodos estocásticos para determinar o valor ótimo do calor específico de uma placa de alumínio a partir de dados experimentais apresentados: Luus-Jaakola e Evolução Diferencial. São apresentados e analisados os resultados obtidos com cada método com diferentes combinações de parâmetros.

1. INTRODUÇÃO

A otimização de funções é uma área fundamental na pesquisa operacional e na ciência da computação, desempenhando um papel crucial em diversos campos como engenharia, economia, biologia, entre outros. A capacidade de encontrar soluções ótimas para problemas complexos pode significar grandes avanços em eficiência e desempenho.

Métodos estocásticos de otimização têm sido amplamente estudados e aplicados devido à sua habilidade de lidar com problemas de otimização contínuos e não-lineares. Esses métodos, muitas vezes baseados em processos probabilísticos e aleatórios, oferecem alternativas robustas aos métodos clássicos de otimização. Neste trabalho, focamos em dois métodos: Luus-Jaakola e Evolução Diferencial, ambos com características únicas que os tornam adequados para diferentes tipos de problemas de otimização.

2. APRESENTAÇÃO DO PROBLEMA E DOS MÉTODOS DE OTIMIZAÇÃO UTILIZADOS

A partir dos dados experimentais fornecidos, foi implementada uma função objetivo que calcula o custo associado ao calor específico da placa de alumínio (*cp*). O custo reflete a discrepância entre os valores de temperatura calculados pela função teórica e os valores observados experimentalmente para diferentes instantes de tempo (*t*). Assim, a função objetivo busca quantificar a precisão do *cp* proposto, permitindo a identificação do valor mais adequado por meio de métodos de otimização.

2.1. Método Luus-Jaakola

O método de Luus-Jaakola é um algoritmo de busca estocástica que se baseia na geração de novas soluções dentro de um intervalo decrescente ao longo das iterações. Ele é apreciado por sua capacidade de encontrar soluções próximas do ótimo global com um consumo reduzido de recursos computacionais.

Este método recebe como parâmetro a função objetivo a ser otimizada (*fObj*), os limites mínimo e máximo para cada parâmetro de *fObj* (*bounds*), o número de iterações internas (*nInt*), o número de iterações externas (*nOut*) e o coeficiente de contração (*coef*).

O método Luus-Jaakola é valorizado pela sua simplicidade e eficiência, proporcionando um bom equilíbrio entre resultado e custo computacional. A parametrização adequada do número de iterações e do coeficiente de contração é essencial para obter bons resultados.

2.1.1. Descrição do Algoritmo

1) Inicialização
   a) A variável *stepSize* é definida como a diferença entre o limite máximo e mínimo para cada parâmetro de *fObj*;
   b) Um valor aleatório (*x*) é gerado para cada parâmetro de *fObj* dentro dos limites estabelecidos;
   c) O resultado função objetivo para *x* é salvo em *fx*;
2) Iterações
   a) O algoritmo realiza *nOut* iterações externas. Em cada iteração externa são realizadas *nInt* iterações internas e em seguida *stepSize* é multiplicado por (1 – *coef*) para reduzir o intervalo de busca;
   b) Em cada iteração interna, uma nova solução (*newX*) é gerada adicionando uma variação aleatória proporcional ao *stepSize* à solução atual (*x*). Esta nova solução é ajustada para ficar dentro dos limites definidos e, em seguida, *fObj* é avaliada com *newX* e o resultado é salvo em *newFx*. Se *newFx* for menor que *fx*, a solução ótima atual *x* e seu valor *fx* são atualizados;
3) Fim
   a) O melhor valor encontrado para o parâmetro de *fObj* e assim como o resultado da função objetivo com esse valor são retornados.

2.2. Método Evolução Diferencial

A Evolução Diferencial é um algoritmo estocástico de otimização que opera com uma população de soluções candidatas, utilizando operações de mutação, recombinação e seleção para explorar o espaço de busca e encontrar soluções ótimas.

Este método recebe como parâmetro a função objetivo a ser otimizada (*fObj*), os limites mínimo e máximo para o parâmetro de *fObj* (*bounds*), o tamanho da população (*nPop*), a quantidade de gerações (*nGen*), a taxa de mutação (*F*) e a taxa de recombinação (*CR*).

A Evolução Diferencial é valorizada por sua capacidade de encontrar soluções precisas através da combinação de indivíduos da população, promovendo a diversidade e explorando eficazmente o espaço de soluções. A correta escolha dos parâmetros é crucial para maximizar a performance e a eficiência do algoritmo, garantindo que ele possa explorar o espaço de busca de maneira equilibrada.

