

Tiago Correa Prata

# **Controle Preditivo Baseado em Modelo (MPC) aplicado a uma planta didática**

São Paulo

2019



Tiago Correa Prata

## **Controle Preditivo Baseado em Modelo (MPC) aplicado a uma planta didática**

Projeto de pesquisa apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de São Paulo para a qualificação no programa de mestrado em engenharia de automação e controle.

Instituto Federal de São Paulo - IFSP

Orientador: Prof. Dr. Alexandre Brincalepe Campo

São Paulo

2019



Tiago Correa Prata

## **Controle Preditivo Baseado em Modelo (MPC) aplicado a uma planta didática**

Projeto de pesquisa apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de São Paulo para a qualificação no programa de mestrado em engenharia de automação e controle.

---

**Prof. Dr. Alexandre Brincalepe  
Campo**  
Orientador

---

**Prof. Dr. Eduardo Alves da Costa**  
Convidado (interno)

---

**Prof. Dr. Diego Colón**  
Convidado (externo)

---

**Prof. Dr. Ricardo Pires**  
Suplente (interno)

---

**Prof. Dr. Wânderson de Oliveira Assis**  
Suplente (externo)

São Paulo  
2019



# Resumo

O **MPC** é uma técnica de controle que vem ganhando atenção e refinamento ao longo dos últimos 50 anos e, atualmente, ela figura entre uma das técnicas com maior destaque no controle preditivo, tendo inúmeras aplicações comerciais, desde sua implementação no controle de processos multivariáveis em indústrias químicas, até a construção de lógicas para a orientação de veículos autônomos.

Esta dissertação trata do desenvolvimento de um controle preditivo baseado em modelo, aplicado a uma planta piloto, abordando todo o processo de construção e desenvolvimento do mesmo, visando proporcionar um entendimento claro de cada um das etapas.

É possível dividir este projeto em 6 diferentes partes: teoria base do controle preditivo; teoria sobre o **MPC**; identificação de sistemas; construção do controlador preditivo; aplicação e análises. Toda a parte prática deste trabalho é desenvolvida em **MATLAB®** e/ou *Python*, e implementada em uma planta educacional de fácil acesso, visando uma fácil replicabilidade.

**Palavras-chave:** controle preditivo baseado em modelo; controle preditivo; teoria de controle; identificação de sistemas



# Abstract

**MPC** is a control technique that has been gaining attention and refinement over the last 50 years and it is currently one of the most prominent techniques in predictive control, having numerous commercial applications since its implementation in multivariate chemical process control, to the construction of logics for the orientation of autonomous vehicles.

This dissertation deals with the development of a model based predictive control, applied to a pilot plant, addressing the whole process of construction and development of it, aiming to provide a clear understanding of each of the stages.

This project can be divided into 6 different parts: basic theory of predictive control; theories about **MPC**; systems identification; construction of the predictive controller; application and analysis. The whole practical part of this project is developed in **MATLAB®** and/or *Python*, and implemented in an easily accessible educational plan, aiming for easy replicability.

**Keywords:** model predictive control; predictive control; control theory; system identification



# Listas de ilustrações

Figura 1 – Pesquisa em grade com número escalar . . . . .	31
Figura 2 – Pesquisa em grade com duas variáveis e sem restrição . . . . .	32
Figura 3 – Pesquisa em grade com duas variáveis e com restrição . . . . .	32
Figura 4 – Influência de $f'(x_k)$ no cálculo de $x_{k+1}$ no método de descidas mais íngremes . . . . .	33
Figura 5 – Método de descida mais íngrime convergindo para mínimo global . . . . .	36
Figura 6 – Método de descida mais íngrime convergindo para mínimo local . . . . .	36
Figura 7 – Diagrama de blocos do quadrado da diferença entre o sistema real e o modelo . . . . .	37
Figura 8 – Horizonte de tempo analisado pelo método Estimador de Horizonte Móvel . . . . .	38
Figura 9 – Simulação utilizando Estimador de Horizonte Móvel . . . . .	42
Figura 10 – Estrutura básica do MPC . . . . .	43
Figura 11 – Ação do Estimador de Horizonte Móvel e do Controle Preditivo Baseado em Modelo . . . . .	45
Figura 12 – Planta de aquecimento de ar . . . . .	47
Figura 13 – Simulação utilizando Controle Preditivo Baseado em Modelo . . . . .	48
Figura 14 – Laboratório de Controle de Temperatura . . . . .	51
Figura 15 – Simulink para coleta de dados SISO . . . . .	60
Figura 16 – Gráfico da coleta de dados SISO . . . . .	60
Figura 17 – Modelos para planta SISO . . . . .	61
Figura 18 – Diagrama de bloco para parametrização de MPC em planta SISO . . . . .	61
Figura 19 – Diagrama de bloco para aplicação de MPC em planta SISO . . . . .	62
Figura 20 – MPC em planta SISO - <i>Setpoint</i> e Valor medido . . . . .	63
Figura 21 – MPC em planta SISO - Potência do aquecedor de entrada . . . . .	63
Figura 22 – MPC em planta SISO - Erro calculado . . . . .	63



# **Lista de tabelas**

Tabela 1 – Valores para modelagem SISO do TCLab . . . . .	52
Tabela 2 – Valores para modelagem MIMO do TCLab . . . . .	53
Tabela 3 – Características do controlador MPC para planta SISO . . . . .	62
Tabela 4 – Cronograma de trabalho . . . . .	65



# **Lista de códigos-fonte**

2.1	Busca por descidas mais íngremes . . . . .	34
A.1	Pesquisa em grade com número escalar . . . . .	73
A.2	Pesquisa em grade com duas variáveis . . . . .	74
A.3	Exemplo do método Estimador de Horizonte Móvel . . . . .	75
A.4	Exemplo de aplicação do Controle Preditivo Baseado em Modelo . . . . .	82



# Lista de abreviaturas e siglas

AR	modelo autorregressivo
ARMA	<i>Auto-Regressive Moving Average</i>
ARMAX	modelo autorregressivo com média móvel e entrada exógena
ARX	modelo autorregressivo com entrada exógena
BYU	<i>Brigham Young University</i>
DMC	<i>Dynamic Matrix Control</i>
GBN	<i>Generalized Binary Noise</i>
GPC	<i>Generalized Predictive Controle</i>
MATLAB®	<i>Matrix Laboratory</i>
MHE	Estimador de Horizonte Móvel
MHPC	<i>Model Heuristic Predictive Control</i>
MIMO	<i>Multiple Input Multiple Output</i>
MPC	Controle Preditivo Baseado em Modelo
NCSU	<i>North Carolina State University</i>
NLP	<i>Nonlinear Programming</i>
PID	Proporcional Integral Derivativo
PRBS	<i>Pseudo-Random Binary Signal</i>
s.a.	sujeito a
SI	Sistema Internacional de Unidades
SISO	<i>Single Input Single Output</i>
STR	<i>Self-Tuning Regulator</i>
TCLab	<i>Temperature Control Lab</i>



# Listado de símbolos

$x_{opt}$  Valor óptimo de  $x$  para minimizar  $f(x)$



# Sumário

<b>I</b>	<b>APRESENTAÇÃO</b>	<b>21</b>
1	<b>INTRODUÇÃO</b>	23
1.1	<b>Introdução histórica ao MPC</b>	24
1.2	<b>Objetivos</b>	25
1.3	<b>Motivação e justificativas</b>	26
<b>II</b>	<b>REVISÃO DE LITERATURA</b>	<b>27</b>
2	<b>OTIMIZAÇÃO</b>	29
2.1	<b>O problema da otimização</b>	29
2.2	<b>Algoritmos de otimização</b>	30
2.2.1	Método de pesquisa em grade	30
2.2.2	Método de busca de descidas mais íngremes	32
2.2.3	Otimizadores de programação não-linear (NLP)	36
2.3	<b>Algumas aplicações de otimização</b>	37
2.3.1	Estimação de parâmetros	37
2.3.2	Estimador de Horizonte Móvel (MHE)	38
2.3.2.1	Exemplo	41
3	<b>CONTROLE PREDITIVO BASEADO EM MODELO</b>	43
3.1	<b>Aplicação prática</b>	44
3.1.1	Exemplo	46
<b>III</b>	<b>MATERIAIS E MÉTODOS</b>	<b>49</b>
4	<b>PLANTA PILOTO</b>	51
4.1	<b>Modelagem da planta piloto</b>	52
4.1.1	Modelo SISO	52
4.1.2	Modelo MIMO	52
5	<b>METODOLOGIA</b>	55
5.1	<b>Modelagem experimental</b>	55
5.1.1	Testes dinâmicos e coleta de dados	55
5.1.2	Escolha da representação matemática	56
5.1.3	Determinação da estrutura do modelos	56

5.1.4	Estimação de parâmetros . . . . .	57
5.1.5	Validação do modelo . . . . .	57
<b>5.2</b>	<b>Desenvolvimento do controlador . . . . .</b>	<b>57</b>
<b>6</b>	<b>PROCEDIMENTOS EXPERIMENTAIS . . . . .</b>	<b>59</b>
<b>6.1</b>	<b>Planta SISO . . . . .</b>	<b>59</b>
6.1.1	Identificação do modelo . . . . .	59
6.1.1.1	Coleta . . . . .	59
6.1.1.2	Determinação da estrutura, parâmetros e validação . . . . .	59
6.1.2	Desenvolvimento do controlador . . . . .	61
6.1.3	Aplicação do controlador MPC na planta SISO . . . . .	62
<b>7</b>	<b>CRONOGRAMA . . . . .</b>	<b>65</b>
	<b>REFERÊNCIAS . . . . .</b>	<b>67</b>
	<b>APÊNDICES</b>	<b>71</b>
	<b>APÊNDICE A – CÓDIGOS-FONTE . . . . .</b>	<b>73</b>
A.1	Método de pesquisa em grade . . . . .	73
A.2	Método de pesquisa em grade com duas variáveis . . . . .	74
A.3	Exemplo do método Estimador de Horizonte Móvel . . . . .	75
A.4	Exemplo de aplicação do Controle Preditivo Baseado em Modelo . . . . .	82
	<b>ANEXOS</b>	<b>91</b>
	<b>ANEXO A – DOCUMENTAÇÃO DE FUNÇÕES . . . . .</b>	<b>93</b>
A.1	MATLAB® - fmincon . . . . .	93
A.2	Python - scipy.optimize.minimize . . . . .	105

Parte I

Apresenta $\tilde{\text{a}}$ o



# 1 Introdução

O controle de processos tem fundamental importância no desenvolvimento industrial, sendo amplamente utilizado em praticamente todos os segmentos da indústria, contribuindo de maneira significativa para a maior velocidade na estabilização de sinais, aumento da qualidade de produtos, diminuição de riscos e redução de custos operacionais (OGATA, 2010). Seu objetivo, de forma simplificada, consiste em avaliar e corrigir desvios entre um valor desejado e o real valor medido na saída da planta para uma dada variável do processo (ou variáveis, nos casos de processos com múltiplas entradas e múltiplas saídas, também conhecidos por sua sigla em inglês MIMO (*Multiple Input Multiple Output*))). A aplicação correta de estratégias de controle acarreta numa operação eficiente da planta, mantendo suas variáveis relevantes em condições próximas às desejadas. A sintonia bem-feita do controle auxilia também na otimização do processo, possibilitando que o sistema opere com menor variabilidade, maximizando a produção e minimizando a utilização de recursos (OGATA, 2010). No capítulo 2 abordaremos mais sobre otimização. Modelos matemáticos podem auxiliar a estratégia de controle uma vez que um modelo da planta ou processo pode ser utilizado para estabelecer a relação existente entre as variáveis manipuladas e variáveis controladas, assim podendo auxiliar na predição do comportamento dinâmico do sistema analisado (GARCIA, 2013). A modelagem matemática pode ser feita utilizando dados empíricos ou através da aplicação de relações físico-químicas.

A estratégia de controle predominante na indústria é o controle PID (Proporcional Integral Derivativo) (CAMACHO; ALBA; BORDONS, 2007) que, além de levar em consideração o efeito proporcional (P) do erro medido, também atua em desvios relativos aos efeitos integrais (I) e derivativos (D). Seu elevado número de aplicações deve-se a uma grande variedade de vantagens como: sua rápida implementação, facilidade de compreensão, disponibilidade em praticamente todas as plataformas industriais de controle e principalmente pelo fato de não requerer um modelo matemático do processo (GONÇALVES, 2012); porém apesar de poder ser aplicado com eficiência em muitos processos, o controle PID aparece com menos frequência em sistemas não-lineares, como em plantas de controle de pH, por exemplo (GONÇALVES, 2012). Em casos como esse, outra técnica bastante utilizada na indústria (porém em proporções bem menores que o PID) pode ser utilizada: o controle MPC (Controle Preditivo Baseado em Modelo, do inglês *Model Predictive Control*). Uma introdução histórica desta técnica é apresentada na seção a seguir e mais detalhes técnicos sobre ela serão apresentados no capítulo 3.

## 1.1 Introdução histórica ao MPC

Segundo Lee (2011) em meados dos anos 50 as características essenciais do MPC já podiam ser observadas nas primeiras instalações de supervisórios de controle computadorizados, porém, apesar de seus potenciais benefícios, esse tipo de controle não se difundiu muito devido aos esforços necessário para mantê-lo e ao seu alto valor, até que aproximadamente na metade dos anos 70 os microprocessadores e sistemas de controle distribuídos se tornassem mais baratos e confiáveis. Não coincidentemente, por volta desta época começaram a aparecer na indústria e em seminários, publicações relatando a aplicação bem sucedida de controle baseado em modelo. (LEE, 2011)

Lee (2011) também relata que ainda durante os anos 70 a publicação das técnicas *Model Heuristic Predictive Control* (MHPC - Controle Preditivo Heurístico de Modelo) e *Dynamic Matrix Control* (DMC - Controle Dinâmico de Matriz) mostrando grande sucesso prático, impulsionaram o uso delas e de técnicas similares em refinarias de todo o mundo ocidental. No geral os algorítmos por trás dessas técnicas apresentavam uma natureza heurística, empregavam modelos baseados em resposta de domínio no tempo, eram completamente determinísticos sem nenhum modelo explícito dos distúrbios, e não possuíam garantias de estabilidade nem métodos para sintonia. Na mesma época, porém independente dos desenvolvimentos na indústria de processos, surgia da comunidade de controle adaptativo o GPC (Controle Preditivo Generalizado, do inglês *Generalized Predictive Control*), que possuía motivações bem diferentes do DMC: o GPC pretendia oferecer uma nova alternativa ao regulador de autoajuste<sup>1</sup>, principalmente visando superar seu problema de robustez. Por causa disso ele aplicava problemas de controle multivariável, porém carecia na inclusão de restrições, que era considerada uma característica indispensável dos problemas de controle de processos.

Ao longo dos anos 80 os algorítmos MPC se espalharam comercialmente a o embasamento teórico da técnica começou a se consolidar. Nessa época muitas empresas (normalmente pequenas), que ofereciam comercialmente soluções utilizando controle MPC, foram adquiridas por empresas maiores como a Aspen Tech e a Honeywell (LEE, 2011). No campo teórico aumentaram as discussões e definições com relação a estabilidade, robustez e ao uso de modelos não-lineares. Além disso os pesquisadores notaram que a utilização de modelos de espaço de estado poderia trazer mais benefícios e juntamente com isso estabeleceram a forma padrão do algorítmo, demonstrada na eq. (1.1) (LEE, 2011) .

$$\min_{\hat{u}(0), \dots, \hat{u}(p-1)} \left\{ V_p \triangleq \sum_{i=0}^{p-1} \left( \hat{x}^T(i) Q \hat{x}(i) + \hat{u}^T(i) R \hat{u}(i) \right) + \hat{x}^T(p) Q_t \hat{x}(p) \right\} \quad (1.1)$$

<sup>1</sup> O regulador de autoajuste (ou apenas STR, do inglês, *Self-Tuning Regulator*) é uma técnica composta por três partes, um estimador de parâmetros, um cálculo de projeto e um regulador com parâmetros ajustáveis (ÅSTRÖM; WITTENMARK, 1985).

Sendo:

$$\hat{x}(i+1) = A\hat{x}(i) + B\hat{u}, \quad \hat{x}(0) = x_0$$

Ao longo das últimas décadas, desde o lançamento do DMC, o MPC tem sido alvo de muito estudo teórico e prático, e passou de uma aplicação indústrial heurística para uma das técnicas de controle mais influentes da atualidade (LEE, 2011). Hoje podemos considerar o MPC, segundo a definição de Seborg, Edgar e Mellichamp (2011), uma técnica de controle avançada para problemas de controle multivariável. Sendo que, mesmo para um processo de múltiplas entradas e múltiplas saídas, onde existem restrições a essas variáveis, é razoável assumir que possuindo o modelo dinâmico do processo e as medições atuais do mesmo, pode-se predizer os valores de saída futuros, e então calcular as mudanças nos valores de entrada baseando-se tanto nas previsões quanto no valor medido.

Após enormes avanços nas técnicas de resolução das equações aplicadas ao MPC, atualmente seu uso não se limita mais à sistemas onde o tempo de resposta seja relativamente lento, podendo então ser utilizado em diversas aplicações que eram consideradas impraticáveis no passado. Lee (2011) relata em seu artigo o uso do MPC no controle de tração de veículos, motorização automotiva, amortecimento de sistemas massa-mola magnéticamente acionados, e muitos outros.

## 1.2 Objetivos

Este trabalho propõe, como objetivo principal, desenvolver um controlador MPC aplicado a um sistema didático com restrições de atuação.

Além disso os seguintes objetivos específicos também serão realizados:

- Estudo do algoritmo do controle MPC
- Avaliação o desempenho do controlador MPC desenvolvido em comparação com um controlador PID
- Comparação entre a implementação em duas ou mais plataformas, como MATLAB<sup>®2</sup>, Python<sup>3</sup> ou similares.
- Compilação de material teórico e experimental sobre MPC para estudantes de graduação e pós-graduação, de língua portuguesa

<sup>2</sup> MATLAB<sup>®</sup> é uma plataforma de programação projetada especificamente para engenheiros e cientistas onde é possível desenvolver algoritmos, realizar a análise de dados, criar modelos, aplicações, dentre outras coisas.

<sup>3</sup> Python é uma linguagem de programação de alto nível, interpretada e orientada a objeto, muito utilizada atualmente para aplicações nas áreas de ciência de dados, aprendizagem de máquina, identificação de sistemas, etc.

### 1.3 Motivação e justificativas

Segundo Parkinson, Balling e Hedengren (2018) o processo de determinar o melhor modelo para uma aplicação ou processo é chamado de otimização. Normalmente engenheiros costumam implementar tais técnicas em seus processos visando aumentar a eficiência diminuindo os gastos, porém, as variáveis e limitantes do processo podem ser inúmeras, fazendo com que a tarefa de otimização se torne difícil. Para casos assim, ferramentas computacionais de otimização são essenciais. (PARKINSON; BALLING; HEDENGREN, 2018)

Dá-se o nome de Otimização Dinâmica ao processo de otimização que é realizado dinamicamente ao longo do processo e, segundo Borrelli, Bemporad e Morari (2017), esta se tornou uma ferramenta padrão na tomada de decisões numa grande variedade de áreas. O controle MPC é um modo de implementação da otimização dinâmica e a execução deste trabalho em torno dessa técnica se deve ao fato de que ao longo das últimas quase 4 décadas (LEE, 2011) ela evoluiu para dominar a indústria de processos, onde tem sido utilizada em milhares de problemas (BORRELLI; BEMPORAD; MORARI, 2017). Além de exemplos de utilização de técnica já apresentados na [seção 1.1](#), é importante destacar também sua utilização no controle dos processos de fabricação de cimento, torres de destilação, plantas de PVC, como descreve Camacho, Alba e Bordons (2007), e também seu crescimento em outros setores, como na indústria automobilista, onde o MPC é utilizado para o controle do sistema dinâmico de um automóvel. (YAKUB; MORI, 2013).

## Parte II

### Revisão de literatura



## 2 Otimização

### 2.1 O problema da otimização

Segundo Haugen (2018) normalmente problemas de otimização são apresentados como problemas de minimização, como: "Encontre o valor ótimo de  $x$  que minimize a função objetivo  $f(x)$ , levando em consideração qualquer restrição sobre  $x$  ou em função de  $x$ . A solução ótima é indicada por  $x_{opt}$ " (HAUGEN, 2018).

Haugen (2018) ainda mostra que há várias formas de formular matematicamente um problema de otimização (minimização), mas que, de forma geral, dado um modelo matemático  $M$ , é possível representá-lo como a minimização de  $x$  para uma função  $f(x)$ , ou seja:

$$\min_x f(x) \quad (2.1)$$

sujeito a (também denotado por "s.a.") restrições, que podem ser na forma de:

- Restrições de desigualdade:

$$g(x) \leq 0 \quad (2.2)$$

onde  $g$  pode ser uma função linear ou não-linear.

- Restrições de igualdade:

$$h(x) = 0 \quad (2.3)$$

onde  $h$  pode ser uma função linear ou não-linear de  $x$ .