2.2.1. Descrição do Algoritmo

1) Inicialização
   a) A variável *pop* é criada como um vetor de *nPop* elementos e cada elemento é gerado aleatoriamente obedecendo os limites mínimo e máximo *bounds*;
   b) A variável *results* é criada como um vetor de *nPop* elementos onde *results[i] = fObj(pop[i])*;
2) Gerações
   a) O algoritmo realiza *nGen* iterações que representam gerações. A cada geração são performadas operações de mutação, recombinação e seleção;

b) Mutação
   i) A variável *mutantPop* é criada como um vetor de nPop elementos;
   ii) São realizadas *nPop* iterações incrementando um contador *i*;
   iii) A cada iteração são gerados 3 outros índices aleatórios diferentes de *i* e diferentes entre si;
   iv) Um elemento mutante (*mutant*) é gerado a partir da seguinte fórmula: *pop[r1] + F * (pop[r2] – pop[r3])*;
   v) *mutant* é ajustado para ficar dentro dos limites de *bounds* e adicionado no vetor *mutantPop;*
c) Recombinação
   i) A variável *trialPop* é criada como uma cópia de *pop*;
   ii) Um índice de *pop* é gerado aleatoriamente (*randI*);
   iii) São realizadas *nPop* iterações incrementando um contador *i*;
   iv) Em cada iteração, um valor aleatório entre 0 e 1 é gerado. Se esse valor for menor ou igual a *CR* ou se *i* for igual a *randI*, *trialPop[i] = mutantPop[i]*;
d) Seleção
   i) O vetor *trialResults* é inicializado com o resultado da função objetivo para cada elemento de *trialPop*;
   ii) Cada elemento de *trialResults* é comparado com o elemento de mesmo índice de *results*. Se for menor, os elementos de *pop* e de *results* dessa posição serão substituídos pelos elementos de *trialPop* e *trialResults*, respectivamente;

3) Fim
   a) Ao fim de todas as gerações, o elemento com menor valor em *results* é retornado juntamente com o elemento de *pop* que o gerou.

3. ANÁLISE DOS RESULTADOS

Com os métodos de Luus-Jaakola e Evolução Diferencial implementados, foram realizadas análises para identificar como os parâmetros de cada método influenciam a estimativa do calor específico da placa de alumínio.

3.1. Luus-Jaakola

A Tabela 1 apresenta os resultados obtidos pelo método Luus-Jaakola para diferentes configurações de *nInt*, *nOut* e *coef*.

Tabela 1 – Resultados Luus-Jaakola

Coeficiente de Contração (coef)	*nInt* =25, *nOut* =25		*nInt* =50, *nOut* =50		*nInt* =75, *nOut* =75		*nInt* =100, *nOut* =100	
	cp	*fObj(cp)*	*cp*	*fObj(cp)*	*cp*	*fObj(cp)*	*cp*	*fObj(cp)*
0.05	918.4419974	8330.38282	918.3738505	8330.382406	918.3890233	8330.382331	918.3931709	8330.382327
0.1	918.3420892	8330.382871	918.3955407	8330.382329	918.3930895	8330.382327	918.3932595	8330.382327
0.2	918.3972937	8330.382331	918.3932661	8330.382328	918.3932544	8330.382327	918.3932597	8330.382327

A tabela apresenta os valores de *cp* estimados e os custos associados (*fObj(cp)*) obtidos a partir da variação do coeficiente de contração (*coef*) e do número de iterações internas (*nInt*) e externas (*nOut*) no método Luus-Jaakola. Observa-se que, à medida que *nInt* e *nOut* aumentam, os valores de *cp* convergem gradualmente para um intervalo mais consistente, independentemente do coeficiente de contração utilizado.

Além disso, para *coef*=0.05, os valores de *cp* apresentam maior variação nas iterações iniciais (*nInt*=25, *nOut*=25), enquanto para valores maiores de *coef* (*coef*=0.1 e *coef*=0.2), o *cp* converge de maneira mais estável, especialmente com *nInt* e *nOut* a partir de 75.

Os custos (*fObj(cp)*) permanecem praticamente constantes, com pequenas diferenças decimais que não afetam significativamente a qualidade da solução. Esses resultados indicam que o aumento de *nInt* e *nOut* melhora a precisão na busca pelo valor ideal de *cp*, enquanto o coeficiente de contração exerce menor influência nos estágios avançados da otimização.

3.2. Evolução Diferencial

A Tabela 2 apresenta os resultados obtidos pelo método Evolução Diferencial para diferentes configurações de *nGen*, *F* e *CR* considerando *nPop*=10.

Tabela 2 – Resultados Evolução Diferencial

Quantidade de Gerações (*nGen*)	Taxa de Mutação (*F*)	*CR* =0.2		*CR* =0.4		*CR* =0.6		*CR* =0.8	
		cp	*fObj(cp)*	*cp*	*fObj(cp)*	*cp*	*fObj(cp)*	*cp*	*fObj(cp)*
25	0.25	918.7673376	8330.411355	918.3939471	8330.382328	918.3936437	8330.382328	918.3931988	8330.382327
	0.75	918.081945	8330.40244	918.198737	8330.390179	918.3928732	8330.382328	918.3970274	8330.38233
	1.25	918.9173402	8330.439296	918.960651	8330.449099	918.4280224	8330.382578	918.3558684	8330.382618
	1.75	918.2186965	8330.388651	918.3497793	8330.38272	918.5261827	8330.385993	917.7942961	8330.456793
50	0.25	918.3847667	8330.382342	918.3932524	8330.382327	918.3932597	8330.382327	918.3932582	8330.382327
	0.75	918.3821536	8330.382353	918.3919511	8330.382328	918.3932564	8330.382327	918.3932555	8330.382327
	1.25	918.5369418	8330.386611	918.393807	8330.382328	918.3921249	8330.382328	918.3932718	8330.382327
	1.75	918.2186965	8330.388651	918.3497793	8330.38272	918.3840819	8330.382345	918.3871197	8330.382335
75	0.25	918.3932362	8330.382327	918.3932543	8330.382327	918.3932546	8330.382327	918.3932597	8330.382327
	0.75	918.3925144	8330.382328	918.3932534	8330.382327	918.3932553	8330.382327	918.3932563	8330.382327
	1.25	918.398196	8330.382333	918.393807	8330.382328	918.3932497	8330.382327	918.3932569	8330.382327
	1.75	918.4154841	8330.38243	918.3985566	8330.382333	918.3941388	8330.382328	918.3932029	8330.382327
100	0.25	918.3932362	8330.382327	918.3932545	8330.382327	918.3932546	8330.382327	918.3932597	8330.382327
	0.75	918.3932642	8330.382327	918.3932599	8330.382327	918.3932564	8330.382327	918.3932563	8330.382327
	1.25	918.3977127	8330.382332	918.3933064	8330.382327	918.3932575	8330.382327	918.3932569	8330.382327
	1.75	918.3925451	8330.382328	918.3938059	8330.382328	918.393233	8330.382327	918.3932529	8330.382327

A análise entre as variáveis do método de evolução diferencial (*nPop*, *nGen*, *F* e *CR*) e o valor de *fObj(cp)* indicou que as variáveis *nPop* e *nGen* possuem o maior impacto no valor de *fObj(cp)*. Observou-se que um aumento no tamanho da população e/ou no número de gerações resulta em uma diminuição significativa no valor de fObj(cp). Entretanto na Tabela 2, foi fixado o tamanho da população como 10 para permitir que o impacto das outras variáveis no valor de *fObj(cp)* fosse observado com mais clareza.

Já a taxa de mutação (*F*) apresentou uma influência direta no valor de *fObj(cp)*. Valores mais baixos de *F* geraram valores mais baixos de *fObj(cp)*.

A taxa de cruzamento (CR), por sua vez, não apresentou tanta influência no valor de *fObj(cp)*, mas foi possível observar que valores de *CR* mais próximos a 1 geraram resultados levemente menores.

Portanto, conclui-se que as variáveis mais influentes no valor de *fObj(cp)* são o tamanho da população e o número de gerações, seguidas pela taxa de mutação e pela taxa de cruzamento.

4. CONCLUSÃO

Neste trabalho, foram comparados dois métodos de otimização: Evolução Diferencial e Luus-Jaakola, com o objetivo de encontrar o melhor valor para o calor específico da placa de alumínio a fim de que a temperatura calculada a partir desse valor seja mais próxima possível da temperatura coletada no experimento. Observou-se que o método de Evolução Diferencial alcançou o menor valor de *fObj(cp)*, destacando-se pela sua eficácia. No entanto, o método de Luus-Jaakola obteve resultados muito próximos aos da Evolução Diferencial, mas com um consumo significativamente menor de recursos computacionais.