- Limites superiores e inferiores

$$x_{li} \leq x \leq x_{ls} \quad (2.4)$$

Onde  $li$  e  $ls$  indicam 'limite inferior' e 'limite superior', respectivamente.

Sendo que as equações 2.2 e 2.3 definem restrições na relação entre as variáveis de otimização, enquanto 2.4 define as regiões limites destas mesmas variáveis.

Existem diversos métodos para encontrar a solução ótima para um problema de otimização e a seção a seguir irá mostrar exemplos e métodos numéricos simples para exemplificar, em maiores detalhes, como um problema de minimização pode ser resolvido. No capítulo 3 será feito uso das minimizações para compreender como o MPC calcula valores ótimos, dadas determinadas restrições em um dado horizonte de controle, pois

uma maior compreensão sobre problemas de minimização pode fazer grande diferença no entendimento do controle MPC em si.

## 2.2 Algoritmos de otimização

### 2.2.1 Método de pesquisa em grade

O método apresentado nesta seção não é aplicável a praticamente nenhum problema real devido a sua ineficiência computacional, porém ele ajuda a ilustrar o objetivo de todo o método numérico voltado para minimização.

Este método consiste em testar todos os valores de todas as variáveis (em um conjunto de dados definido) para verificar qual combinação minimiza a função objetivo, ou seja, testar todos os valores possíveis para  $x_1, x_2, x_3, \dots, x_n$  com o objetivo de encontrar o  $x_{opt}$ , valor de  $x$  que minimiza  $f$ .

No caso de um único  $x$ , um laço condicional simples poderia testar todos os valores da função objetivo. Para ilustrar essa ideia, no [apêndice A](#) o código-fonte [A.1](#), em Python, mostra como a função  $f(x)$  abaixo poderia ser computada, caso o intervalo de teste de  $x$  fosse igual 100, ou seja,  $N = 100$ .

$$f(x) = 0,00232x^4 - 0,111x^3 + 1,8x^2 - 11,6x + 34,4 \quad (2.5)$$

Sendo que:

$$2 \leq x \leq 22$$

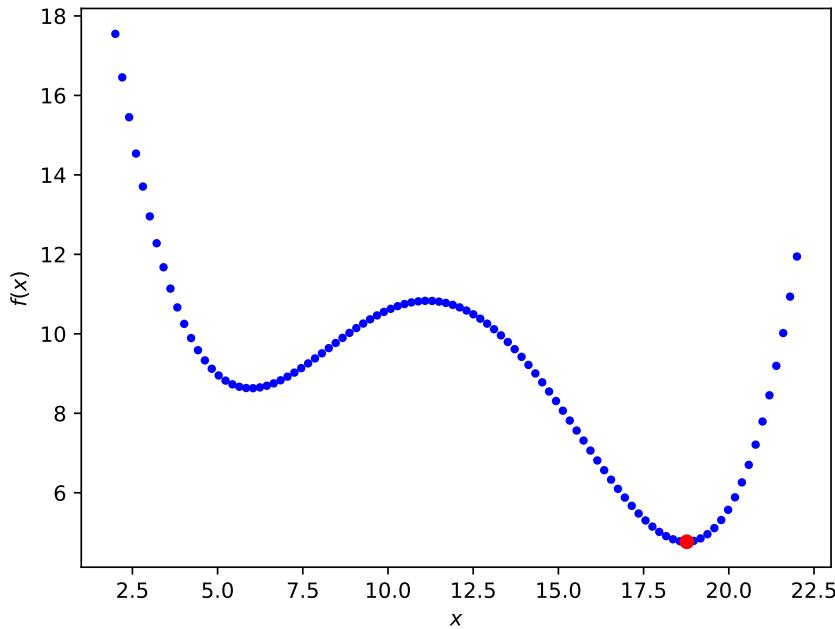
O intervalo  $N = 100$  indica que serão analisados 100 valores entre 2 e 22.

A [fig. 1](#) foi construída a partir do código-fonte [A.1](#), sendo destacado em vermelho, o valor de  $x$  onde a  $f(x)$  apresentava seu menor valor. Repare que o gráfico apresenta dois vales distintos: um deles é o já mencionado destaque em vermelho, onde o valor  $x$  é 18,76 e outro onde  $x$  vale aproximadamente 5,5. O vale do gráfico onde o valor de  $x$  produz o menor valor de  $f(x)$  é conhecido como *mínimo global*, todos os outros são *mínimos locais*, pois são os valores mínimos da função apenas para uma região limitada.

Algoritmos que buscam encontrar o valor mínimo de uma função podem erroneamente convergir para mínimos locais. O método de pesquisa em grade não é um desses algoritmos, pois ao verificar todos os valores de  $x$  ele sempre encontrará o valor de  $x$  que minimiza a função objetivo, porém nas próximas sessões serão apresentados métodos que, apesar de serem mais eficientes computacionalmente, podem tender para mínimos locais.

Ainda neste método, casos onde há uma maior quantidade de variáveis ( $x_1, x_2, \dots$ ) deve-se utilizar laços aninhados para que a varredura de todas as possibilidades possa ser

Figura 1 – Pesquisa em grade com número escalar



Fonte: Autor, adaptado de Haugen (2018)

feita.

Como exemplo, minimizemos a eq. (2.6), sendo  $0 \leq x_1 \leq 2$  e  $1 \leq x_2 \leq 3$ .

$$f(x) = (x_1 - 1)^2 + (x_2 - 2)^2 + 0,5 \quad (2.6)$$

Neste caso, de forma direta nota-se que  $f_{min} = 0,5$ ,  $x_{1_{opt}} = 1$  e  $x_{2_{opt}} = 2$ .<sup>1</sup> Porém se a restrição da eq. (2.7) for aplicada novos valores são encontrados, descritos nas equações presentes em eq. (2.8).

$$f(x) = x_1 - x_2 + 1,5 \leq 0 \quad (2.7)$$

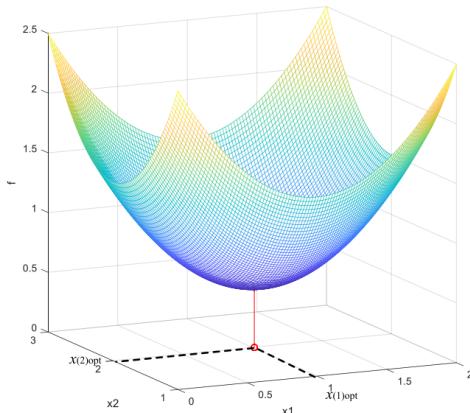
$$\begin{aligned} f_{min} &= 0,628 \\ x_{1_{opt}} &= 0,748 \\ x_{2_{opt}} &= 2,25 \end{aligned} \quad (2.8)$$

O motivo de os valores de  $x_{1_{opt}}$  e de  $x_{2_{opt}}$  serem diferentes quando a restrição da eq. (2.7) é aplicada pode ser visualmente observado nas figuras 2 e 3 onde elas mostram

<sup>1</sup> O código-fonte em Python para este cálculo pode ser encontrado no apêndice A, código-fonte A.2.

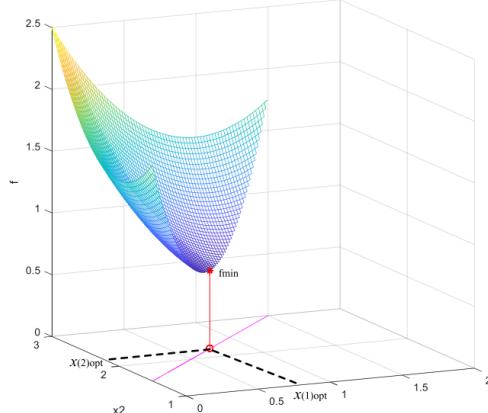
uma alteração do ponto mínimo da função custo (outra forma de chamarmos a função objetivo) devido à redução do conjunto imagem de  $f(x)$ .

Figura 2 – Pesquisa em grade com duas variáveis e sem restrição



Fonte: Haugen (2018)

Figura 3 – Pesquisa em grade com duas variáveis e com restrição



Fonte: Haugen (2018)

## 2.2.2 Método de busca de descidas mais íngremes

Tal qual o método de pesquisa em grade, a maioria dos outros algoritmos de otimização consiste em testar valores de  $x$  e indicar qual deles retorna o menor  $f(x)$ , porém, diferentemente do método anterior, a técnica apresentada nesta seção não testa todos os valores possíveis de  $x$ , na realidade ela calcula o próximo valor de  $x$  baseando-se na derivada da função custo calculada no ponto  $x$ .

Exemplificando para um caso escalar podemos dizer que o próximo valor de  $x$ , isto é,  $x_{k+1}$ , será dado por:

$$x_{k+1} = x_k + \Delta x_k \quad (2.9a)$$

$$\Delta x_k = -K f'(x_k) \quad (2.9b)$$

Onde:

$x_k$  =  $x$  atual

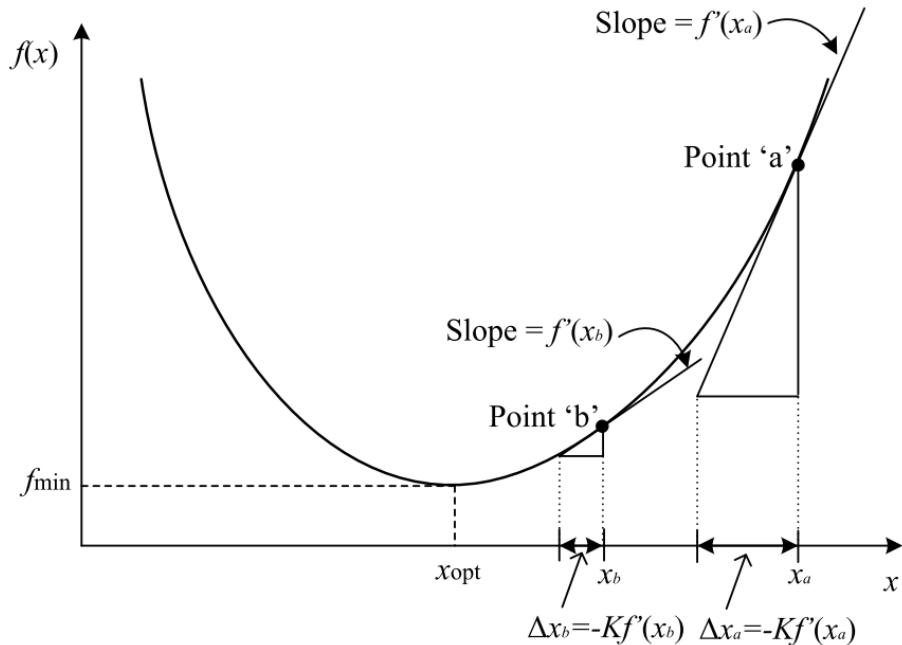
$\Delta x_k$  = diferença entre  $x_k$  e  $x_{k+1}$

$K$  = fator multiplicador do incremento

$f'(x_k)$  = derivada da função custo calculada em  $x_k$

A derivada  $f'(x_k)$  indica quão inclinada está a função custo no ponto  $x_k$ , assim o fator  $K$  determina o peso que essa inclinação terá para o cálculo do próximo valor de  $x$ .

Figura 4 – Influência de  $f'(x_k)$  no cálculo de  $x_{k+1}$  no método de descidas mais íngremes



Fonte: Haugen (2018)

A fig. 4 mostra graficamente a influência da derivada da função custo na amplitude de  $\Delta x_k$  entre iterações. Na figura vemos a derivada aplicada sobre dois pontos distintos,  $x_a$  e  $x_b$ , e podemos notar que  $f'(x_b)$  será menor que  $f'(x_a)$  uma vez que  $x_b$  está mais próximo ao ponto mínimo de  $f'$ .

Como indicado na eq. (2.9a), o cálculo de  $x_{k+1}$  depende de  $x_k$ , ou seja, o valor do próximo  $x$  calculado depende do  $x$  anterior. Para o cálculo de  $x_1$  é necessário fazer uma estimativa de  $x_0$ . A maior parte das funções de otimização requerem como argumento um valor de  $x_0$  estimado, pois é a partir dele que serão iniciadas as buscas até o ponto mínimo da função.

As mesmas equações se aplicam para casos onde  $x$  é um vetor, com a diferença que ao invés de calcularmos a derivada sobre o ponto  $x_k$ , calculamos o gradiente do vetor  $x_k$ , tal qual vemos na eq. (2.10b).

$$x_{k+1} = x_k + \Delta x_k \quad (2.10a)$$

$$\Delta x_k = -K\nabla f(x_k) \quad (2.10b)$$

Sendo que:

$$\mathbf{x}_k = \begin{bmatrix} x(1)_k \\ x(2)_k \\ \vdots \\ x(n)_k \end{bmatrix}$$

$K$  = fator multiplicador do incremento

$\nabla f(\mathbf{x}_k)$  = gradiente da função custo calculada em  $\mathbf{x}_k$

Além de requerer um valor estimado para  $x_0$ , este método também apresenta um critério de parada, isto é, uma condição que quando alcançada irá parar o laço de iterações por já ter encontrado o  $f_{min}$ . Este critério é dado pela eq. (2.11) e ao alcançá-la então  $x_{opt}$  será  $x_{k+1}$ .

$$|f(x_{k+1}) - f(x_k)| \leq df \quad (2.11)$$

Onde:

$df$  = Valor do critério de parada (ex. 0,00001)

O código-fonte 2.1 exemplifica a utilização deste método para as equações descritas em 2.12.

$$\min_x f(x) \quad (2.12a)$$

$$f(\mathbf{x}) = [x(1) - 1]^2 + [x(2) - 2]^2 + 0,5 \quad (2.12b)$$

$$|f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)| \leq df \quad (2.12c)$$

$$df = 10^{-4} \quad (2.12d)$$

Código-fonte 2.1 – Busca por descidas mais íngremes

---

```
# Linguagem:           Python
# Autor:              Tiago Correia Prata
# Disponível em:
# <https://github.com/TiagoPrata/MasterThesis/blob/master/4_codes/steepest_descent_
vectorial.py>

import numpy as np

# definindo a função custo descrita na eq. (2.12b)
```

```

def f_obj(x):
    return (x[0]-1)**2 + (x[1]-2)**2 + 0.5

x_guess = np.array([0., 1.]).T           # x0 estimado
x_k = x_guess
N = 10000      # limite de tentativas
abs_df = 1e-4    # valor do criterio de parada, como na eq. (2.12d)

for k in range(1, N-1):
    G_k = np.array([2*(x_k[0]-1), 2*(x_k[1]-2)]).T
    K = 0.1
    dx_k = -K * G_k
    x_kp1 = x_k + dx_k
    f_k = f_obj(x_k)
    f_kp1 = f_obj(x_kp1)

    # implementacao do criterio de parada da eq. (2.12c)
    df = f_kp1 - f_k
    x_k = x_kp1
    if abs(df) < abs_df:
        break

x_opt = x_kp1
f_min = f_obj(x_opt)

print(k)
print(x_opt)
print(f_min)

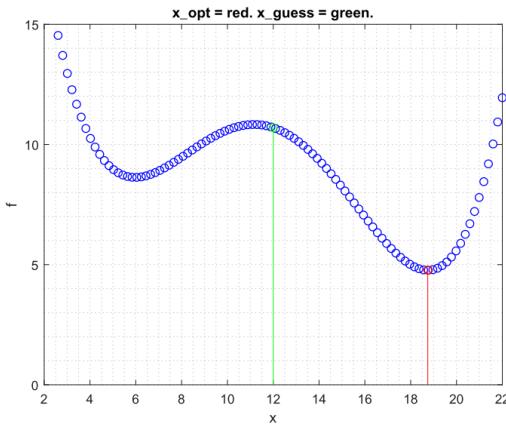
```

Fonte: Autor, adaptado de Haugen (2018)

Um problema conhecido deste método é o fato de ele poder não distinguir entre mínimos locais e mínimos globais fazendo com que o valor que é apontado como  $f_{min}$  possa ser na verdade apenas um mínimo local. O valor de  $x_0$  estimado tem impacto direto neste resultado, pois como este método inicia o cálculo da derivada de  $f(x)$  no ponto  $x_0$ , então as iterações seguintes dependerão exclusivamente da iteração anterior, fazendo assim uma convergência para o mínimo mais próximo, seja ele local ou global. As figs. 5 e 6 mostram mais detalhadamente esta diferença.

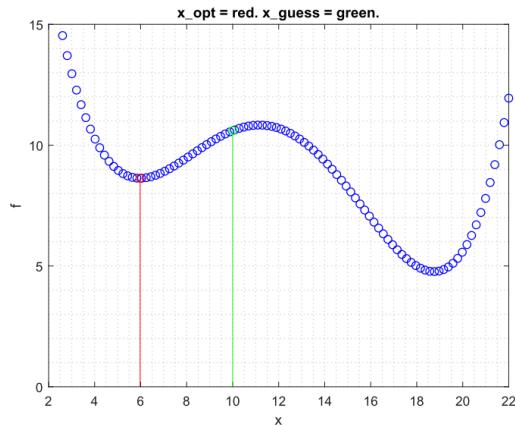
As figs. 5 e 6 apresentam a mesma função custo, porém na fig. 5  $x_0 = 12$ , e as derivadas consecutivas a partir desse valor convergem para o mínimo global; já na fig. 6 estimasse  $x_0 = 10$  resultando em um  $f_{min}$  completamente diferente, convergindo para um mínimo local. Em ambas as figuras a linha verde indica o valor de  $x_0$  e a linha vermelha indica o  $f_{min}$  quando o critério de parada é atingido.

Figura 5 – Método de descida mais íngrime convergindo para mínimo global



Fonte: Haugen (2018)

Figura 6 – Método de descida mais íngrime convergindo para mínimo local



Fonte: Haugen (2018)

### 2.2.3 Otimizadores de programação não-linear (NLP)

Muitos métodos de otimização não suportam restrições como as descritas nas eqs. (2.2) e (2.3), este é, por exemplo, o caso do método de busca de descidas mais íngremes, porém outros métodos podem levar esse tipo de restrição em conta, e estes são conhecidos como Otimizadores de Programação não-linear (do inglês *Nonlinear Programming*, ou simplesmente **NLP**) (HAUGEN, 2018).

Otimizadores **NLP** são úteis quando há a necessidade de métodos de otimização que sejam mais robustos e mais flexíveis do que técnicas mais simples, como o método de busca de Newton, ou aqueles apresentados nas sessões anteriores deste trabalho e por isso muitos softwares como o **MATLAB®** e bibliotecas da linguagem Python dispõem de ferramentas para cálculo de métodos **NLP**. No **MATLAB®** o *Optimization Toolbox™*<sup>2</sup> possui a função *fmincon* enquanto no Python podemos encontrar a função *scipy.optimize.minimize* na biblioteca *SciPy*<sup>3</sup>.

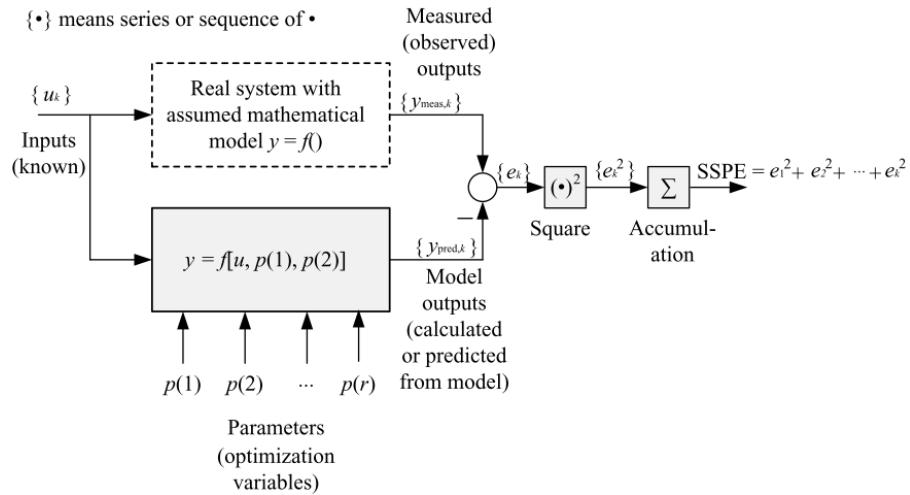
Os autores Haugen (2018), Ruggiero e Lopes (2000) apresentam em maiores detalhes os métodos descritos nesta seção, além de muitos outros.

A documentação referente as funções *fmincon* e *scipy.optimize.minimize* encontram-se, respectivamente, no anexo A, seções A.1 e A.2.