Essa análise demonstra que, embora a Evolução Diferencial tenha se mostrado ligeiramente mais eficaz em termos de valor absoluto da função objetivo, o método de Luus-Jaakola apresentou um melhor custo-benefício devido à sua eficiência computacional.

É importante ressaltar que ambos os métodos requerem uma parametrização adequada para alcançar os melhores resultados possíveis. A correta escolha dos parâmetros é crucial para maximizar a performance e a eficiência dos algoritmos de otimização.

Dessa forma, conclui-se que tanto a Evolução Diferencial quanto o Luus-Jaakola são métodos de otimização viáveis, sendo a escolha do método dependente das especificidades do problema e dos recursos disponíveis.

Apêndice 1 – Implementação do Luus-Jaakola

```python
import numpy as np

def luusJaakola(func, bounds, coef = 0.01, nInt = 100, nOut = 100):
    dim = len(bounds)

    stepSize = np.array([high - low for low, high in bounds])

    x = np.array([np.random.uniform(low, high) for low, high in bounds])

    fx = func(x)

    for i in range(int(nOut)):
        for j in range(int(nInt)):
            newX = x + stepSize * (np.random.rand(dim) - 0.5)
            newX = np.clip(newX, [low for low, high in bounds], [high for low, high in bounds])


            newFx = func(newX)

            if newFx < fx:
                x, fx = newX, newFx
                print(f"x: {x} | fx: {fx}")

        stepSize = np.multiply(stepSize, 1 - coef)

        if (np.max(stepSize) < 10 ** -6):
            return x, fx

    return x, fx
```

Apêndice 2 – Implementação da Evolução Diferencial

```python
import numpy as np

def evolucaoDiferencial(fObj, bounds, nPop=20, nGen=1000, F=0.8, CR=0.7):
    min, max = bounds
    pop = np.random.uniform(min, max, nPop)
    results = np.array([fObj([x]) for x in pop])

    bestI = np.argmin(results)

    print(f"Inicio | Best x: {pop[bestI]} | Best F(x): {results[bestI]}")

    for gen in range(nGen):
        trialPop = np.copy(pop)

        # Mutation
        mutantPop = []

        for i in range(nPop):
            indexes = list(range(nPop))
            indexes.remove(i)
            r1, r2, r3 = np.random.choice(indexes, 3, replace=False)

            mutant = pop[r1] + F * (pop[r2] - pop[r3])
            mutant = np.clip(mutant, min, max)

            mutantPop.append(mutant)

        #Crossover
        randI = np.random.choice(list(range(nPop)), 1)

        for i in range(nPop):
            if i == randI or np.random.rand() <= CR:
                trialPop[i] = mutantPop[i]

        # Selection
        trialResults = np.array([fObj([x]) for x in trialPop])

        oldBestFx = results[np.argmin(results)]

        for i in range(nPop):
            if (trialResults[i] < results[i]):
                pop[i] = trialPop[i]
                results[i] = trialResults[i]

        bestI = np.argmin(results)

        if (results[bestI] < oldBestFx):
            print(f"Gen: {gen + 1} | Best x: {pop[bestI]} | Best F(x): {results[bestI]}")

    bestI = np.argmin(results)
    bestX = pop[bestI]
    bestFx = results[bestI]

    return bestX, bestFx
```

Apêndice 3 – Implementação da Função Objetivo

```python
import numpy as np
from expData import expData

def tCalc(t, cp):
    a1 = 0.001849 # m^2
    a2 = 0.0023177 # m^2
    v = 5.0385 * (10 ** -6) # m^3
    tInf = 25 # ºC
    h = 21.7 # W / m^2 * K
    tIni = 25 # ºC
    q2 = 742.2 # W / m^2
    p = 2702 # kg / m^3

    if (t == 0):
        return tIni

    x = (q2 * a1) / (h * a2)
    y = 1 - np.exp((-h * a2 * t) / (p * v * cp))

    return tInf + x * y

def tMed(cp):
    results = []

    for exp in expData:
        t = exp['t']
        results.append(tCalc(t, cp))

    return results

def fObj(cp):
    tCalc = tMed(cp[0])
    soma = 0

    for i in range(len(expData)):
        soma += ((tCalc[i] - expData[i]['tExp']) ** 2) / (0.07 ** 2)

    return soma
```