<sup>2</sup> O **MATLAB® Optimization Toolbox™** é um conjunto de ferramentas de otimização que provê funções para encontrar parâmetros que minimizem ou maximizem objetivos, satisfazendo as condições limites. Mais detalhes em [www.mathworks.com/products/optimization.html](http://www.mathworks.com/products/optimization.html)

<sup>3</sup> A biblioteca *SciPy* (do inglês *SciPy Library*) faz parte do conjunto de soluções do programa *SciPy*. A biblioteca *SciPy* fornece uma interface de usuário amigável para cálculos de integração numérica, interpolação, otimização, álgebra linear, estatística, entre outros. Mais detalhes em [www.scipy.org/scipylib/index.html](http://www.scipy.org/scipylib/index.html)

Figura 7 – Diagrama de blocos do quadrado da diferença entre o sistema real e o modelo



Fonte: Haugen (2018)

## 2.3 Algumas aplicações de otimização

### 2.3.1 Estimação de parâmetros

Estimar parâmetros de um modelo pode ser uma das inúmeras aplicações dos métodos de otimização. Segundo Haugen (2018) um problema de estimação de parâmetros pode ser formulado como mostrado na eq. (2.13), visando encontrar os parâmetros que apresentem menor diferença entre o sistema real e o modelo sendo avaliado. A comparação o sistema real e o virtual é feita através do acúmulo da diferença das saídas dos dois sistemas elevada ao quadrado ao longo de  $N$  iterações.

$$\min_P \sum_{k=1}^N e(k)^2 \quad (2.13)$$

Onde:

$P$  = vetor de parâmetros a ser estimado. Sendo  $P = [p(1), p(2), \dots, p(r)]^T$

$e$  = erro de predição. Sendo  $e = y_{medido} - y_{modelo}$

$y_{medido}$  = medição da saída do sistema real

$y_{modelo}$  = cálculo da saída do modelo simulado

$N$  = quantidade de amostras

O diagrama de blocos apresentado na fig. 7 apresenta de forma visual a comparação entre o sistema real e o modelo matemático.

Para estimar os parâmetros  $P$  de um dado modelo é necessário dividir os dados amostrais  $N$  em dois grupos: um deles destinado a adaptação do modelo, ou seja, encontrar

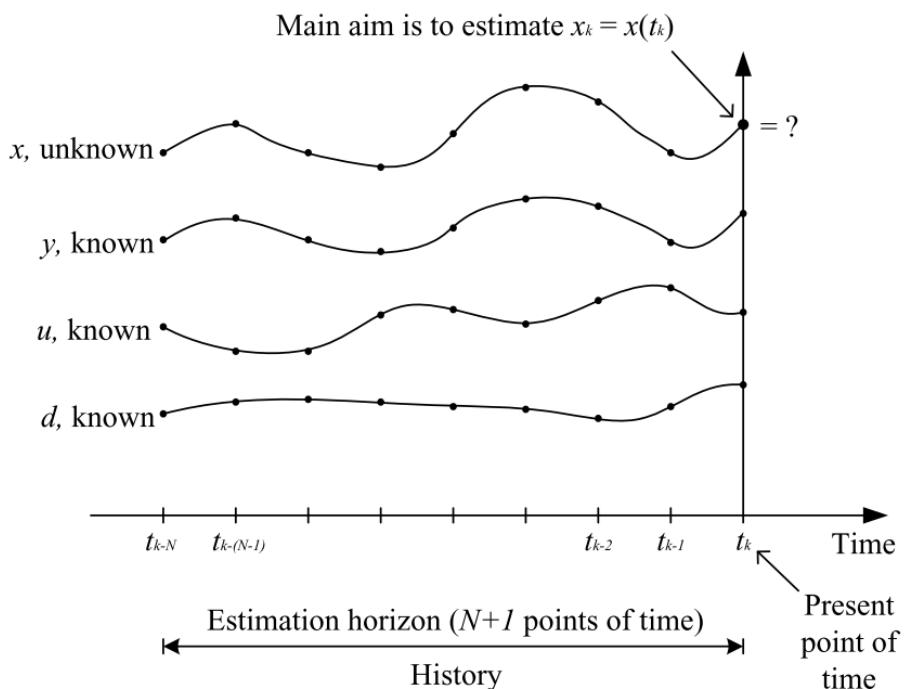
o vetor  $P$  que minimize a função custo, e o outro grupo destinado a validação, isto é, verificar se os parâmetros encontrados durante a adaptação realmente é válido e preciso.

Quando dispomos de dados previamente coletados do sistema é possível fazer uma *estimação em batelada* (também conhecida como *offline* (BOLOGNANI et al., 2009)), onde as fases de coleta e estimação acontecem em momentos distintos. Já a *estimação recursiva* (ou *online* (STADLER; POLAND; GALLESTEY, 2011)) ocorre quando a coleta dos dados e a estimação dos parâmetros acontece ao mesmo tempo. A [seção 2.3.2](#) descreve com mais detalhes a **Estimador de Horizonte Móvel** (do inglês, *Moving Horizon Estimation*, ou simplesmente **MHE**), que, assim como o filtro de Kalman, pode ser utilizado como estimador de parâmetros recursivo.

### 2.3.2 Estimador de Horizonte Móvel (MHE)

O método conhecido como **Estimador de Horizonte Móvel** é um método estimador recursivo de parâmetros que utiliza as medições atuais, anteriores e também as entradas do sistema sob um horizonte fixo de tempo para calcular o estado atual do sistema, assim como ilustrado na [fig. 8](#) (HAUGEN, 2018).

Figura 8 – Horizonte de tempo analisado pelo método **Estimador de Horizonte Móvel**



Fonte: Haugen (2018)

O modelo de espaço de estado que descreve o sistema pode ser descrito como indicado na [eq. \(2.14\)](#).

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \dots) + \mathbf{w}_k \quad (2.14a)$$

$$\mathbf{y}_k = \mathbf{g}(\mathbf{x}_k, \dots) + \mathbf{v}_k \quad (2.14b)$$

Onde:

- $\mathbf{x}$  é o vetor de estado a ser estimado.

$$\mathbf{x}_k = \begin{bmatrix} x(1)_k \\ x(2)_k \\ \vdots \\ x(n)_k \end{bmatrix} \quad (2.15)$$

- $\mathbf{f}$  é um vetor de  $n$  funções lineares ou não-lineares de  $x_k$

$$\mathbf{f} = \begin{bmatrix} f_1(x_k, \dots) \\ f_2(x_k, \dots) \\ \vdots \\ f_n(x_k, \dots) \end{bmatrix} \quad (2.16)$$

Os pontos representam possíveis argumentos adicionais de  $f$  como a variável de controle ( $u_k$ ), os distúrbios do processo ( $d_k$ ), e os parâmetros ( $p$ ).

- $\mathbf{w}_k$  representa os distúrbios não conhecidos ou não modelados do processo agindo sobre o estado, no instante  $k$ . Não é necessário assumir nenhuma propriedade estatística particular para este distúrbio, porém é sensato assumir um ruído, como o ruído branco, com uma matriz de covariância  $Q$ , assim como utilizado no filtro de Kalman. A matriz  $Q$  será apresentada com mais detalhes a seguir.
- $\mathbf{y}$  é o vetor de saída com  $m$  elementos.
- $\mathbf{g}$  é um vetor de  $m$  funções lineares ou não-lineares de  $\mathbf{x}_k$ , onde os pontos também representam possíveis argumentos adicionais.
- $\mathbf{v}_k$  representa o erro medido, e, assim como  $\mathbf{w}_k$ , não necessita assumir nenhuma propriedade estatística particular, porém também é comum utilizar um ruído branco, com uma matriz de covariância  $R$ , também melhor apresentada a seguir.

O problema de otimização do MHE pode ser descrito como na eq. (2.17).

$$\min_{\mathbf{X}} \sum_{i=k-N}^{k-1} \|x_{i+1} - f(x_i, \dots)\|_{Q^{-1}}^2 + \sum_{i=k-N}^k \|y_i - g(x_i, \dots)\|_{R^{-1}}^2 \quad (2.17)$$

Sendo que  $X$  é a matriz contendo o estado a cada ponto do horizonte de estimação:

$$\begin{aligned} X &= [x_{k-N}, x_{k-N-1}, \dots, x_{k-1}, x_k] \\ &= \begin{bmatrix} x(1)_{k-N} & x(1)_{k-(n-1)} & \cdots & x(1)_{k-1} & x(1)_k \\ x(2)_{k-N} & x(2)_{k-(n-1)} & \cdots & x(2)_{k-1} & x(2)_k \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x(n)_{k-N} & x(n)_{k-(n-1)} & \cdots & x(n)_{k-1} & x(n)_k \end{bmatrix} \end{aligned} \quad (2.18)$$

E com base na eq. (2.14) este problema também pode ser descrito como:

$$\min_X \sum_{i=k-N}^{k-1} \|w_i\|_{Q^{-1}}^2 + \sum_{i=k-N}^k \|v_i\|_{R^{-1}}^2 \quad (2.19)$$

Ou mesmo:

$$\min_X \sum_{i=k-N}^{k-1} w_i^T Q^{-1} w_i + \sum_{i=k-N}^k v_i^T R^{-1} v_i \quad (2.20)$$

Onde:

- $w_i$  é o vetor de distúrbio do processo.

$$w_i = \begin{bmatrix} w(1)_i \\ w(2)_i \\ \vdots \\ w(n)_i \end{bmatrix} \quad (2.21)$$

- $v_i$  é um vetor de erros de medição.

$$v_i = \begin{bmatrix} v(1)_i \\ v(2)_i \\ \vdots \\ v(m)_i \end{bmatrix} \quad (2.22)$$

- $Q^{-1}$  é a matriz de ponderação de  $w_i$

$$Q^{-1} = \begin{bmatrix} \frac{1}{Q_{11}} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{Q_{nn}} \end{bmatrix} \quad (2.23)$$

Assumindo que  $w$  é aleatório, então  $Q$  pode ser interpretado como a inversa da matriz de covariância do distúrbio do processo.

- $R^{-1}$  é a matriz de ponderação de  $v_i$

$$R^{-1} = \begin{bmatrix} \frac{1}{R_{11}} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{R_{nn}} \end{bmatrix} \quad (2.24)$$

Assumindo que  $v$  é aleatório, então  $R$  pode ser interpretado como a inversa da matriz de covariância do distúrbio do processo.

Desta forma a eq. (2.20) pode ser reescrita como:

$$\min_X \sum_{i=k-N}^{k-1} \left[ \frac{w(1)_i^2}{Q_{11}} + \cdots + \frac{w(n)_i^2}{Q_{nn}} \right] + \sum_{i=k-N}^k \left[ \frac{v(1)_i^2}{R_{11}} + \cdots + \frac{v(n)_i^2}{R_{mm}} \right] \quad (2.25)$$

Resumidamente observa-se que o MHE utiliza as medições (minimizando a influência dos erros de medição) e o modelo (minimizando os erros de modelo) para calcular o estado estimado.

### 2.3.2.1 Exemplo

A fig. 9 mostra o gráfico de uma simulação executada através do código-fonte A.3 exemplificando a aplicação da técnica Estimador de Horizonte Móvel para o modelo simplificado da velocidade de um motor de corrente contínua.

O modelo dinâmico do motor é apresentado na eq. (2.26) e sua representação no espaço de estados na eq. (2.27), sendo que esta é a representação utilizada no código-fonte A.3.

$$T\dot{S} = -S + Ku + L \quad (2.26)$$

Onde:

$u$  = sinal de controle (tensão aplicada ao motor)

$S$  = sinal medido (velocidade do motor)

$K$  = ganho

$T$  = constante de tempo

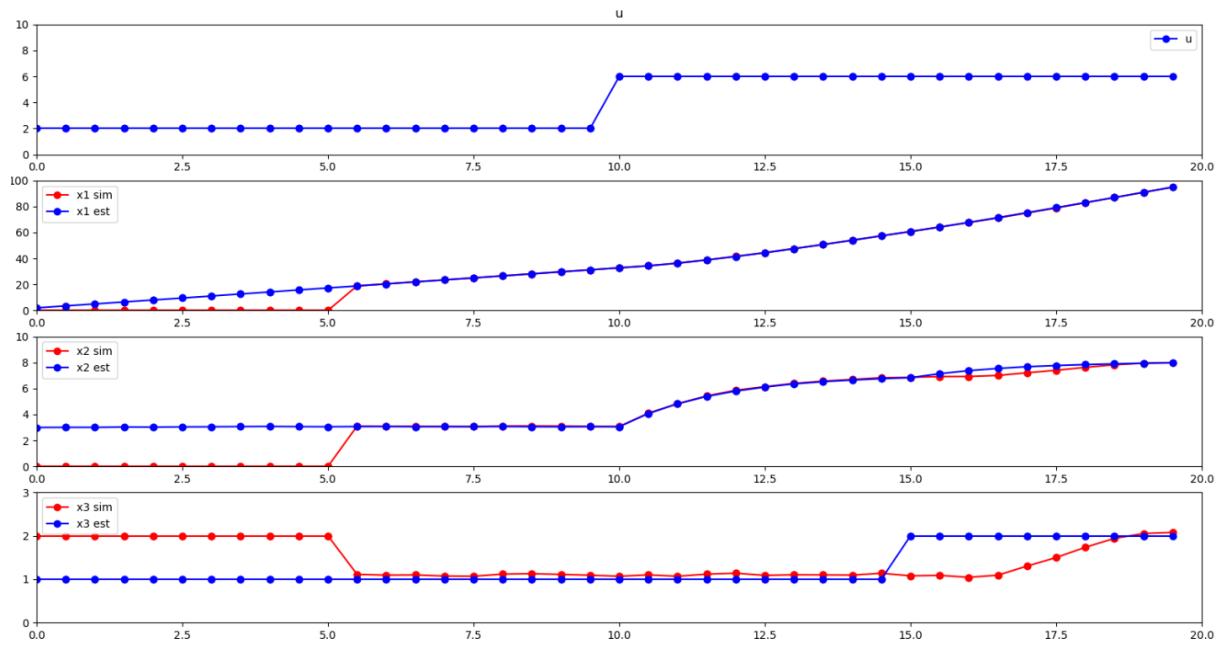
$L$  = carga

$$\dot{x}(1) = x(2) \quad (2.27a)$$

$$\dot{x}(2) = [-x(2) + Ku]/T + d \quad (2.27b)$$

$$y = x(1) \quad (2.27c)$$

Figura 9 – Simulação utilizando Estimador de Horizonte Móvel

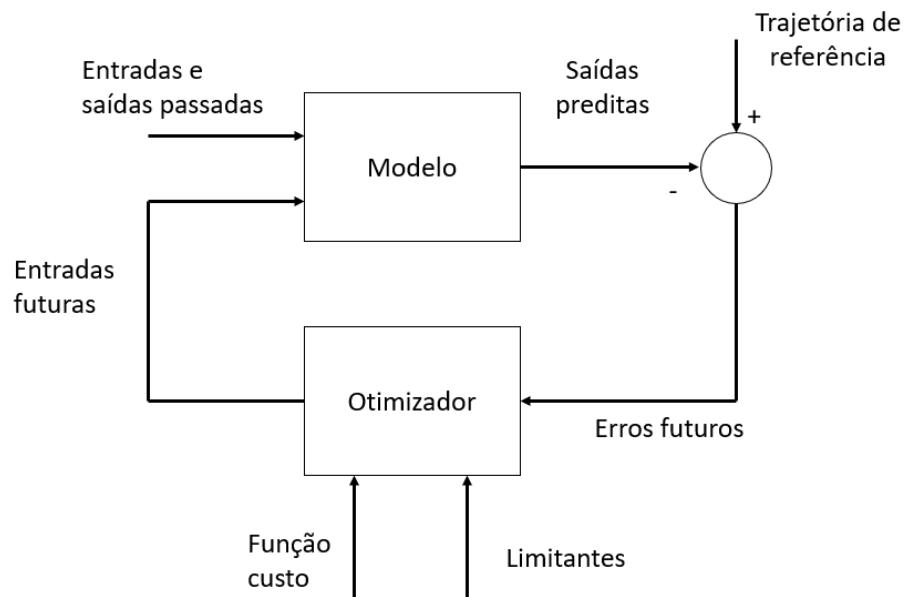


Fonte: Autor, adaptado de Haugen (2018)

### 3 Controle Preditivo Baseado em Modelo

Segundo Rossiter (2003) uma das principais diferenças entre o MPC e outras técnicas de controle (como o PID, por exemplo) é que a maioria delas não leva em consideração as ações futuras do controle, porém o MPC sim, e para essas ações futuras sejam consideradas é necessário possuir um modelo do sistema que mostre as dependências das saídas e das atuais variáveis medidas e as entradas atuais e futuras. Este modelo não precisa ser linear, e nem tampouco ser extremamente fiel em descrever todas as interações físico-químicas do sistema. Na verdade, ainda segundo Rossiter (2003), a regra básica do modelo é que ele deve ser o mais simples possível, ou seja, deve ser o modelamento mínimo necessário para que possamos observar previsões com o nível de precisão necessário.

Figura 10 – Estrutura básica do MPC



Fonte: Autor, adaptado de Camacho, Alba e Bordons (2007)

Além de comparar com outros métodos de controle e de descrever os modelos utilizados no MPC, Rossiter (2003) também salienta alguns outros parâmetros e características que são importantes para a técnica:

- A **seleção das entradas** do sistema deve ser feita considerando a função custo que se deseja minimizar.
- O intervalo de tempo futuro, no qual o MPC fará o cálculo das previsões de controle (também conhecido como **horizonte retrocedente** (*Receding Horizon*) deve incluir

todas as dinâmicas significativas para o sistema, caso contrário o controle apresentará um desempenho ruim e alguns eventos não poderão ser observados.

- O controle será tão preciso quanto seu modelo permitir. Para conseguir um controle mais preciso é necessário também possuir um modelo mais preciso.
- Um dos principais pontos do MPC é a capacidade de **lidar com restrições** on-line de maneira sistemática, permitindo assim uma melhor performance.
- Ao utilizar um modelo, este método calcula a otimização do custo levando em conta as **mudanças futuras na trajetória** desejada e distúrbios mensuráveis.
- Outra característica extremamente importante do MPC é sua capacidade intrínseca de controlar sistemas MIMO.

### 3.1 Aplicação prática

Segundo Haugen (2018) tanto o MPC quanto o MHE são problemas praticamente idênticos do ponto de vista matemático, pois ambos exploram um modelo matemático sobre dado um horizonte de tempo, porém, apesar de matematicamente parecidos, eles atuam de forma inversa um ao outro, pois enquanto o MHE utiliza as variáveis de controle e os valores medidos do processo para estimar as variáveis estados (como já apresentado na seção 2.3.2), o MPC utiliza as variáveis de estado e os valores medidos do processo para estimar as variáveis de controle. Na fig. 11 é possível observar o MHE atuando sob um horizonte de estimativa, utilizando dados passados, e o MPC atuando sob um horizonte de predição, predizendo os valores futuros.

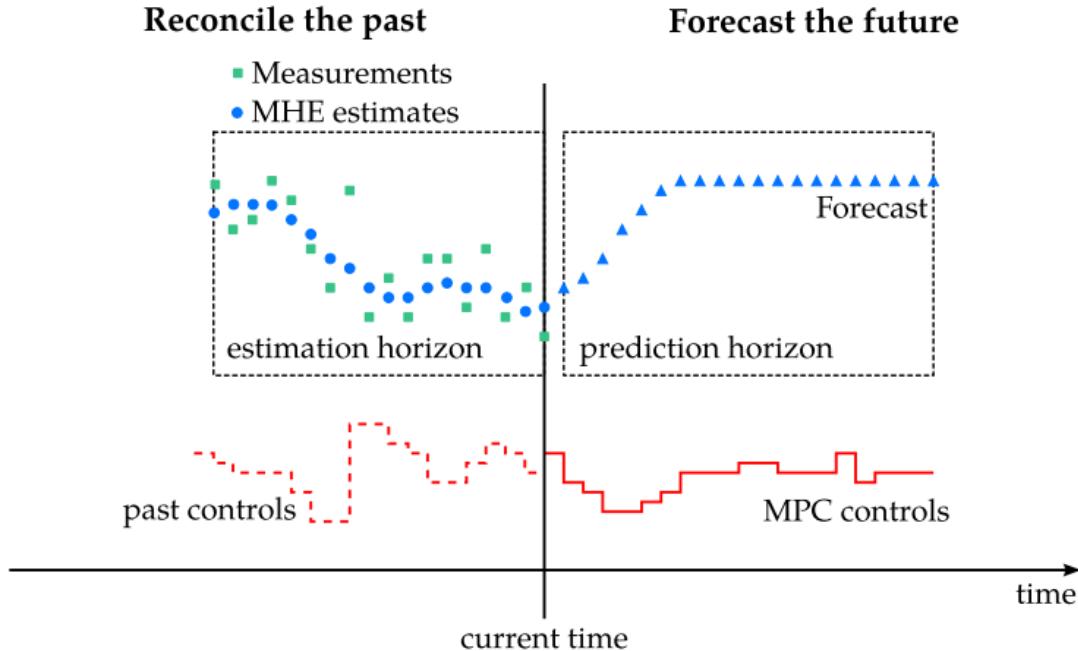
Podemos descrever o MPC através de um problema de otimização de forma análoga ao que foi feito na seção 2.3.2 para descrever o MHE, porém, ao invés minimizarmos as variáveis de estado, devemos encontrar os valores ótimos (minimizar) das variáveis de controle. As eqs. (3.1) a (3.9) descrevem este problema de otimização.

$$\min_U \sum_{i=k}^{k+N} (\|e_i\|_{C_e}^2 + \|du\|_{C_{du}}^2) \quad (3.1)$$

Onde:

- U é uma matriz contendo  $r$  variáveis de controle nos instantes  $k$  do horizonte de

Figura 11 – Ação do Estimador de Horizonte Móvel e do Controle Preditivo Baseado em Modelo



Fonte: Vukov (2015)

predição.

$$\begin{aligned} \mathbf{U} &= [\mathbf{u}_k, \mathbf{u}_{k+1}, \dots, \mathbf{u}_{k+(N-1)}, \mathbf{u}_N] \\ &= \begin{bmatrix} u(1)_k & u(1)_{k+1} & \cdots & u(1)_{k+(N-1)} & u(1)_N \\ u(2)_k & u(2)_{k+1} & \cdots & u(2)_{k+(N-1)} & u(2)_N \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ u(n)_k & u(n)_{k+1} & \cdots & u(n)_{k+(N-1)} & u(n)_N \end{bmatrix} \end{aligned} \quad (3.2)$$

- $\mathbf{e}_i$  é o vetor de erro de controle

$$\mathbf{e}_i = \begin{bmatrix} e(1)_i \\ e(2)_i \\ \vdots \\ e(m)_i \end{bmatrix} \quad (3.3)$$

Onde  $e(j)_i$  é o erro de controle da saída  $j$  do processo, no instante de tempo  $i$ , sendo que:

$$e(j)_i = y(j)_{sp_i} - y(j)_i \quad (3.4)$$

- $\text{du}_i$  é o vetor de incremento da variável de controle

$$\text{du}_i = \begin{bmatrix} du(1)_i \\ du(2)_i \\ \vdots \\ du(r)_i \end{bmatrix} \quad (3.5)$$

Onde  $du(j)_i$  é o incremento da variável de controle relativo a esta mesma variável no instante de tempo anterior:

$$du(j)_i = u(j)_i - u(j)_{i-1} \quad (3.6)$$

- As matrizes  $C_e$  e  $C_{du}$  são matrizes de custo (peso) e são utilizadas como elementos de sintonia.

$$C_e = \begin{bmatrix} C_e(1, 1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & C_e(m, m) \end{bmatrix} \quad (3.7a)$$

$$C_{du} = \begin{bmatrix} C_{du}(1, 1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & C_{du}(r, r) \end{bmatrix} \quad (3.7b)$$

Substituindo as normas matriciais da eq. (3.1), temos:

$$\min_U \sum_{i=k}^{k+N} (e_i^T C_e e_i + du_i^T C_{du} du_i) \quad (3.8)$$

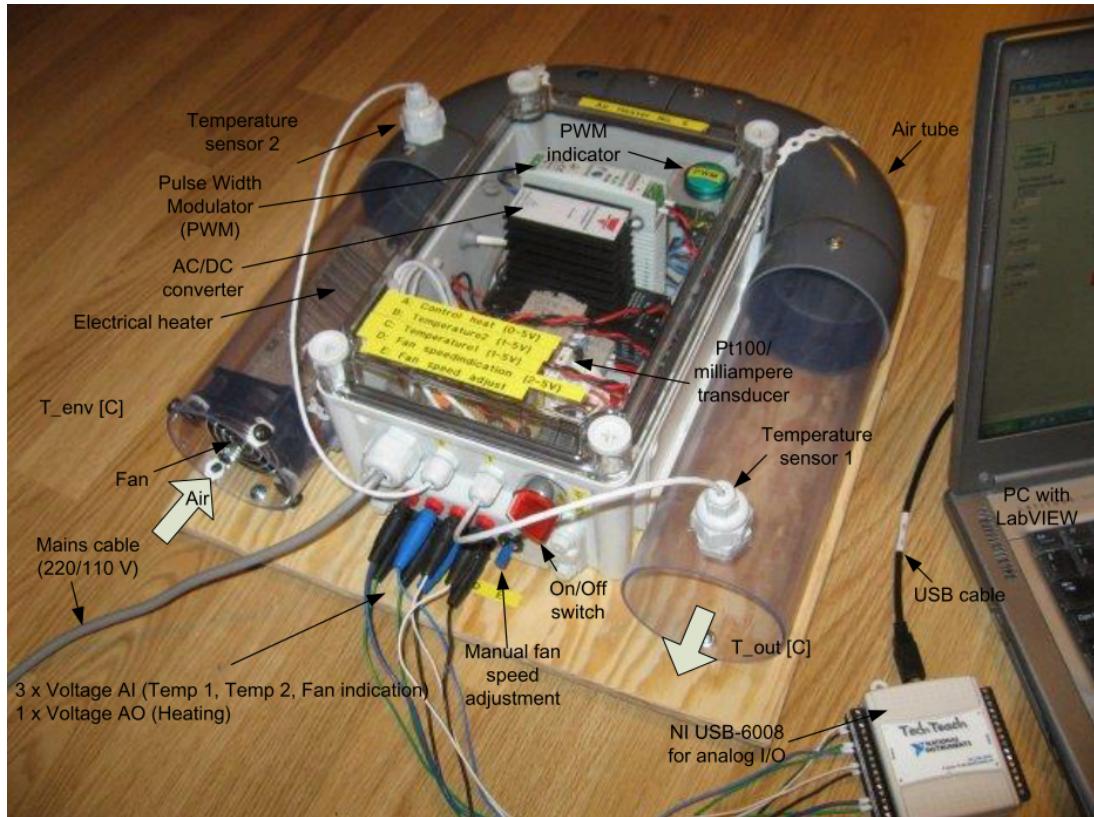
Expandindo então as eqs. (3.2) a (3.7) na eq. (3.8), obtemos:

$$\min_U \sum_{i=k}^{k+N} [C_e(1, 1)e(1)_i^2 + \cdots + C_e(m, m)e(m)_i^2] + [C_{du}(1, 1)du(1)_i^2 + \cdots + C_{du}(r, r)du(r)_i^2] \quad (3.9)$$

### 3.1.1 Exemplo

O exemplo descrito nesta seção apresenta a simulação do modelo de uma planta didática de aquecimento de ar utilizado na *University College of Southeast Norway*, situada em Porsgrunn, Noruega. Vale salientar que esta não é a planta objeto de estudo deste trabalho, sendo este apenas um exemplo bibliográfico de aplicação de MPC em planta

Figura 12 – Planta de aquecimento de ar



Fonte: Haugen (2018)

didática, cujo modelo matemático está indicado na eq. (3.10) e seu detalhamento construtivo pode ser melhor compreendido na fig. 12.

$$\theta_t \dot{T}_{heat}(t) = -T_{heat}(t) + K_h[u(t - \theta_d) + d] \quad (3.10a)$$

$$T_{out}(t) = T_{heat}(t) + T_{env} \quad (3.10b)$$

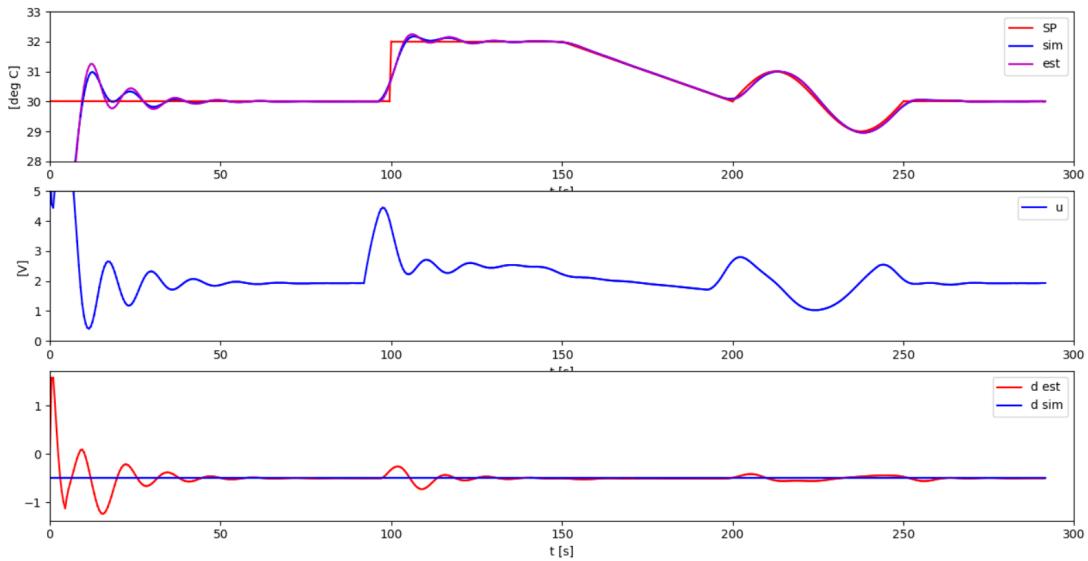
Onde:

- $K_h$  é o ganho do aquecedor
- $T_{env}$  é a temperatura ambiente
- $T_{heat}$  é a contribuição para a temperatura total  $T_{out}$  devido ao aquecedor
- $T_{out}$  é a temperatura do ar saindo da planta. Medido através de sensor
- $u$  é o sinal de controle do aquecedor
- $d$  é o distúrbio de entrada (adicionado ao sinal de controle)

- $\theta_d$  é o atraso que representa o transporte de ar e a dinâmica do aquecedor
- $\theta_t$  é o atraso referente à dinâmica do aquecedor

A fig. 13 ilustra os dados obtidos através da execução do código-fonte A.4, no apêndice A, escrito em Python.

Figura 13 – Simulação utilizando Controle Preditivo Baseado em Modelo



Fonte: Autor, adaptado de Haugen (2018)

# Parte III

## Materiais e métodos



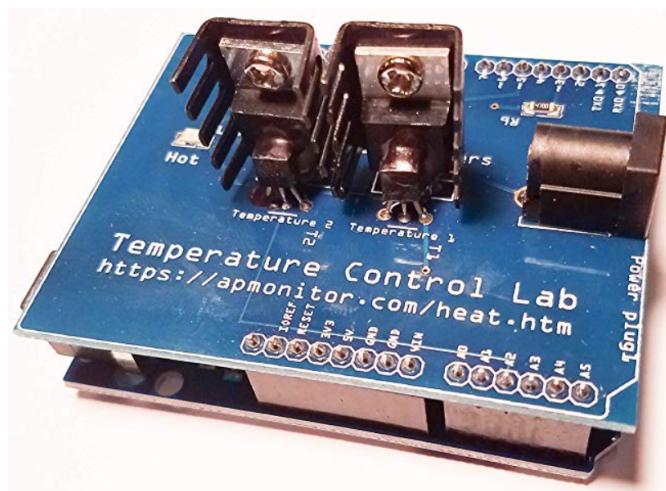
## 4 Planta piloto

O sistema experimental usado para o projeto de ações de controle é uma mini planta piloto chamada *Temperature Control Lab* (do inglês, Laboratório de Controle de Temperatura), ou apenas **TCLab**<sup>1</sup>. Esta planta foi desenvolvida na *Brigham Young University* (BYU) e apresentada pela primeira vez no *2017 ASEE Summer School*, na *North Carolina State University* (NCSU). Foi desenvolvida com o propósito de facilitar o acesso de estudantes a um laboratório de testes de controle.

Esta mini planta é essencialmente um *shield* para Arduino<sup>2</sup> contendo 2 aquecedores e 2 sensores de temperatura, mostrado na fig. 14. A energia dos aquecedores é transferida através de condução, convecção e radiação até os sensores de temperatura. Tanto o controle da potência dos aquecedores quanto as medições realizadas pelos sensores são efetuados através do Arduino.

O **TCLab** possibilita fazer a programação do controle da mini planta utilizando linguagem Python, **MATLAB®** ou Simulink.

Figura 14 – Laboratório de Controle de Temperatura



Fonte: Hedengren (201-?b)

<sup>1</sup> Maiores informações sobre o **TCLab** podem ser encontrados em <http://apmonitor.com/heat.htm>.

<sup>2</sup> Arduino é uma plataforma de prototipagem eletrônica de hardware livre e de placa única, projetada com um microcontrolador Atmel AVR com suporte de entrada/saída embutido.

## 4.1 Modelagem da planta piloto

A modelagem teórica de balanço energético desta planta piloto é fornecida pela equipe responsável pelo **TCLab**. São fornecidos dois modelos distintos: um para o uso de apenas um aquecedor e um sensor, sendo assim um sistema de apenas uma entrada e uma saída, ou seja, um sistema **SISO** (do inglês *Single Input Single Output*), e outro utilizando ambos os aquecedores e sensores condidos na planta, onde devido a proximidade existe a interferência de um no outro, sendo assim um sistema **MIMO**. Ambos os modelos teóricos são apresentados nas seções 4.1.1 e 4.1.2 a seguir.

### 4.1.1 Modelo SISO

Segundo a equipe de desenvolvimento do **TCLab**, para a utilização de apenas um aquecedor e um sensor é possível utilizar a eq. (4.1), sendo que, para isso, deve-se assumir que: o aquecedor e o sensor de temperatura estão na mesma temperatura; que o efeito da condução de calor é desprezado e que as únicas trocas de calor acontecem através de radiação ou convecção; que o aquecedor inicialmente está desligado e que tanto o aquecedor quanto o sensor inicialmente estão em temperatura ambiente.

$$mC_p \frac{dT}{dt} = UA(T_\infty - T) + \epsilon\sigma A(T_\infty^4 - T^4) + Q \quad (4.1)$$

A **tabela 1** indica a descrição de cada uma das siglas da eq. (4.1) e também apresenta o valor de cada um.

Tabela 1 – Valores para modelagem SISO do **TCLab**

Sigla	Descrição	Valor	Valor (SI)
$T_0$	Temperatura inicial	23°C	296,15K
$T_\infty$	Temperatura ambiente	23°C	296,15K
$Q$	Saída do aquecedor	0 à 3W	0 à 3W
$C_p$	Capacidade de aquecimento	500 J/kg.K	500 J/kg.K
$A$	Área de superfície	12cm <sup>2</sup>	1,2 × 10 <sup>-3</sup> m <sup>2</sup>
$m$	Massa	4g	0,004kg
$U$	Coeficiente global de transf. de calor	10 W/m <sup>2</sup> K	10 W/m <sup>2</sup> K
$\epsilon$	Emissividade	0,9	0,9
$\sigma$	Constante de Stefan Boltzmann	5,67 × 10 <sup>-8</sup> W/m <sup>2</sup> K <sup>4</sup>	5,67 × 10 <sup>-8</sup> W/m <sup>2</sup> K <sup>4</sup>

Fonte: Autor, adaptado de **Hedengren (201-?)a**

### 4.1.2 Modelo MIMO

As eqs. (4.2a) e (4.2b) apresentam os modelos para um sistema **MIMO**, onde ambos os aquecedores e sensores são utilizados. As mesmas premissas descritas para o modelo

SISO na seção 4.1.1 devem ser aplicadas neste modelo.

$$mC_p \frac{dT_1}{dt} = UA(T_\infty - T_1) + \epsilon\sigma A(T_\infty^4 - T_1^4) + UA_s(T_2 - T_1) + \epsilon\sigma A_s(T_2^4 - T_1^4) + Q_1 \quad (4.2a)$$

$$mC_p \frac{dT_2}{dt} = UA(T_\infty - T_2) + \epsilon\sigma A(T_\infty^4 - T_2^4) + UA_s(T_1 - T_2) + \epsilon\sigma A_s(T_1^4 - T_2^4) + Q_1 \quad (4.2b)$$

A tabela 2 indica a descrição de cada uma das siglas das eqs. (4.2a) e (4.2b) e também apresenta o valor de cada um.

Tabela 2 – Valores para modelagem MIMO do TCLab

Sigla	Descrição	Valor	Valor (SI)
$T_0$	Temperatura inicial	$23^\circ C$	$296,15K$
$T_\infty$	Temperatura ambiente	$23^\circ C$	$296,15K$
$Q$	Saída do aquecedor	$0 \text{ à } 3W$	$0 \text{ à } 3W$
$C_p$	Capacidade de aquecimento	$500 \text{ J/kg.K}$	$500 \text{ J/kg.K}$
$A$	Área de superfície	$10cm^2$	$1 \times 10^{-3}m^2$
$A_s$	Área de superfície (entre dissipadores)	$2cm^2$	$2 \times 10^{-4}m^2$
$m$	Massa	$4g$	$0,004kg$
$U$	Coeficiente global de transf. de calor	$10 \text{ W/m}^2 K$	$10 \text{ W/m}^2 K$
$\epsilon$	Emissividade	$0,9$	$0,9$
$\sigma$	Constante de Stefan Boltzmann	$5,67 \times 10^{-8} \text{ W/m}^2 K^4$	$5,67 \times 10^{-8} \text{ W/m}^2 K^4$

Fonte: Autor, adaptado de [Hedengren \(201-?b\)](#)



# 5 Metodologia

## 5.1 Modelagem experimental

Além dos modelos teóricos descritos no capítulo anterior, neste projeto também será executada a identificação experimental do sistema.

Segundo Gevers (2006) a teoria de identificação de sistemas data da década de 60 e as duas principais técnicas utilizadas na identificação de sistemas atualmente (método de identificação por subespaço de estados e método do erro de predição) tiveram suas bases estruturadas com os trabalhos de Ho e Kalman (1966 apud GEVERS, 2006) e de Åström e Torsten (1965 apud GEVERS, 2006).

De acordo com Dunia, Edgar e Haugen (2008 apud PRACEK et al., 2011) a identificação do sistema permite que simulações sejam desenvolvidas de modo a garantir um melhor desempenho dos sistemas de controle projetados.

As etapas para a identificação de um sistema, segundo Aguirre (2015) são:

- Testes dinâmicos e coleta de dados
- Escolha da representação matemática a ser utilizadas
- Determinação da estrutura do modelos
- Estimação de parâmetros
- Validação do modelo

As seções a seguir detalham cada uma dessas etapas.

### 5.1.1 Testes dinâmicos e coleta de dados

Esta etapa abrange os procedimentos necessários para geração do conjunto de dados que serão utilizados para a identificação do sistema. Algumas das atividades realizadas nessa etapa são: escolha das variáveis, definição dos sinais de excitação, definição do período de amostragem e a execução em si dos testes. (AGUIRRE, 2015)

- Período de amostragem: três regras práticas auxiliam na definição deste período: usar um tempo de amostragem de aproximadamente 1/10 da maior constante de tempo (GUSTAVSSON, 1975 apud BALLIN, 2008); 10% do tempo de acomodação de uma resposta degrau (BALLIN, 2008); ou escolher um tempo de amostragem

de 5 a 10 vezes maior do que a maior frequência de interesse contida nos dados. (AGUIRRE, 2015).

- Sinais de excitação: a escolha dos sinais de entrada pode ter grande impacto nos dados coletados, pois determinarão o ponto de operação do sistema e quais das suas características serão excitadas durante o experimento (AGUIRRE, 2015). Diversos tipos distintos de sinais podem ser utilizados. Entre os mais utilizados destacam-se: impulsivo; degrau; PRBS - Sinal Binário Pseudo-Aleatório (do inglês, *Pseudo-Random Binary Signal*); ARMA - Média Móvel Auto-Regressiva (do inglês, *Auto-Regressive Moving Average*); GBN - Ruido Binário Generalizado (do inglês, *Generalized Binary Noise*); soma de senóides e etc. (AGUIRRE, 2015)
- Duração do experimento: a duração do experimento, segundo Garcia (2005), deveria ser a maior possível, uma vez que a variância das estimativas é proporcional ao inverso da duração do experimento, porém sob o ponto de vista prático e experimental, a duração do experimento deveria ser a menor possível para a obtenção de um modelo aceitável, pois ao longo do experimento o processo estará sujeito a perturbações extras que podem impactar na operação da planta, na qualidade dos produtos ou até na segurança do processo.

### 5.1.2 Escolha da representação matemática

Existem diversas representações matemáticas distintas para modelos lineares, sendo que, segundo Aguirre (2015), a mais utilizada é a função de transferência, porém Wang (2009) destaca que nos últimos anos tem se observado um aumento na popularidade dos modelos de espaço de estados para desenvolvimento de controle preditivo. Para Aguirre (2015) é importante ainda salientar o modelo AR (modelo autorregressivo), o modelo ARX (modelo autorregressivo com entrada exógena) e o modelo ARMAX (modelo autorregressivo com média móvel e entrada exógena).

[...] quando se trata da modelagem obtida por meios fenomenológicos é comum que se adote a base de tempo contínuo, em virtude de a maioria das leis da física serem expressas nesse tempo. Por sua vez, quando se trata de identificação de sistemas por processos experimentais, trabalha-se com amostras de dados coletados a cada intervalo de tempo, nesses casos usualmente adota-se o tempo discreto. (GARCIA, 2005 apud FAVARO, 2012)

### 5.1.3 Determinação da estrutura do modelos

Determinar a ordem de um modelo é um dos aspectos mais importantes na determinação de sua estrutura, uma vez que, caso sua ordem seja muito menor do que a ordem efetiva do sistema real, o modelo não refletirá a completamente sua complexidade

estrutural. Analogamente, escolher um modelo que a ordem seja muito maior do que a necessária, provavelmente causará uma estimativa de parâmetros mal condicionada. (AGUIRRE, 2015)

#### 5.1.4 Estimação de parâmetros

A estimativa de parâmetros, segundo Eykhoff (1974 apud FAVARO, 2012) é a determinação experimental de valores de parâmetros que governam a dinâmica e/ou o comportamento não-linear, assumindo que a estrutura do modelo seja conhecida.