Apêndice 4 – Implementação do arquivo main.py

```python
from luus_jaakola import luusJaakola
from evolucao_diferencial import evolucaoDiferencial
from temperaturaMedia import fObj

nInt=100
nOut=100
coef=0.2

x1, fx1 = luusJaakola(fObj, [(750, 1000)], coef, nInt, nOut)

print(f"coef={coef} | nInt={nInt} | nOut={nOut}")
print(f"Luus Jaakola -> fObj({x1[0]}): {fx1}")


nPop=10
nGen=100
F=0.25
CR=0.8

x2, fx2 = evolucaoDiferencial(fObj, (750, 1000), nPop, nGen, F, CR)

print(f"nPop={nPop} | nGen={nGen} | F={F} | CR={CR}")
print(f"Evolução Diferencial -> fObj({x2}): {fx2}")
```

Apêndice 5 – Dados experimentais

```
expData = [
    {"i": 1, "t": 5, "tExp": 25.01},
    {"i": 2, "t": 20, "tExp": 25.65},
    {"i": 3, "t": 30, "tExp": 27.23},
    {"i": 4, "t": 40, "tExp": 28.66},
    {"i": 5, "t": 60, "tExp": 31.31},
    {"i": 6, "t": 80, "tExp": 33.41},
    {"i": 7, "t": 100, "tExp": 35.26},
    {"i": 8, "t": 150, "tExp": 38.93},
    {"i": 9, "t": 200, "tExp": 41.47},
    {"i": 10, "t": 250, "tExp": 43.41},
    {"i": 11, "t": 300, "tExp": 44.78},
    {"i": 12, "t": 350, "tExp": 45.93},
    {"i": 13, "t": 400, "tExp": 46.82},
    {"i": 14, "t": 450, "tExp": 47.50},
    {"i": 15, "t": 500, "tExp": 48.14},
    {"i": 16, "t": 550, "tExp": 48.68},
    {"i": 17, "t": 600, "tExp": 48.95},
    {"i": 18, "t": 650, "tExp": 49.16},
    {"i": 19, "t": 700, "tExp": 49.51},
    {"i": 20, "t": 750, "tExp": 49.83},
    {"i": 21, "t": 800, "tExp": 50.08},
    {"i": 22, "t": 850, "tExp": 50.23},
    {"i": 23, "t": 900, "tExp": 50.22},
    {"i": 24, "t": 950, "tExp": 50.39},
    {"i": 25, "t": 1000, "tExp": 50.61},
    {"i": 26, "t": 1050, "tExp": 50.68},
    {"i": 27, "t": 1100, "tExp": 50.88},
    {"i": 28, "t": 1150, "tExp": 51.00},
    {"i": 29, "t": 1200, "tExp": 51.19},
    {"i": 30, "t": 1250, "tExp": 51.29},
    {"i": 31, "t": 1300, "tExp": 51.42},
    {"i": 32, "t": 1350, "tExp": 51.39},
    {"i": 33, "t": 1400, "tExp": 50.85},
    {"i": 34, "t": 1450, "tExp": 51.07},
    {"i": 35, "t": 1500, "tExp": 51.22},
    {"i": 36, "t": 1550, "tExp": 51.36},
    {"i": 37, "t": 1600, "tExp": 51.41},
    {"i": 38, "t": 1650, "tExp": 51.45},
    {"i": 39, "t": 1700, "tExp": 51.53},
    {"i": 40, "t": 1750, "tExp": 51.73},
    {"i": 41, "t": 1800, "tExp": 51.75},
    {"i": 42, "t": 1850, "tExp": 51.68},
    {"i": 43, "t": 1900, "tExp": 51.65},
    {"i": 44, "t": 1950, "tExp": 51.67},
    {"i": 45, "t": 2000, "tExp": 51.69},
    {"i": 46, "t": 2100, "tExp": 51.64},
    {"i": 47, "t": 2200, "tExp": 51.80},
    {"i": 48, "t": 2300, "tExp": 51.93},
    {"i": 49, "t": 2400, "tExp": 52.00},
    {"i": 50, "t": 2500, "tExp": 52.02},
    {"i": 51, "t": 2600, "tExp": 52.07},
    {"i": 52, "t": 2700, "tExp": 52.10},
    {"i": 53, "t": 2800, "tExp": 52.11},
    {"i": 54, "t": 2900, "tExp": 52.14},
    {"i": 55, "t": 3000, "tExp": 52.30},
    {"i": 56, "t": 3100, "tExp": 52.18}
]
```