Essa etapa começa com a escolha do algoritmo a ser utilizado (AGUIRRE, 2015). Dentre eles, os mais amplamente empregados na literatura são: método da análise de frequência; método da resposta transitória e método dos mínimos quadrados (FAVARO, 2012).

#### 5.1.5 Validação do modelo

Em problemas de validação, a questão é tentar determinar se um dado modelo é válido ou não e para isso, deve-se simulá-lo sem qualquer ajuste adicional e compará-lo a dados medidos em testes diferentes daquele usado no desenvolvimento da sintonia do mesmo. Ao se obter um conjunto de modelos, deve-se verificar se eles incorporaram as características de interesse do sistema original (AGUIRRE, 2015).

Aguirre (2015) em seu livro apresenta diversas ferramentas para auxiliar na validação dos modelos.

### 5.2 Desenvolvimento do controlador

Através das eqs. (4.1), (4.2a) e (4.2b) e também do modelo experimental que será obtido da planta piloto, utilizaremos a representação de espaço de estados para a construção do modelo que irá compor os controladores.

Serão utilizadas duas abordagens principais:

- **Utilização apenas de linguagens de programação, sem o auxílio de ferramentas de desenvolvimento MPC:** esta abordagem tem como meta aplicar o controle MPC através dos algoritmos descritos nas obras de Wang (2009), Rawlings e Mayne (2015), Rossiter (2003), entre outros, com a finalidade de detalhar cada etapa dos cálculos realizados pelo controlador visando cumprir os seguintes objetivos descritos na seção 1.2: estudo do algoritmo do controle MPC; comparação entre a implementação em duas ou mais plataformas e compilação de material teórico e experimental sobre MPC.

- **Utilização de ferramentas de desenvolvimento MPC:** esta abordagem visa utilizar ferramentas consolidadas de mercado para o desenvolvimento de controlador MPC, como o *Model Predictive Control Toolbox*, disponível no MATLAB® para poder realizar a comparação entre o desempenho de um controlador MPC em comparação a um controlador PID, como também foi proposto nos objetivos da seção 1.2.

# 6 Procedimentos Experimentais

Com o intuito de poder fazer uma comparação entre as respostas obtidas em todos os experimentos descritos nas seções a seguir, as condições de operação utilizadas serão as mesmas em todos os casos.

## 6.1 Planta SISO

### 6.1.1 Identificação do modelo

#### 6.1.1.1 Coleta

Seguindo os passos de identificação do sistema descrito na seção 5.1.1 e com o auxílio da ferramenta de identificação de sistemas do MATLAB®, foi possível modelar o sistema em sua configuração SISO.

Foi aplicado um sinal do tipo degrau na entrada do sistema e coletados os valores da sua saída. A coleta de dados teve duração de 700s e período de amostragem de 0,5s, sendo que o sinal de degrau foi disparado apenas após 30s do início do experimento, fazendo com que o aquecedor de entrada sofresse uma excitação de 0 à 70% de seu valor total<sup>1</sup>.

A fig. 15 mostra a configuração dos blocos no Simulink para a obtenção dos dados de entrada e saída da planta durante a simulação. O gráfico de resposta do teste com entrada degrau pode ser visualizado na fig. 16.

#### 6.1.1.2 Determinação da estrutura, parâmetros e validação

Através da ferramenta de identificação de sistemas do MATLAB® foi possível utilizar os dados obtidos durante a coleta para obtenção de funções de transferência e de representações através de espaço de estados.

A fig. 17 mostra os diferentes modelos testados e indica o Maior Valor do Fator de Ajuste (FIT) para cada um deles.

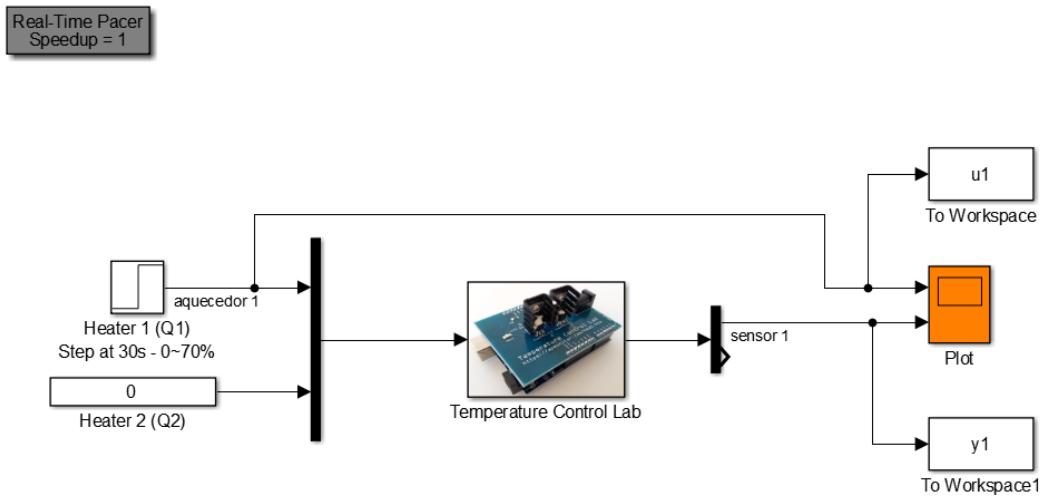
A partir dos modelos testados é possível destacar a função de transferência e o modelo de espaço de estados que apresentam maior FIT e menor ordem. Eles estão indicados nas eq. (6.1) e eq. (6.2) respectivamente.

$$tf\_4p2z = \frac{3.1952 \times 10^{-4}s^2 - 7.0930 \times 10^{-8}s + 6.7656 \times 10^{-8}}{s^4 + 6.3814 \times 10^{-4}s^3 + 1.3026 \times 10^{-5}s + 9.0146 \times 10^{-8}} \quad (6.1)$$

---

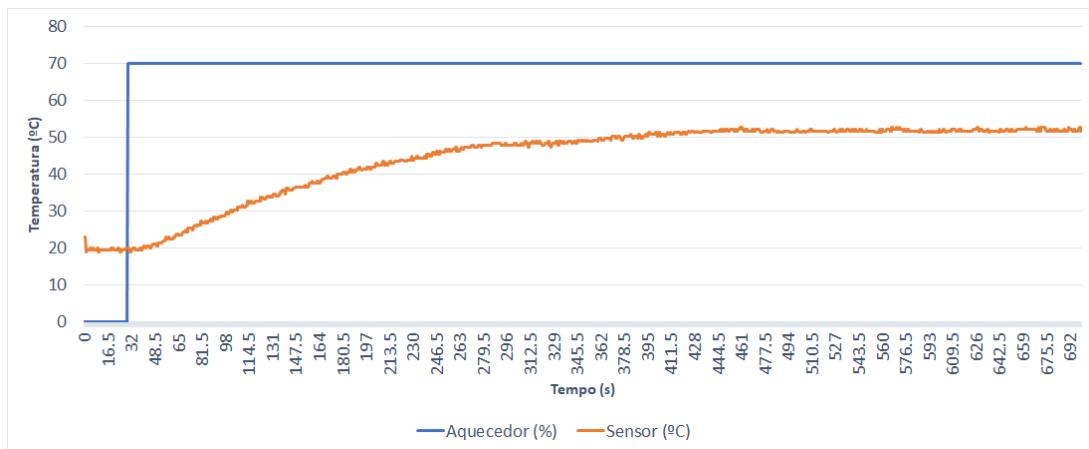
<sup>1</sup> Dados do teste de resposta degrau disponíveis em: <[https://github.com/TiagoPrata/MasterThesis/blob/master/7\\_matlab/01-SISO/StepTest.csv](https://github.com/TiagoPrata/MasterThesis/blob/master/7_matlab/01-SISO/StepTest.csv)>

Figura 15 – Simulink para coleta de dados SISO



Fonte: Autor

Figura 16 – Gráfico da coleta de dados SISO

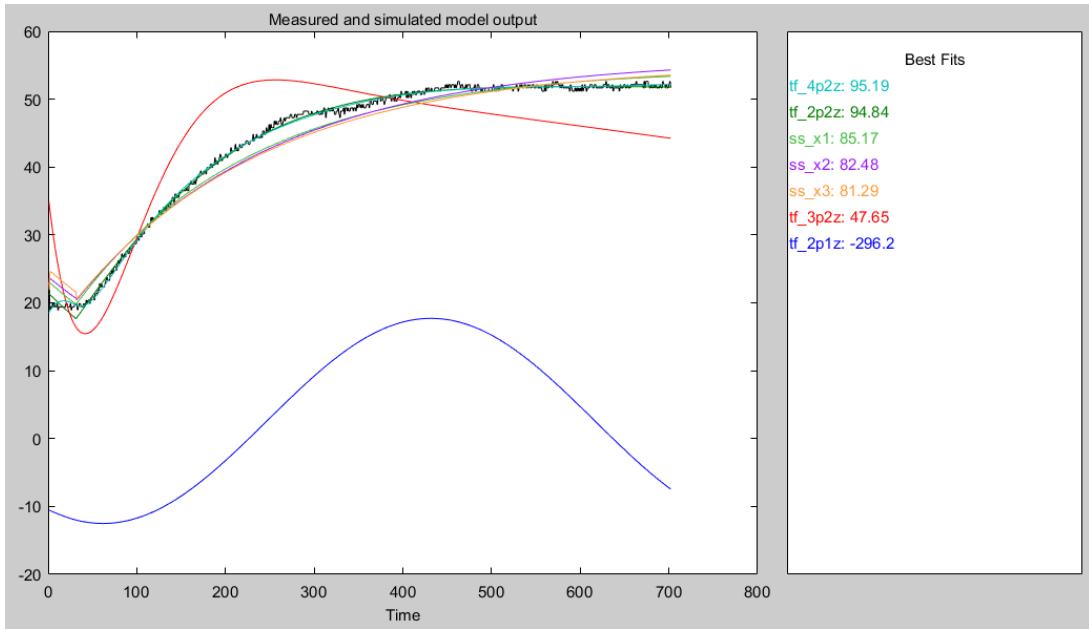


Fonte: Autor

$$\begin{aligned} \frac{dx}{dt} &= -5.071 \times 10^{-3} x(t) + 1.069 \times 10^{-5} u(t) + 4.776 \times 10^{-3} e(t) \\ y(t) &= 369.5 x(t) + 0 u(t) + e(t) \end{aligned} \quad (6.2)$$

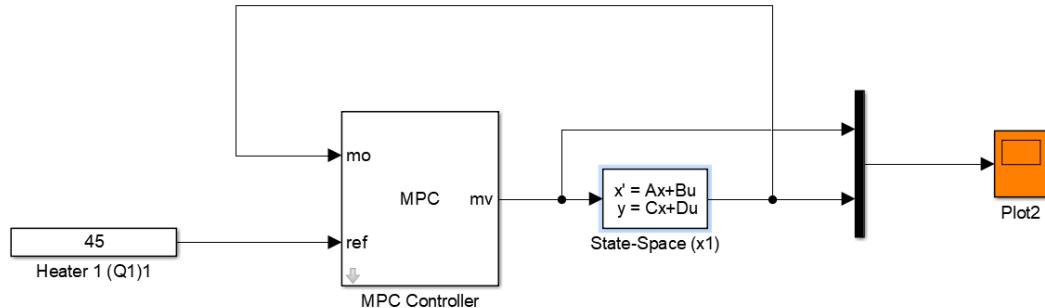
Conforme indicado na fig. 17, o modelo da eq. (6.1) apresenta uma aproximação de 95,19% e o modelo da eq. (6.2) de 85,17%.

Figura 17 – Modelos para planta SISO



Fonte: Autor

Figura 18 – Diagrama de bloco para parametrização de MPC em planta SISO



Fonte: Autor

### 6.1.2 Desenvolvimento do controlador

A partir do modelo de espaço de estados apresentado na eq. (6.2), criou-se então um diagrama de blocos para a parametrização de um controlador **MPC** utilizando o *Model Predictive Control Toolbox* do **MATLAB®**.

A fig. 18 mostra o diagrama de blocos utilizado para a parametrização do controle **MPC**.

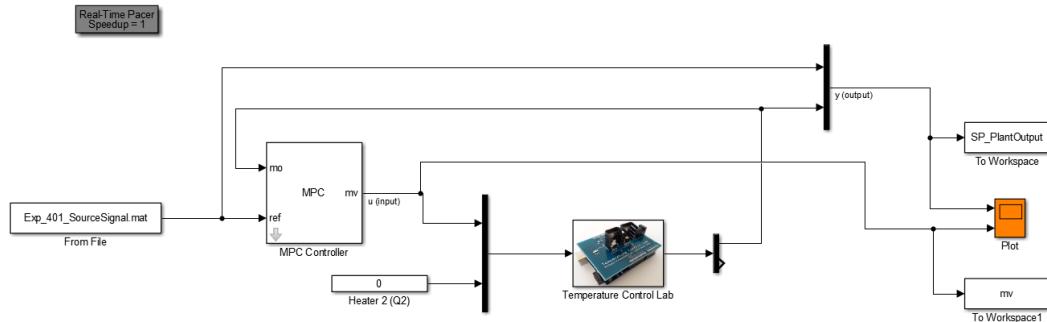
A sintonia do controlador foi baseada tanto na teoria de controle como em resultados experimentais, sendo que os dados do controlador são detalhados na tabela 3.

Tabela 3 – Características do controlador MPC para planta SISO

Descrição	Valor
Quantidade de estados	1
Quantidade de entradas	1
Quantidade de saídas	1
Quantidade de variáveis manipuladas	1
Quantidade de distúrbios medidos	0
Quantidade de distúrbios não medidos	0
Quantidade de saídas medidas	1
Tempo de amostragem	1 segundo
Horizonte de predição	100
Horizonte de controle	15
Peso: Variáveis Manipuladas	0
Peso: Taxa de Variáveis Manipuladas	0.0619
Peso: Variáveis de Saída	1.6161
Peso: ECR	100000
Estimador de Estados	Filtro de Kalman padrão
Limitantes: Entrada	Entre 0 e 100%

Fonte: Autor

Figura 19 – Diagrama de bloco para aplicação de MPC em planta SISO

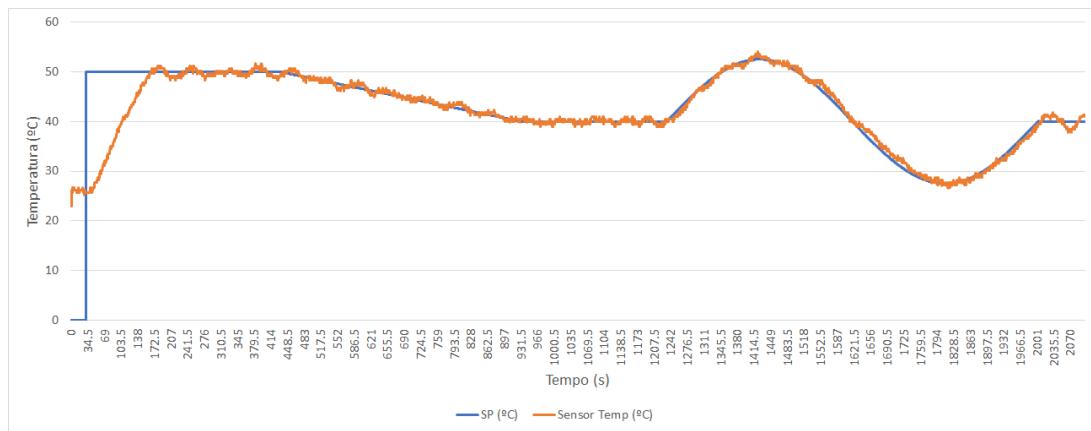


Fonte: Autor

### 6.1.3 Aplicação do controlador MPC na planta SISO

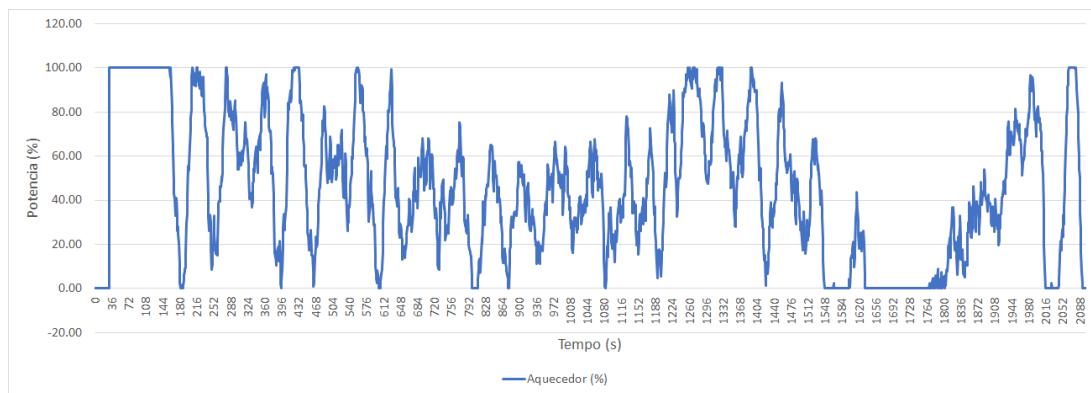
Através do controlador descrito na [tabela 3](#) pode-se então desenvolver um diagrama de blocos para a implementação do controle na planta [TCLab](#). O diagrama de blocos é apresentado na [fig. 19](#).

O resultado da aplicação do controlador conforme o diagrama de blocos da [fig. 19](#) pode ser verificado nos gráficos presentes das [figs. 20 a 22](#).

Figura 20 – MPC em planta SISO - *Setpoint* e Valor medido

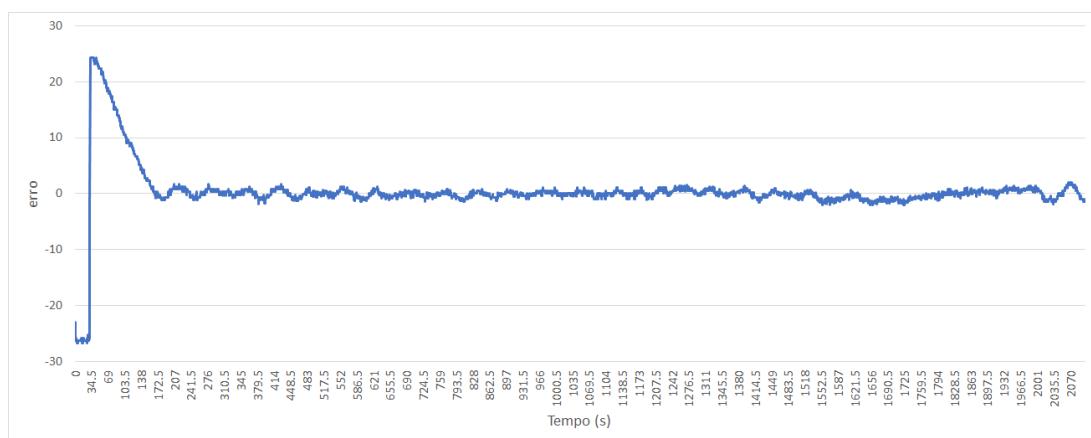
Fonte: Autor

Figura 21 – MPC em planta SISO - Potência do aquecedor de entrada



Fonte: Autor

Figura 22 – MPC em planta SISO - Erro calculado



Fonte: Autor



## 7 Cronograma

Tabela 4 – Cronograma de trabalho

Atividade	Jul-19	Ago-19	Set-19	Out-19	Nov-19
Modelagem experimental	X				
Desenvolvimento do controlador	X	X	X		
Análise de resultados			X	X	
Ajuste de dissertação					X

Fonte: Autor



## Referências

- AGUIRRE, L. A. **Introdução à Identificação de Sistemas - Técnicas Lineares e Não Lineares: Teoria e Aplicação.** 4. ed. Belo Horizonte: Editora UFMG, 2015. 776 p. ISBN 978-85-423-0079-6. Disponível em: <<https://www.editora.ufmg.br/#/pages/obra/444>>. Citado 3 vezes nas páginas 55, 56 e 57.
- ÅSTRÖM, K. J.; TORSTEN, B. Numerical Identification of Linear Dynamic Systems from Normal Operating Records. **IFAC Proceedings Volumes**, v. 2, n. 2, p. 96–111, sep 1965. ISSN 14746670. Disponível em: <<https://linkinghub.elsevier.com/retrieve/pii/S1474667017690244>>. Citado na página 55.
- ÅSTRÖM, K. J.; WITTENMARK, B. The Self-Tuning Regulators Revisited. **IFAC Proceedings Volumes**, v. 18, n. 5, p. xxix–xxxvii, jul 1985. ISSN 14746670. Disponível em: <<https://linkinghub.elsevier.com/retrieve/pii/S1474667017605342>>. Citado na página 24.
- BALLIN, S. L. **Controlador preditivo multivariável com restrição de excitação para identificação de processos em malha fechada.** Tese (Mestrado) — Universidade de São Paulo, São Paulo, apr 2008. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/3/3137/tde-15092008-153026/>>. Citado na página 55.
- BOLOGNANI, S. et al. Design and Implementation of Model Predictive Control for Electrical Motor Drives. **IEEE Transactions on Industrial Electronics**, v. 56, n. 6, p. 1925–1936, 2009. ISSN 0278-0046. Disponível em: <<http://ieeexplore.ieee.org/document/4663825/>>. Citado na página 38.
- BORRELLI, F.; BEMPORAD, A.; MORARI, M. **Predictive control for linear and hybrid systems.** New York: Cambridge University Press, 2017. 440 p. ISBN 978-1107652873. Disponível em: <[www.cambridge.org/9781107016880](http://www.cambridge.org/9781107016880)>. Citado na página 26.
- CAMACHO, E. F.; ALBA, C. B.; BORDONS, C. **Model Predictive control.** [s.n.], 2007. 405 p. ISSN 1098-6596. ISBN 9781852336943. Disponível em: <[link.springer.com/10.1007/978-0-85729-398-5](http://link.springer.com/10.1007/978-0-85729-398-5)>. Citado 3 vezes nas páginas 23, 26 e 43.
- DUNIA, R.; EDGAR, T. F.; HAUGEN, F. A. A complete programming framework for process control education. In: **2008 IEEE International Conference on Control Applications**. IEEE, 2008. p. 516–521. ISBN 978-1-4244-2222-7. Disponível em: <<http://ieeexplore.ieee.org/document/4629594>>. Citado na página 55.
- EYKHOFF, P. **System Identification: Parameter and State Estimation.** London: Wiley, 1974. ISBN 9780471249801. Citado na página 57.
- FAVARO, J. **Controle preditivo aplicado à planta piloto de neutralização de pH.** Tese (Mestrado) — Universidade de São Paulo, São Paulo, sep 2012. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/3/3139/tde-16072013-170810>>. Citado 2 vezes nas páginas 56 e 57.

GARCIA, C. **Modelagem e Simulação de Processos Industriais e de Sistemas Eletromecânicos**. 2. ed. São Paulo: Edusp, 2005. 688 p. ISBN 978-85-314-0904-2. Disponível em: <<https://www.edusp.com.br/detlivro.asp?ID=3140904>>. Citado na página 56.

GARCIA, C. **Modelagem e Simulação de Processos Industriais e de Sistemas Eletromecânicos**. 2. ed. rev. ed. São Paulo: Edusp, 2013. 678 p. ISBN 978-85-314-0904-2. Disponível em: <<https://www.edusp.com.br/detlivro.asp?ID=3140904>>. Citado na página 23.

GEVERS, M. A Personal view of the development of system identification: A 30-year journey through an exciting field. **IEEE Control Systems**, IEEE, v. 26, n. 6, p. 93–105, 2006. ISSN 1066033X. Disponível em: <<https://ieeexplore.ieee.org/document/4019326>>. Citado na página 55.

GONÇALVES, D. **Implementação Prática de um Controlador Preditivo a um Processo Não-Linear**. 98 p. Tese (Mestrado) — Universidade Federal de Uberlândia, 2012. Disponível em: <<https://repositorio.ufu.br/bitstream/123456789/15185/1/d.pdf>>. Citado na página 23.

GUSTAVSSON, I. Survey of applications of identification in chemical and physical processes. **Automatica**, v. 11, n. 1, p. 3–24, jan 1975. ISSN 00051098. Disponível em: <<https://linkinghub.elsevier.com/retrieve/pii/0005109875900059>>. Citado na página 55.

HAUGEN, F. A. **A brief introduction to optimization methods Contents**. 2018. Disponível em: <[http://techteach.no/fag/process\\_control\\_nmbu\\_2018/optim/optimization\\_finn\\_haugen\\_2018.pdf](http://techteach.no/fag/process_control_nmbu_2018/optim/optimization_finn_haugen_2018.pdf)>. Citado 16 vezes nas páginas 29, 31, 32, 33, 35, 36, 37, 38, 42, 44, 47, 48, 74, 75, 82 e 89.

HEDENGREN, J. D. **Temperature Control Lab A: SISO (Single Input, Single Output) Model**. 201–? Disponível em: <[apmonitor.com/do/uploads/Main/Lab\\_A\\_SISO\\_Model.pdf](http://apmonitor.com/do/uploads/Main/Lab_A_SISO_Model.pdf)>. Citado na página 52.

HEDENGREN, J. D. **Temperature Control Lab B: MIMO (Multiple Input, Multiple Output) Model**. 201–? Disponível em: <[https://apmonitor.com/do/uploads/Main/Lab\\_B\\_MIMO\\_Model.pdf](http://apmonitor.com/do/uploads/Main/Lab_B_MIMO_Model.pdf)>. Citado 2 vezes nas páginas 51 e 53.

HO, B. L.; KALMAN, R. E. Effective construction of linear state-variable models from input/output functions. **At-Automatisierungstechnik**, v. 14, n. 1-12, p. 545–548, jan 1966. ISSN 2196677X. Disponível em: <<http://www.degruyter.com/view/j/auto.1966.14.issue-1-12/auto.1966.14.112.545/auto.1966.14.112.545.xml>>. Citado na página 55.

LEE, J. H. Model predictive control: Review of the three decades of development. **International Journal of Control, Automation and Systems**, v. 9, n. 3, p. 415–424, jun 2011. ISSN 1598-6446. Disponível em: <<http://link.springer.com/10.1007/s12555-011-0300-6>>. Citado 3 vezes nas páginas 24, 25 e 26.

OGATA, K. **Engenharia de Controle Moderno**. 5. ed. São Paulo: Pearson Prentice Hall, 2010. 809 p. ISBN 978-85-7605-810-6. Citado na página 23.

PARKINSON, A.; BALLING, R.; HEDENGREN, J. D. **Optimization Methods for Engineering Design**. Brigham Young University, 2018. v. 2. ISBN 0521665698.

Disponível em: <<http://apmonitor.com/me575/index.php/Main/BookChapters>>. Citado na página 26.

PRACEK, B. et al. Aplicação de controle feedforward para regulação da temperatura num protótipo de túnel de vento. In: . [s.n.], 2011. p. 532–535. Disponível em: <<http://dincon.org.br/articles/view/id/4fe9bd8d1ef1fa0163000006>>. Citado na página 55.

RAWLINGS, J. B.; MAYNE, D. Q. **Model Predictive Control: Theory and Design**. Madison: Nob Hill Publishing, 2015. 533 p. ISBN 9780975937709. Citado na página 57.

ROSSITER, J. A. **Model- Based Predictive Control - A practical approach**. [S.l.: s.n.], 2003. 337 p. ISBN 0849312914. Citado 2 vezes nas páginas 43 e 57.

RUGGIERO, M. A. G.; LOPES, V. L. d. R. **Cálculo Numérico: aspectos teóricos e computacionais**. 2. ed. São Paulo: Pearson Makron Books, 2000. 406 p. ISBN 978-85-346-0204-4. Citado na página 36.

SEBORG, D. E.; EDGAR, T. F.; MELLICHAMP, D. A. Model Predictive Control. In: **Process Dynamics and Control**. [S.l.: s.n.], 2011. p. 414–438. ISBN 0471000779. Citado na página 25.

STADLER, K. S.; POLAND, J.; GALLESTEY, E. Model predictive control of a rotary cement kiln. **Control Engineering Practice**, Elsevier, v. 19, n. 1, p. 1–9, 2011. ISSN 09670661. Disponível em: <<http://dx.doi.org/10.1016/j.conengprac.2010.08.004>>. Citado na página 38.

VUKOV, M. **Embedded Model Predictive Control and Moving Horizon Estimation for Mechatronics Applications**. Tese (Doutorado) — KU Leuven, 2015. Disponível em: <[https://lirias.kuleuven.be/handle/123456789/487664https://lirias.kuleuven.be/bitstream/123456789/487664/1/thesis\\_final\\_print.pdf](https://lirias.kuleuven.be/handle/123456789/487664https://lirias.kuleuven.be/bitstream/123456789/487664/1/thesis_final_print.pdf)>. Citado na página 45.

WANG, L. **Model Predictive Control System Design and Implementation Using MATLAB®**. London: Springer London, 2009. 403 p. (Advances in Industrial Control, 9781848823303). ISSN 1430-9491. ISBN 978-1-84882-330-3. Disponível em: <<http://link.springer.com/10.1007/978-1-84882-331-0>>. Citado 2 vezes nas páginas 56 e 57.

YAKUB, F.; MORI, Y. Model predictive control for car vehicle dynamics system - Comparative study. In: **2013 IEEE Third International Conference on Information Science and Technology (ICIST)**. Yangzhou: IEEE, 2013. p. 150–155. ISBN 978-1-4673-2764-0. Disponível em: <<http://ieeexplore.ieee.org/document/6747530/>>. Citado na página 26.



# Apêndices



# APÊNDICE A – Códigos-fonte

## A.1 Método de pesquisa em grade

Código-fonte A.1 – Pesquisa em grade com número escalar

---

```

# Linguagem:           Python
# Autor:               Tiago Correia Prata
# Disponível em:
# <https:
#   //github.com/TiagoPrata/MasterThesis/blob/master/4_codes/grid_search_scalar.py>

import numpy as np
import matplotlib.pyplot as plt

def f_obj(x):
    return x**4.0 * 0.00232 - x**3.0 * 0.111 + x**2.0 * 1.80 - x * 11.6
    + 34.4

x_min = 2
x_max = 22
N_x = 100
x_array = np.linspace(x_min, x_max, N_x)
f_array = np.dot(x_array, 0)

f_min = float('inf')
x_opt = float('-inf')

# Laco varrendo todos os valores de x
for i in range(len(x_array)):
    x = x_array[i]

    f = f_obj(x)

    # Armazena os valores caso seja a melhor solucao
    if f <= f_min:
        f_min = f
        x_opt = x

    f_array[i] = f
    x_array[i] = x

# Apresenta valores calculados
print(f_min)

```

---

```

print(x_opt)

# Configurações de gráfico
plt.plot(x_array, f_array, 'b.')
plt.plot(x_opt, f_min, 'ro')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
# Plota gráfico (Exibido na fig. 1)
plt.show()

```

---

Fonte: Autor, adaptado de Haugen (2018)

## A.2 Método de pesquisa em grade com duas variáveis

Código-fonte A.2 – Pesquisa em grade com duas variáveis

---

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# TODO: Inserir comentários

x1_min = 0
x1_max = 2
N_x1 = 100
x1_array = np.linspace(x1_min, x1_max, N_x1)

x2_min = 1
x2_max = 3
N_x2 = 100
x2_array = np.linspace(x2_min, x2_max, N_x2)

f_min = float('inf')
x1_opt = float('-inf')
x2_opt = float('-inf')

f_matrix = np.zeros([len(x1_array), len(x2_array)])

def fun_obj(x1, x2):
    return (x1-1)**2 + (x2-2)**2 + 0.5

for k_x1 in range(len(x1_array)):
    x1 = x1_array[k_x1]
    for k_x2 in range(len(x2_array)):
        x2 = x2_array[k_x2]

```

---

```

f = fun_obj(x1, x2)

if not (x1-x2+1.5 <= 0):
    f = float('inf')

if f <= f_min:
    f_min = f
    x1_opt = x1
    x2_opt = x2

f_matrix[k_x1, k_x2] = f

print(f_min)
print(x1_opt)
print(x2_opt)

# TODO: Fazer plot para imagem 3d
# fig = plt.figure()
# ax = Axes3D(fig)
# x, y = np.meshgrid(x1_array, x2_array)
# ax.plot_surface(x1_array, x2_array, f_matrix, rstride=1, cstride=1)
# plt.show()

```

Fonte: Autor, adaptado de Haugen (2018)

### A.3 Exemplo do método Estimador de Horizonte Móvel

Código-fonte A.3 – Exemplo do método Estimador de Horizonte Móvel

```

# Linguagem:           Python
# Autor:              Tiago Correa Prata
# Disponível em:
# <https://github.com/TiagoPrata/MasterThesis/blob/master/4\_codes/mhe\_example.py>

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# model param
K = 1      # Gain
T = 2      # Time constant
n = 3      # number of state variables
model_params = {'K': K, 'T': T}
cov_process_disturb_w1 = 0.001

```

```

cov_process_disturb_w2 = 0.001
cov_process_disturb_w3 = 0.001
cov_process_disturb_w = np.diag([cov_process_disturb_w1 ,
    cov_process_disturb_w2 , cov_process_disturb_w3])
cov_meas_noise_v1 = 0.01
cov_meas_noise_v = np.diag([cov_meas_noise_v1])

# Time settings
Ts = 0.5          # s
t_start = 0        # s
t_stop = 20        # s
t_array = np.linspace(t_start, t_stop-Ts, t_stop / Ts)
N = len(t_array)
t_mhe = 5
N_mhe = int(np.floor(t_mhe/Ts))
number_optim_vars = n*N_mhe

# Preallocation of arrays for storage
u_sim_array = t_array*0
x1_sim_array = t_array*0
x2_sim_array = t_array*0
x3_sim_array = t_array*0
y1_sim_array = t_array*0
x1_est_optim_plot_array = t_array*0
x2_est_optim_plot_array = t_array*0
x3_est_optim_plot_array = t_array*0

# Sim initialization
x1_sim_init = 2
x2_sim_init = 3
x1_sim_k = x1_sim_init
x2_sim_k = x2_sim_init

# MHE initialization
mhe_array = np.zeros(N_mhe)

x1_est_init_guess = 0
x2_est_init_guess = 0
x3_est_init_guess = 2
x1_est_optim_array = np.zeros(N_mhe) + x1_est_init_guess
x2_est_optim_array = np.zeros(N_mhe) + x2_est_init_guess
x3_est_optim_array = np.zeros(N_mhe) + x3_est_init_guess

x_est_guess_matrix = np.concatenate(([x1_est_optim_array] , [
    x2_est_optim_array] , [x3_est_optim_array]))

u_mhe_array = mhe_array * 0

```

```

y1_meas_mhe_array = mhe_array * 0

# Configuring continuous plotting
plt.ion()

def objective(x_est_matrix):
    # Reshaping matrix because it is flattened after minimine call
    x_est_matrix = x_est_matrix.reshape(3, N_mhe)

    K = model_params['K']
    T = model_params['T']
    Q = covars['Q']
    R = covars['R']

    J_km1 = 0

    for k in range(N_mhe-1):
        u_k = u_mhe_array[k]
        x_k = x_est_matrix[:, k]
        x1_k = x_k[0]
        x2_k = x_k[1]
        x3_k = x_k[2]
        h1_k = x1_k
        y1_meas_k = y1_meas_mhe_array[k]
        v1_k = y1_meas_k - h1_k
        v_k = v1_k

        if k <= N_mhe-1:
            x_kp1 = x_est_matrix[:, k+1]
            x1_kp1 = x_kp1[0]
            x2_kp1 = x_kp1[1]
            x3_kp1 = x_kp1[2]

            dx1_dt_k = x2_k
            dx2_dt_k = (-x2_k + K*u_k + x3_k)/T
            dx3_dt_k = 0
            f1_k = x1_k + Ts*dx1_dt_k
            f2_k = x2_k + Ts*dx2_dt_k
            f3_k = x3_k + Ts*dx3_dt_k
            w1_k = [x1_kp1 - f1_k]
            w2_k = [x2_kp1 - f2_k]
            w3_k = [x3_kp1 - f3_k]
            w_k = np.concatenate([w1_k, w2_k, w3_k]).T

            dJ_K = np.dot(np.dot(w_k.T, np.linalg.inv(Q)), w_k) + np.dot(np.
                dot(v_k.T, np.linalg.inv(R)), v_k)
            J_k = J_km1 + dJ_K

```

```
# Time shift
J_km1 = J_k

return J_k

# Sim loop
for k in range(N):
    t_k = k*T_s

    # Process simulator
    if t_k < 2:
        u_k = 2
        d_k = 1

    # Change of u
    if t_k >= 2:
        u_k = 2

    # Change of d
    if t_k >= 8:
        d_k = 1

    # Change of u
    if t_k >= 10:
        u_k = 6

    # Change of d
    if t_k >= 15:
        d_k = 2

    # derivatives
    dx1_sim_dt_k = x2_sim_k
    dx2_sim_dt_k = (-x2_sim_k + K*u_k + d_k)/T
    f1_sim_k = x1_sim_k + T_s*dx1_sim_dt_k
    f2_sim_k = x2_sim_k + T_s*dx2_sim_dt_k

    # Integration and adding disturbance
    w1_sim_k = np.sqrt(cov_process_disturb_w1) * np.random.rand()
    w2_sim_k = np.sqrt(cov_process_disturb_w2) * np.random.rand()
    x1_sim_kp1 = f1_sim_k + w1_sim_k
    x2_sim_kp1 = f2_sim_k + w2_sim_k

    # Calculating output and adding meas noise
    v1_sim_k = np.sqrt(cov_meas_noise_v1) * np.random.rand()
    y1_sim_k = x1_sim_k + v1_sim_k
```

```

# Storage
t_array[k] = t_k
u_sim_array[k] = u_k
x1_sim_array[k] = x1_sim_k
x2_sim_array[k] = x2_sim_k
x3_sim_array[k] = d_k
y1_sim_array[k] = y1_sim_k

# Preparing for time shift:
x1_sim_k = x1_sim_kp1
x2_sim_k = x2_sim_kp1

# Updating u and y for use in MHE
u_mhe_array = np.append(u_mhe_array[1:N_mhe], u_k)
y1_meas_mhe_array = np.append(y1_meas_mhe_array[1:N_mhe], y1_sim_k)
y_meas_mhe_array = [y1_meas_mhe_array]

if k > N_mhe:
    Q = cov_process_disturb_w
    R = cov_meas_noise_v
    covars = {'Q': Q, 'R': R}

# minimize initialization
x1_est_init_error = [0]
x2_est_init_error = [0]
x3_est_init_error = [0]
x_est_init_error = np.concatenate(([x1_est_init_error], [
    x2_est_init_error], [x3_est_init_error]))

# Guessed optim states:
# Guessed present state ( $x_k$ ) is needed to calculate optimal
# present meas
# ( $y_k$ ). Model is used in prediction:
x1_km1 = x1_est_optim_array[N_mhe-1]
x2_km1 = x2_est_optim_array[N_mhe-1]
x3_km1 = x3_est_optim_array[N_mhe-1]

dx1_dt_km1 = x2_km1
dx2_dt_km1 = (-x2_km1 + K*u_k + x3_km1)/T
dx3_dt_km1 = 0

x1_pred_k = x1_km1 + Ts*dx1_dt_km1
x2_pred_k = x2_km1 + Ts*dx2_dt_km1
x3_pred_k = x3_km1 + Ts*dx3_dt_km1

# Now, guessed optimal states are:

```

```

x1_est_guess_array = np.append(x1_est_optim_array[1:N_mhe] ,
                               x1_pred_k)
x2_est_guess_array = np.append(x2_est_optim_array[1:N_mhe] ,
                               x2_pred_k)
x3_est_guess_array = np.append(x3_est_optim_array[1:N_mhe] ,
                               x3_pred_k)
# x_est_guess_matrix = np.concatenate((x1_est_guess_array,
#                                       x2_est_guess_array, x3_est_guess_array))
x_est_guess_matrix = [[x1_est_guess_array], [x2_est_guess_array],
                      [x3_est_guess_array]]

# Lower and upper limits of optim variables:
x1_est_max = 100
x2_est_max = 10
x3_est_max = 10

x1_est_min = -100
x2_est_min = -10
x3_est_min = -10

# Creating a flatten array of bounderies
# The bounderies must have the same size of x0 (
# x_est_guess_matrix)
bnds = [(x1_est_min,x1_est_max)]*N_mhe + [(x2_est_min,x2_est_max)
                                              ]*N_mhe + [(x3_est_min,x3_est_max)]*N_mhe

x_est_optim_matrix_obj = minimize(objective, x_est_guess_matrix,
                                    method='SLSQP', bounds=bnds)
# Reshaping matrix because it is flattened after minimine call
x_est_optim_matrix = x_est_optim_matrix_obj.x.reshape(3,N_mhe)

x1_est_optim_array = x_est_optim_matrix[0]
x2_est_optim_array = x_est_optim_matrix[1]
x3_est_optim_array = x_est_optim_matrix[2]

x1_est_optim_plot_array[k] = x1_est_optim_array[-1]
x2_est_optim_plot_array[k] = x2_est_optim_array[-1]
x3_est_optim_plot_array[k] = x3_est_optim_array[-1]

# Continuous plotting
x_lim_array = [t_start, t_stop]
if (k>0 and k<N):
    if k > N_mhe:
        plt.pause(1)
    else:
        plt.pause(0.1)

```

```
plt.subplot(4,1,1)
lineU, = plt.plot([t_array[k-1],t_array[k]],[u_sim_array[k-1],
u_sim_array[k]],'b-o')
if (k == 1):
    lineU.set_label('u')
    plt.legend()
    plt.xlim(x_lim_array)
    plt.ylim([0, 10])
    plt.title('u')

plt.subplot(4,1,2)
lineX1Sim, lineX1Est, = plt.plot([t_array[k-1],t_array[k]],
x1_est_optim_plot_array[k-1],x1_est_optim_plot_array[k],'r-o',
',[t_array[k-1],t_array[k]],[x1_sim_array[k-1],x1_sim_array[k]],
'b-o')
if (k == 1):
    lineX1Sim.set_label('x1 sim')
    lineX1Est.set_label('x1 est')
    plt.legend()
    plt.xlim(x_lim_array)
    plt.ylim([0, 100])

plt.subplot(4,1,3)
lineX2Sim, lineX2Est, = plt.plot([t_array[k-1],t_array[k]],
x2_est_optim_plot_array[k-1],x2_est_optim_plot_array[k],'r-o',
',[t_array[k-1],t_array[k]],[x2_sim_array[k-1],x2_sim_array[k]],
'b-o')
if (k == 1):
    lineX2Sim.set_label('x2 sim')
    lineX2Est.set_label('x2 est')
    plt.legend()
    plt.xlim(x_lim_array)
    plt.ylim([0, 10])

plt.subplot(4,1,4)
lineX3Sim, lineX3Est, = plt.plot([t_array[k-1],t_array[k]],
x3_est_optim_plot_array[k-1],x3_est_optim_plot_array[k],'r-o',
',[t_array[k-1],t_array[k]],[x3_sim_array[k-1],x3_sim_array[k]],
'b-o')
if (k == 1):
    lineX3Sim.set_label('x3 sim')
    lineX3Est.set_label('x3 est')
    plt.legend()
    plt.xlim(x_lim_array)
    plt.ylim([0, 3])

plt.draw()
```

```
plt.show(block=True)
```

Fonte: Autor, adaptado de Haugen (2018)

## A.4 Exemplo de aplicação do Controle Preditivo Baseado em Modelo

Código-fonte A.4 – Exemplo de aplicação do Controle Preditivo Baseado em Modelo

```
# Linguagem: Python
# Autor: Tiago Correa Prata
# Disponível em:
# <https://github.com/TiagoPrata/MasterThesis/blob/master/4_codes/mpc_example.py>

import numpy as np
import math
from scipy import signal
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# -----
# Process params:
gain = 3.5                      #[deg C]/[V]
theta_const = 23                   #[s]
theta_delay = 3                    #[s]
model_params = {
    'gain': gain,
    'theta_const': theta_const,
    'theta_delay': theta_delay
}

Temp_env_k = 25                   #[deg C]

# -----
# Time settings:

Ts = 0.5                          #Time-step [s]
t_pred_horizon = 8

N_pred = int(t_pred_horizon/Ts)
t_start = 0
t_stop = 300
N_sim = int((t_stop-t_start)/Ts)
```

```

t = np.linspace(t_start, t_stop-Ts, t_stop / Ts)

# -----
# Storage allocations
Temp_sp_array = t*0

# -----
# MPC costs:

C_e = 1
C_du = 20
mpc_costs = {
    'C_e': C_e,
    'C_du': C_du
}

# -----
# Defining sequence for temp_out setpoint:

Temp_sp_const = 30          #[C]
Ampl_step = 2                #[C]
Slope = -0.04                 #[C/s]
Ampl_sine = 1                  #[C]
T_period = 50                  #[s]

t_const_start = t_start
t_const_stop = 100
t_step_start = t_const_stop
t_step_stop = 150
t_ramp_start = t_step_stop
t_ramp_stop = 200
t_sine_start = t_ramp_stop
t_sine_stop = 250
t_const2_start = t_sine_stop
t_const2_stop = t_stop

for k in range(N_sim):
    if (t[k] >= t_const_start and t[k] < t_const_stop):
        Temp_sp_array[k] = Temp_sp_const

    if (t[k] >= t_step_start and t[k] < t_step_stop):
        Temp_sp_array[k] = Temp_sp_const + Ampl_step

    if (t[k] >= t_ramp_start and t[k] < t_ramp_stop):
        Temp_sp_array[k] = Temp_sp_const + Ampl_step + Slope * (t[k] -
            t_ramp_start)

```

```

if (t[k] >= t_sine_start and t[k] < t_sine_stop):
    Temp_sp_array[k] = Temp_sp_const + Ampl_sine * math.sin(2*math.
        pi * (1/T_period) * (t[k] - t_sine_start))

if (t[k] >= t_const2_start):
    Temp_sp_array[k] = Temp_sp_const

#-----
#Initialization:

u_init = 0
N_delay = math.floor(theta_delay/Ts) + 1
delay_array = np.zeros(N_delay) + u_init

#-----
#Initial guessed optimal control sequence:

Temp_heat_sim_k = 0           #[C]
Temp_out_sim_k = 28           #[C]
d_sim_k = -0.5

#-----
#Initial values for estimator:

Temp_heat_est_k = 0           #[C]
Temp_out_est_k = 25           #[C]
d_est_k = 0                   #[V]

#-----
#Initial guessed optimal control sequence:
u_guess = np.zeros(N_pred) + u_init
u_guess = np.transpose(u_guess)

#-----
#Initial value of previous optimal value:
u_opt_km1 = u_init

# -----
# Defining arrays for plotting:

t_plot_array = np.zeros(N_sim)
Temp_out_sp_plot_array = np.zeros(N_sim)
Temp_out_sim_plot_array = np.zeros(N_sim)
u_plot_array = np.zeros(N_sim)
d_est_plot_array = np.zeros(N_sim)
d_sim_plot_array = np.zeros(N_sim)

```

```

# -----
# Matrices defining linear constraints for use in fmincon:

# A = []
# B = []
# Aeq = []
# Beq = []

# -----
# Lower and upper limits of optim variable for use in fmincon:

u_max = 5
u_min = 0
u_ub = np.zeros(N_pred) + u_max
u_lb = np.zeros(N_pred) + u_min

u_delayed_k = 2

# -----
# Calculation of observer gain for estimation of input disturbance, d:

A_est = np.array([[-1/theta_const, gain/theta_const],[0, 0]])
C_est = np.array([[1, 0]])
n_est = len(A_est)
Tr_est = 5
T_est = Tr_est/n_est
estimpoly = np.array([T_est*T_est, math.sqrt(2)*T_est, 1])
eig_est = np.roots(estimpoly)
K_est1 = signal.place_poles(A_est.transpose(), C_est.transpose(),
    eig_est)
K_est = K_est1.gain_matrix.transpose()

# Configuring continuous plotting
plt.ion()

def objective(u, state_est, Temp_sp_to_mpc_array):
    gain = model_params['gain']
    theta_const = model_params['theta_const']
    theta_delay = model_params['theta_delay']

    C_e = mpc_costs['C_e']
    C_du = mpc_costs['C_du']

    Temp_heat_k = state_est['Temp_heat_est_k']
    d_k = state_est['d_est_k']

    N_delay = math.floor(theta_delay/Ts) + 1

```

```

delay_array = np.zeros(N_delay) + u[0]

ru_km1 = u_opt_km1
u_km1 = u[0]
J_km1 = 0

for k in range(N_pred):
    u_k = u[k]
    Temp_sp_k = Temp_sp_to_mpc_array[k]

    # Time delay
    u_delayed_k = delay_array[N_delay-1]
    u_nondelayed_k = u_k
    delay_array = np.insert(delay_array, 0, u_nondelayed_k)
    delay_array = delay_array[:-1]

    # Solving diff eq
    dTemp_heat_dt_k = (1/theta_const) * (-Temp_heat_k + gain *(
        u_delayed_k + d_k))
    Temp_heat_kp1 = Temp_heat_k + Ts*dTemp_heat_dt_k
    Temp_out_k = Temp_heat_k + Temp_env_k

    # Updating objective function
    e_k = Temp_sp_k - Temp_out_k
    du_k = (u_k - u_km1)/Ts
    J_k = J_km1 + Ts*(C_e * e_k**2 + C_du * du_k**2)

    # Time shift
    Temp_heat_k = Temp_heat_kp1
    u_km1 = u_k
    J_km1 = J_k

return J_k

def constraints(u):
    cineq = np.array([])
    ceq = np.array([])

    return (cineq, ceq)

# Creating matrix
Temp_out_est_plot_array = np.zeros(N_sim - N_pred)

for k in range(N_sim - N_pred):
    t_k = t[k]
    t_plot_array[k] = t_k

```

```

# -----
# Observer for estimating input-disturbance d using Temp_out:
# Note: The time-delayed u is used as control signal shere.
e_est_k = Temp_out_sim_k - Temp_out_est_k

dTTemp_heat_est_dt_k = (1/theta_const) * (-Temp_heat_est_k + gain*(u_delayed_k + d_est_k)) + K_est[0] * e_est_k
dd_est_dt_k = 0 + K_est[1] * e_est_k

Temp_heat_est_kp1 = Temp_heat_est_k + Ts*dTemp_heat_est_dt_k
d_est_kp1 = d_est_k + Ts*dd_est_dt_k

Temp_out_est_k = Temp_heat_est_k + Temp_env_k

# -----
# Storage for plotting
Temp_out_est_plot_array[k] = Temp_out_est_k
d_est_plot_array[k] = d_est_k

# -----
# Setpoint array to optimizer
Temp_sp_to_mpc_array = Temp_sp_array[k:k+N_pred]
Temp_out_sp_plot_array[k] = Temp_sp_array[k]

# -----
# Estimated state to optimizer
state_est = {
    'Temp_heat_est_k': Temp_heat_est_k,
    'd_est_k': d_est_k
}

# -----
# Calculating optimal control sequence
u_opt = minimize(objective, u_guess, args=(state_est,
    Temp_sp_to_mpc_array), method='SLSQP')

u_guess = u_opt.x.reshape(1,N_pred)
u_k = u_opt.x[0]
u_plot_array[k] = u_k
u_opt_km1 = u_opt.x[0]

# -----
# Applying optimal control signal to simulated process
d_sim_k = -0.5
d_sim_plot_array[k] = d_sim_k

u_delayed_k = delay_array[N_delay-1]

```

```
u_nondelayed_k = u_k
delay_array = np.insert(delay_array, 0, u_nondelayed_k)
delay_array = delay_array[:-1]

dTTemp_heat_sim_dt_k = (1/theta_const) * (-Temp_heat_sim_k + gain * 
    u_delayed_k + d_sim_k))
Temp_heat_sim_kp1 = Temp_heat_sim_k + Ts*dTemp_heat_sim_dt_k
Temp_out_sim_k = Temp_heat_sim_k + Temp_env_k

Temp_out_sim_plot_array[k] = Temp_out_sim_k

# -----
# Time shift for estimator and for simulator
Temp_heat_est_k = Temp_heat_est_kp1
d_est_k = d_est_kp1

Temp_heat_sim_k = Temp_heat_sim_kp1

# -----
# Continuous plotting
x_lim_array = t_start
x_lim_array = np.append(x_lim_array, t_stop)

if (k > 0 and k < N_sim):
    plt.pause(0.1)

    plt.subplot(3,1,1)
    lineSP, lineSim, lineEst = \
        plt.plot([t_plot_array[k-1], t_plot_array[k]], \
            [Temp_out_sp_plot_array[k-1], Temp_out_sp_plot_array[k]], \
            'r-', \
            [t_plot_array[k-1], t_plot_array[k]], \
            [Temp_out_sim_plot_array[k-1], Temp_out_sim_plot_array[k]], \
            'b-', \
            [t_plot_array[k-1], t_plot_array[k]], \
            [Temp_out_est_plot_array[k-1], Temp_out_est_plot_array[k]], \
            'm-')
    if (k == 1):
        lineSP.set_label('SP')
        lineSim.set_label('sim')
        lineEst.set_label('est')
        plt.legend()
        plt.xlim(x_lim_array)
        plt.ylim([28, 33])
```

```
plt.xlabel('t [s]')
plt.ylabel('[deg C]')

plt.subplot(3,1,2)
lineU, = \
plt.plot([t_plot_array[k-1], t_plot_array[k]], \
         [u_plot_array[k-1], u_plot_array[k]], \
         'b-')
if (k == 1):
    lineU.set_label('u')
    plt.legend()
    plt.xlim(x_lim_array)
    plt.ylim([0, 5])
    plt.xlabel('t [s]')
    plt.ylabel('[V]')

plt.subplot(3,1,3)
line_d_est, line_d_sim = \
plt.plot([t_plot_array[k-1], t_plot_array[k]], \
         [d_est_plot_array[k-1], d_est_plot_array[k]], \
         'r-', \
         [t_plot_array[k-1], t_plot_array[k]], \
         [d_sim_plot_array[k-1], d_sim_plot_array[k]], \
         'b-')
if (k == 1):
    line_d_est.set_label('d est')
    line_d_sim.set_label('d sim')
    plt.legend()
    plt.xlim(x_lim_array)
    # plt.ylim([0, 5])
    plt.xlabel('t [s]')

plt.draw()

plt.show(block=True)
```

Fonte: Autor, adaptado de Haugen (2018)



## Anexos



# ANEXO A – Documentação de funções

## A.1 MATLAB® - fmincon

## fmincon

Find a minimum of a constrained nonlinear multivariable function

$$\min_x f(x) \text{ subject to}$$

$$\begin{aligned} c(x) &\leq 0 \\ ceq(x) &= 0 \\ A \cdot x &\leq b \\ Aeq \cdot x &= beq \\ lb \leq x &\leq ub \end{aligned}$$

where  $x$ ,  $b$ ,  $beq$ ,  $lb$ , and  $ub$  are vectors,  $A$  and  $Aeq$  are matrices,  $c(x)$  and  $ceq(x)$  are functions that return vectors, and  $f(x)$  is a function that returns a scalar.  $f(x)$ ,  $c(x)$ , and  $ceq(x)$  can be nonlinear functions.

### Syntax

```

x = fmincon(fun,x0,A,b)
x = fmincon(fun,x0,A,b,Aeq,beq)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2, ...)
[x,fval] = fmincon(...)
[x,fval,exitflag] = fmincon(...)
[x,fval,exitflag,output] = fmincon(...)
[x,fval,exitflag,output,lambda] = fmincon(...)
[x,fval,exitflag,output,lambda,grad] = fmincon(...)
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)
```

### Description

`fmincon` finds a constrained minimum of a scalar function of several variables starting at an initial estimate. This is generally referred to as *constrained nonlinear optimization* or *nonlinear programming*.

`x = fmincon(fun,x0,A,b)` starts at `x0` and finds a minimum `x` to the function described in `fun` subject to the linear inequalities `A*x <= b`. `x0` can be a scalar, vector, or matrix.

`x = fmincon(fun,x0,A,b,Aeq,beq)` minimizes `fun` subject to the linear equalities `Aeq*x = beq` as well as `A*x <= b`. Set `A=[]` and `b=[]` if no inequalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)` defines a set of lower and upper bounds on the design variables, `x`, so that the solution is always in the range `lb <= x <= ub`. Set `Aeq=[]` and `beq=[]` if no equalities exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` subjects the minimization to the nonlinear inequalities `c(x)` or equalities `ceq(x)` defined in `nonlcon`. `fmincon` optimizes such that `c(x) <= 0` and `ceq(x) = 0`. Set `lb = []` and/or `ub = []` if no bounds exist.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` minimizes with the optimization parameters specified in the structure `options`. Use `optimset` to set these parameters.

`x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the functions `fun` and `nonlcon`. Pass empty matrices as placeholders for `A`, `b`, `Aeq`, `beq`, `lb`, `ub`, `nonlcon`, and `options` if these arguments are not needed.

`[x,fval] = fmincon(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fmincon(...)` returns a value `exitflag` that describes the exit condition of `fmincon`.

`[x,fval,exitflag,output] = fmincon(...)` returns a structure `output` with information about the optimization.

`[x,fval,exitflag,output,lambda] = fmincon(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

`[x,fval,exitflag,output,lambda,grad] = fmincon(...)` returns the value of the gradient of `fun` at the solution `x`.

`[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(...)` returns the value of the Hessian of `fun` at the solution `x`.

## Input Arguments

[Function Arguments](#) contains general descriptions of arguments passed in to `fmincon`. This "Arguments" section provides function-specific details for `fun`, `nonlcon`, and `options`:

`fun` The function to be minimized. `fun` is a function that accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle.

```
x = fmincon(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ... % Compute function value at x
```

`fun` can also be an inline object.

```
x = fmincon(inline('norm(x)^2'),x0,A,b);
```

If the gradient of `fun` can also be computed *and* the `GradObj` parameter is 'on', as set by

```
options = optimset('GradObj','on')
```

then the function `fun` must return, in the second output argument, the gradient value `g`, a vector, at `x`. Note that by checking the value of `nargout` the function can avoid computing `g` when `fun` is called with only one output argument (in the case where the optimization algorithm only needs the value of `f` but not `g`).

```
function [f,g] = myfun(x)
f = ... % Compute the function value at x
if nargout > 1 % fun called with two output arguments
    g = ... % Compute the gradient evaluated at x
end
```

The gradient consists of the partial derivatives of `f` at the point `x`. That is, the *i*th component of `g` is the partial derivative of `f` with respect to the *i*th component of `x`. If the Hessian matrix can also be computed *and* the `Hessian` parameter is 'on', i.e., `options = optimset('Hessian','on')`, then the function `fun` must return the Hessian value `H`, a symmetric matrix, at `x` in a third output argument. Note that by checking the value of `nargout` we can avoid computing `H` when `fun` is called with only one or two output arguments (in the case where the optimization algorithm only needs the values of `f` and `g` but not `H`).

```
function [f,g,H] = myfun(x)
f = ... % Compute the objective function value at x
if nargout > 1 % fun called with two output arguments
    g = ... % Gradient of the function evaluated at x
    if nargout > 2
        H = ... % Hessian evaluated at x
    end
end
```

The Hessian matrix is the second partial derivatives matrix of `f` at the point `x`. That is, the  $(i,j)$ th component of `H` is the second partial derivative of `f` with respect to  $x_i$  and  $x_j$ ,  $\frac{\partial^2 f}{\partial x_i \partial x_j}$ . The Hessian is by definition a symmetric matrix.

`nonlcon` The function that computes the nonlinear inequality constraints  $c(x) \leq 0$  and the nonlinear equality constraints  $ceq(x) = 0$ . The function `nonlcon` accepts a vector  $x$  and returns two vectors `c` and `ceq`. The vector `c` contains the nonlinear inequalities evaluated at  $x$ , and `ceq` contains the nonlinear equalities evaluated at  $x$ . The function `nonlcon` can be specified as a function handle.

```
x = fmincon(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)
```

where `mycon` is a MATLAB function such as

```
function [c,ceq] = mycon(x)
c = ... % Compute nonlinear inequalities at x.
ceq = ... % Compute nonlinear equalities at x.
```

If the gradients of the constraints can also be computed *and* the `GradConstr` parameter is 'on', as set by

```
options = optimset('GradConstr','on')
```

then the function `nonlcon` must also return, in the third and fourth output arguments, `GC`, the gradient of  $c(x)$ , and `GCEq`, the gradient of  $ceq(x)$ . Note that by checking the value of `nargout` the function can avoid computing `GC` and `GCEq` when `nonlcon` is called with only two output arguments (in the case where the optimization algorithm only needs the values of `c` and `ceq` but not `GC` and `GCEq`).

```
function [c,ceq,GC,GCEq] = mycon(x)
c = ... % Nonlinear inequalities at x
ceq = ... % Nonlinear equalities at x
if nargout > 2 % nonlcon called with 4 outputs
    GC = ... % Gradients of the inequalities
    GCEq = ... % Gradients of the equalities
end
```

If `nonlcon` returns a vector `c` of  $m$  components and  $x$  has length  $n$ , where  $n$  is the length of  $x_0$ , then the gradient `GC` of  $c(x)$  is an  $n$ -by- $m$  matrix, where  $GC(i,j)$  is the partial derivative of  $c(j)$  with respect to  $x(i)$  (i.e., the  $j$ th column of `GC` is the gradient of the  $j$ th inequality constraint  $c(j)$ ). Likewise, if `ceq` has  $p$  components, the gradient `GCEq` of  $ceq(x)$  is an  $n$ -by- $p$  matrix, where  $GCEq(i,j)$  is the partial derivative of  $ceq(j)$  with respect to  $x(i)$  (i.e., the  $j$ th column of `GCEq` is the gradient of the  $j$ th equality constraint  $ceq(j)$ ).

`options` [Options](#) provides the function-specific details for the `options` parameters.

## Output Arguments

[Function Arguments](#) contains general descriptions of arguments returned by `fmincon`. This section provides function-specific details for `exitflag`, `lambda`, and `output`:

exitflag	Describes the exit condition:
> 0	The function converged to a solution $\mathbf{x}$ .
0	The maximum number of function evaluations or iterations was exceeded.
< 0	The function did not converge to a solution.
lambda	Structure containing the Lagrange multipliers at the solution $\mathbf{x}$ (separated by constraint type). The fields of the structure are:
lower	Lower bounds $\mathbf{lb}$
upper	Upper bounds $\mathbf{ub}$
ineqlin	Linear inequalities
eqlin	Linear equalities
ineqnonlin	Nonlinear inequalities
eqnonlin	Nonlinear equalities
output	Structure containing information about the optimization. The fields of the structure are:
iterations	Number of iterations taken.
funcCount	Number of function evaluations.
algorithm	Algorithm used.
cgiterations	Number of PCG iterations (large-scale algorithm only).
stepsize	Final step size taken (medium-scale algorithm only).
firstorderopt	Measure of first-order optimality (large-scale algorithm only). For large-scale bound constrained problems, the first-order optimality is the infinity norm of $\mathbf{v} \cdot \mathbf{g}$ , where $\mathbf{v}$ is defined as in <a href="#">Box Constraints</a> , and $\mathbf{g}$ is the gradient. For large-scale problems with only linear equalities, the first-order optimality is the infinity norm of the <i>projected</i> gradient (i.e. the gradient projected onto the nullspace of $\mathbf{A}_{\text{eq}}$ ).

## Options

Optimization options parameters used by `fmincon`. Some parameters apply to all algorithms, some are only relevant when using the large-scale algorithm, and others are only relevant when using the medium-scale algorithm. You can use [`optimset`](#) to set or change the values of these fields in the `parameters` structure, `options`. See [Optimization Parameters](#), for detailed information.

We start by describing the `LargeScale` option since it states a *preference* for which algorithm to use. It is only a preference since certain conditions must be met to use the large-scale algorithm. For `fmincon`, you must provide the gradient (see the description of `fun` above to see how) or else the medium-scale algorithm is used:

`LargeScale` Use large-scale algorithm if possible when set to '`on`'. Use medium-scale algorithm when set to '`off`'.

**Medium-Scale and Large-Scale Algorithms.** These parameters are used by both the medium-scale and large-scale algorithms:

`Diagnostics` Print diagnostic information about the function to be minimized.

`Display` Level of display. '`off`' displays no output; '`iter`' displays output at each iteration; '`final`' (default) displays just the final output.

`GradObj` Gradient for the objective function defined by user. See the description of `fun` above to see how to define the gradient in `fun`. You must provide the gradient to use the large-scale method. It is optional for the medium-scale method.

`MaxFunEvals` Maximum number of function evaluations allowed.

`MaxIter` Maximum number of iterations allowed.

`TolFun` Termination tolerance on the function value.

`TolCon` Termination tolerance on the constraint violation.

`TolX` Termination tolerance on `x`.

**Large-Scale Algorithm Only.** These parameters are used only by the large-scale algorithm:

`Hessian` If '`on`', `fmincon` uses a user-defined Hessian (defined in `fun`), or Hessian information (when using `HessMult`), for the objective function. If '`off`', `fmincon` approximates the Hessian using finite differences.

`HessMult` Function handle for Hessian multiply function. For large-scale structured problems, this function computes the Hessian matrix product  $H^*Y$  without actually forming  $H$ . The function is of the form

```
W = hmfun(Hinfo, Y, p1, p2, ...)
```

where `Hinfo` and the additional parameters `p1, p2, ...` contain the matrices used to compute  $H^*Y$ . The first argument must be the same as the third argument returned by the objective function `fun`.

`[f,g,Hinfo] = fun(x,p1,p2,...)`

The parameters `p1, p2, ...` are the same additional parameters that are passed to `fmincon` (and to `fun`).

`fmincon(fun,...,options,p1,p2,...)`

`Y` is a matrix that has the same number of rows as there are dimensions in the problem. `W = H^*Y` although `H` is not formed explicitly. `fmincon` uses `Hinfo` to compute the preconditioner.

**Note** 'Hessian' must be set to 'on' for `Hinfo` to be passed from `fun` to `hmfun`.

See [Nonlinear Minimization with a Dense but Structured Hessian and Equality Constraints](#) for an example.

<code>HessPattern</code>	Sparsity pattern of the Hessian for finite-differencing. If it is not convenient to compute the sparse Hessian matrix <code>H</code> in <code>fun</code> , the large-scale method in <code>fmincon</code> can approximate <code>H</code> via sparse finite-differences (of the gradient) provided the <i>sparsity structure</i> of <code>H</code> -- i.e., locations of the nonzeros -- is supplied as the value for <code>HessPattern</code> . In the worst case, if the structure is unknown, you can set <code>HessPattern</code> to be a dense matrix and a full finite-difference approximation is computed at each iteration (this is the default). This can be very expensive for large problems so it is usually worth the effort to determine the sparsity structure.
<code>MaxPCGIter</code>	Maximum number of PCG (preconditioned conjugate gradient) iterations (see the <i>Algorithm</i> section below).
<code>PrecondBandWidth</code>	Upper bandwidth of preconditioner for PCG. By default, diagonal preconditioning is used (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations.
<code>TolPCG</code>	Termination tolerance on the PCG iteration.
<code>TypicalX</code>	Typical <code>x</code> values.

**Medium-Scale Algorithm Only.** These parameters are used only by the medium-scale algorithm:

<code>DerivativeCheck</code>	Compare user-supplied derivatives (gradients of the objective and constraints) to finite-differencing derivatives.
<code>DiffMaxChange</code>	Maximum change in variables for finite-difference gradients.
<code>DiffMinChange</code>	Minimum change in variables for finite-difference gradients.

## Examples

Find values of  $x$  that minimize  $f(x) = -x_1 x_2 x_3$ , starting at the point  $x = [10; 10; 10]$  and subject to the constraints

$$0 \leq x_1 + 2x_2 + 2x_3 \leq 72$$

First, write an M-file that returns a scalar value  $f$  of the function evaluated at  $x$ .

```
function f = myfun(x)
f = -x(1) * x(2) * x(3);
```

Then rewrite the constraints as both less than or equal to a constant,

$$\begin{aligned} -x_1 - 2x_2 - 2x_3 &\leq 0 \\ x_1 + 2x_2 + 2x_3 &\leq 72 \end{aligned}$$

Since both constraints are linear, formulate them as the matrix inequality  $A \cdot x \leq b$  where

$$A = \begin{bmatrix} -1 & -2 & -2 \\ 1 & 2 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 0 \\ 72 \end{bmatrix}$$

Next, supply a starting point and invoke an optimization routine.

```
x0 = [10; 10; 10]; % Starting guess at the solution
[x, fval] = fmincon(@myfun, x0, A, b)
```

After 66 function evaluations, the solution is

```
x =
24.0000
12.0000
12.0000
```

where the function value is

```
fval =
-3.4560e+03
```

and linear inequality constraints evaluate to be  $\leq 0$

```
A*x-b=
-72
0
```

## Notes

**Large-Scale Optimization.** To use the large-scale method, the gradient must be provided in `fun` (and the `GradObj` parameter is set to 'on'). A warning is given if no gradient is provided and the `LargeScale` parameter is not 'off'. The function `fmincon` permits  $g(x)$  to be an approximate gradient but this option is not recommended; the numerical behavior of most optimization codes is considerably more robust when the true gradient is used.

The large-scale method in `fmincon` is most effective when the matrix of second derivatives, i.e., the Hessian matrix  $H(x)$ , is also computed. However, evaluation of the true Hessian matrix is not required. For example, if you can supply the Hessian sparsity structure (using the `HessPattern` parameter in `options`), then `fmincon` computes a sparse finite-difference approximation to  $H(x)$ .

If  $x_0$  is not strictly feasible, `fmincon` chooses a new strictly feasible (centered) starting point.

If components of  $x$  have no upper (or lower) bounds, then `fmincon` prefers that the corresponding components of `ub` (or `lb`) be set to `Inf` (or `-Inf` for `lb`) as opposed to an arbitrary but very large positive (or negative in the case of lower bounds) number.

Several aspects of linearly constrained minimization should be noted:

- A dense (or fairly dense) column of matrix `Aeq` can result in considerable fill and computational cost.
- `fmincon` removes (numerically) linearly dependent rows in `Aeq`; however, this process involves repeated matrix factorizations and therefore can be costly if there are many dependencies.
- Each iteration involves a sparse least-squares solve with matrix

$$\overline{Aeq} = Aeq^T R^{-T}$$

where  $R^T$  is the Cholesky factor of the preconditioner. Therefore, there is a potential conflict between choosing an effective preconditioner and minimizing fill in  $\overline{Aeq}$ .

**Medium-Scale Optimization.** Better numerical results are likely if you specify equalities explicitly using `Aeq` and `beq`, instead of implicitly using `lb` and `ub`.

If equality constraints are present and dependent equalities are detected and removed in the quadratic subproblem, 'dependent' is printed under the `Procedures` heading (when you ask for output by setting the `Display` parameter to 'iter'). The dependent equalities are only removed when the equalities are consistent. If the system of equalities is not consistent, the subproblem is infeasible and 'infeasible' is printed under the `Procedures` heading.

## Algorithm

**Large-Scale Optimization.** By default `fmincon` will choose the large-scale algorithm *if* the user supplies the gradient in `fun` (and `GradObj` is 'on' in `options`) *and if only* upper and lower bounds exist *or only* linear equality constraints exist. This algorithm is a subspace trust region method and is based on the interior-reflective Newton method described in [1], [2]. Each iteration involves the approximate solution of a large linear system using the method of preconditioned conjugate gradients (PCG). See the trust-region and preconditioned conjugate gradient method descriptions in the [Large-Scale Algorithms](#) chapter.

**Medium-Scale Optimization.** `fmincon` uses a Sequential Quadratic Programming (SQP) method. In this method, a Quadratic Programming (QP) subproblem is solved at each iteration. An estimate of the Hessian of the Lagrangian is updated at each iteration using the BFGS formula (see [fminunc](#), references [7], [8]).

A line search is performed using a merit function similar to that proposed by [4], [5], and [6]. The QP subproblem is solved using an active set strategy similar to that described in [3]. A full description of this algorithm is found in [Constrained Optimization](#) in "Introduction to Algorithms."

See also [SQP Implementation](#) in "Introduction to Algorithms" for more details on the algorithm used.

## Diagnostics

**Large-Scale Optimization.** The large-scale code does not allow equal upper and lower bounds. For example if `lb(2)==ub(2)` , then `fmincon` gives the error

Equal upper and lower bounds not permitted in this large-scale method.

Use equality constraints and the medium-scale method instead.

If you only have equality constraints you can still use the large-scale method. But if you have both equalities and bounds, you must use the medium-scale method.

## Limitations

The function to be minimized and the constraints must both be continuous. `fmincon` may only give local solutions.

When the problem is infeasible, `fmincon` attempts to minimize the maximum constraint value.

The objective function and constraint function must be real-valued, that is they cannot return complex values.

**Large-Scale Optimization.** To use the large-scale algorithm, the user must supply the gradient in `fun` (and `GradObj` must be set 'on' in `options`) , and only upper and lower bounds constraints may be specified, or only linear equality constraints must exist and `Aeq` cannot have more rows than columns. `Aeq` is typically sparse. See [Table 2-4, Large-Scale Problem Coverage and Requirements](#), for more information on what problem formulations are covered and what information must be provided.

Currently, if the analytical gradient is provided in `fun`, the `options` parameter `DerivativeCheck` cannot be used with the large-scale method to compare the analytic gradient to the finite-difference gradient. Instead, use the medium-scale method to check the derivative with `options` parameter `MaxIter` set to 0 iterations. Then run the problem with the large-scale method.

## See Also

[`@\(function\_handle\)`](#), [fminbnd](#), [fminsearch](#), [fminunc](#), [optimset](#)

## References

- [1] Coleman, T.F. and Y. Li, "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds," *SIAM Journal on Optimization*, Vol. 6, pp. 418-445, 1996.

- [2] Coleman, T.F. and Y. Li, "On the Convergence of Reflective Newton Methods for Large-Scale Nonlinear Minimization Subject to Bounds," *Mathematical Programming*, Vol. 67, Number 2, pp. 189-224, 1994.
- [3] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, Academic Press, London, 1981.
- [4] Han, S.P., "A Globally Convergent Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.
- [5] Powell, M.J.D., "A Fast Algorithm for Nonlineary Constrained Optimization Calculations," *Numerical Analysis*, ed. G.A. Watson, *Lecture Notes in Mathematics*, Springer Verlag, Vol. 630, 1978.
- [6] Powell, M.J.D., "The Convergence of Variable Metric Methods For Nonlinearly Constrained Optimization Calculations," *Nonlinear Programming 3* (O.L. Mangasarian, R.R. Meyer, and S.M. Robinson, eds.) Academic Press, 1978.

 fminbnd 

## A.2 Python - `scipy.optimize.minimize`

## 6.18.4 Optimization

### Scalar Functions Optimization

---

<code>minimize_scalar(fun[, bracket, bounds, ...])</code>	Minimization of scalar function of one variable.
---	--

---

#### `scipy.optimize.minimize_scalar`

```
scipy.optimize.minimize_scalar(fun, bracket=None, bounds=None, args=(), method='brent',
                               tol=None, options=None)
```

Minimization of scalar function of one variable.

#### Parameters

<b>fun</b>	[callable] Objective function. Scalar function, must return a scalar.
<b>bracket</b>	[sequence, optional] For methods ‘brent’ and ‘golden’, <code>bracket</code> defines the bracketing interval and can either have three items ( <code>a</code> , <code>b</code> , <code>c</code> ) so that $a < b < c$ and $\text{fun}(b) < \text{fun}(a)$ , $\text{fun}(c)$ or two items <code>a</code> and <code>c</code> which are assumed to be a starting interval for a downhill bracket search (see <code>bracket</code> ); it doesn’t always mean that the obtained solution will satisfy $a \leq x \leq c$ .
<b>bounds</b>	[sequence, optional] For method ‘bounded’, <code>bounds</code> is mandatory and must have two items corresponding to the optimization bounds.
<b>args</b>	[tuple, optional] Extra arguments passed to the objective function.
<b>method</b>	[str or callable, optional] Type of solver. Should be one of: <ul style="list-style-type: none"><li>• ‘Brent’ (<a href="#">see here</a>)</li><li>• ‘Bounded’ (<a href="#">see here</a>)</li><li>• ‘Golden’ (<a href="#">see here</a>)</li><li>• custom - a callable object (added in version 0.14.0), see below</li></ul>
<b>tol</b>	[float, optional] Tolerance for termination. For detailed control, use solver-specific options.
<b>options</b>	[dict, optional] A dictionary of solver options. <ul style="list-style-type: none"><li><b>maxiter</b> [int] Maximum number of iterations to perform.</li><li><b>disp</b> [bool] Set to True to print convergence messages.</li></ul> See <code>show_options</code> for solver-specific options.

#### Returns

<b>res</b>	[OptimizeResult] The optimization result represented as a <code>OptimizeResult</code> object. Important attributes are: <code>x</code> the solution array, <code>success</code> a Boolean flag indicating if the optimizer exited successfully and <code>message</code> which describes the cause of the termination. See <code>OptimizeResult</code> for a description of other attributes.
------------	--

#### See also:

##### `minimize`

Interface to minimization algorithms for scalar multivariate functions

##### `show_options`

Additional options accepted by the solvers

#### Notes

This section describes the available solvers that can be selected by the ‘method’ parameter. The default method is *Brent*.

Method *Brent* uses Brent's algorithm to find a local minimum. The algorithm uses inverse parabolic interpolation when possible to speed up convergence of the golden section method.

Method *Golden* uses the golden section search technique. It uses analog of the bisection method to decrease the bracketed interval. It is usually preferable to use the *Brent* method.

Method *Bounded* can perform bounded minimization. It uses the Brent method to find a local minimum in the interval  $x_1 < x_{opt} < x_2$ .

### Custom minimizers

It may be useful to pass a custom minimization method, for example when using some library frontend to `minimize_scalar`. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as `bracket`, `tol`, etc.), except the `options` dict, which has its contents also passed as `method` parameters pair by pair. The method shall return an `OptimizeResult` object.

The provided `method` callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the `scipy.optimize` tutorial.

New in version 0.11.0.

## Examples

Consider the problem of minimizing the following function.

```
>>> def f(x):
...     return (x - 2) * x * (x + 2)**2
```

Using the *Brent* method, we find the local minimum as:

```
>>> from scipy.optimize import minimize_scalar
>>> res = minimize_scalar(f)
>>> res.x
1.28077640403
```

Using the *Bounded* method, we find a local minimum with specified bounds as:

```
>>> res = minimize_scalar(f, bounds=(-3, -1), method='bounded')
>>> res.x
-2.0000002026
```

The `minimize_scalar` function supports the following methods:

### `minimize_scalar(method='brent')`

```
scipy.optimize.minimize_scalar(fun, args=(), method='brent', tol=None, options={'func': None,
'brack': None, 'xtol': 1.48e-08, 'maxiter': 500})
```

#### See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize_scalar`

#### *Options*

- |                      |  |
|----------------------|--|
| <code>maxiter</code> | [int] Maximum number of iterations to perform.                           |
| <code>xtol</code>    | [float] Relative error in solution $x_{opt}$ acceptable for convergence. |

## Notes

Uses inverse parabolic interpolation when possible to speed up convergence of golden section method.

### `minimize_scalar(method='bounded')`

```
scipy.optimize.minimize_scalar(fun, bounds=None, args=(), method='bounded', tol=None, options={'func': None, 'xatol': 1e-05, 'maxiter': 500, 'disp': 0})
```

#### See also:

For documentation for the rest of the parameters, see [`scipy.optimize.minimize\_scalar`](#)

#### *Options*

**maxiter** [int] Maximum number of iterations to perform.  
**disp**: int, optional

#### If non-zero, print messages.

0 : no message printing. 1 : non-convergence notification messages only. 2 : print a message on convergence too. 3 : print iteration results.

**xatol** [float] Absolute error in solution  $x_{opt}$  acceptable for convergence.

### `minimize_scalar(method='golden')`

```
scipy.optimize.minimize_scalar(fun, args=(), method='golden', tol=None, options={'func': None, 'brack': None, 'xtol': 1.4901161193847656e-08, 'maxiter': 5000})
```

#### See also:

For documentation for the rest of the parameters, see [`scipy.optimize.minimize\_scalar`](#)

#### *Options*

**maxiter** [int] Maximum number of iterations to perform.  
**xtol** [float] Relative error in solution  $x_{opt}$  acceptable for convergence.

## Local (Multivariate) Optimization

---

`minimize(fun, x0[, args, method, jac, hess, ...])` Minimization of scalar function of one or more variables.

### `scipy.optimize.minimize`

```
scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None)
```

Minimization of scalar function of one or more variables.

#### *Parameters*

**fun** [callable] The objective function to be minimized.  
     $\text{fun}(x, *args) \rightarrow \text{float}$   
where  $x$  is an 1-D array with shape (n,) and  $args$  is a tuple of the fixed parameters needed to completely specify the function.  
**x0** [ndarray, shape (n,)] Initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables.  
**args** [tuple, optional] Extra arguments passed to the objective function and its derivatives ( $fun$ ,  $jac$  and  $hess$  functions).

---

<b>method</b>	[str or callable, optional] Type of solver. Should be one of <ul style="list-style-type: none"><li>• ‘Nelder-Mead’ (<a href="#">see here</a>)</li><li>• ‘Powell’ (<a href="#">see here</a>)</li><li>• ‘CG’ (<a href="#">see here</a>)</li><li>• ‘BFGS’ (<a href="#">see here</a>)</li><li>• ‘Newton-CG’ (<a href="#">see here</a>)</li><li>• ‘L-BFGS-B’ (<a href="#">see here</a>)</li><li>• ‘TNC’ (<a href="#">see here</a>)</li><li>• ‘COBYLA’ (<a href="#">see here</a>)</li><li>• ‘SLSQP’ (<a href="#">see here</a>)</li><li>• ‘trust-constr’ (<a href="#">see here</a>)</li><li>• ‘dogleg’ (<a href="#">see here</a>)</li><li>• ‘trust-ncg’ (<a href="#">see here</a>)</li><li>• ‘trust-exact’ (<a href="#">see here</a>)</li><li>• ‘trust-krylov’ (<a href="#">see here</a>)</li><li>• custom - a callable object (added in version 0.14.0), see below for description.</li></ul> If not given, chosen to be one of BFGS, L-BFGS-B, SLSQP, depending if the problem has constraints or bounds.
<b>jac</b>	[{callable, ‘2-point’, ‘3-point’, ‘cs’, bool}, optional] Method for computing the gradient vector. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is a callable, it should be a function that returns the gradient vector:  <code>jac(x, *args) -&gt; array_like, shape (n,)</code> where x is an array with shape (n,) and args is a tuple with the fixed parameters. Alternatively, the keywords {‘2-point’, ‘3-point’, ‘cs’} select a finite difference scheme for numerical estimation of the gradient. Options ‘3-point’ and ‘cs’ are available only to ‘trust-constr’. If jac is a Boolean and is True, fun is assumed to return the gradient along with the objective function. If False, the gradient will be estimated using ‘2-point’ finite difference estimation.
<b>hess</b>	[{callable, ‘2-point’, ‘3-point’, ‘cs’, HessianUpdateStrategy}, optional] Method for computing the Hessian matrix. Only for Newton-CG, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is callable, it should return the Hessian matrix:  <code>hess(x, *args) -&gt; {LinearOperator, spmatrix, array}, (n, n)</code> where x is a (n,) ndarray and args is a tuple with the fixed parameters. LinearOperator and sparse matrix returns are allowed only for ‘trust-constr’ method. Alternatively, the keywords {‘2-point’, ‘3-point’, ‘cs’} select a finite difference scheme for numerical estimation. Or, objects implementing <a href="#">HessianUpdateStrategy</a> interface can be used to approximate the Hessian. Available quasi-Newton methods implementing this interface are: <ul style="list-style-type: none"><li>• <a href="#">BFGS</a>;</li><li>• <a href="#">SR1</a>.</li></ul> Whenever the gradient is estimated via finite-differences, the Hessian cannot be estimated with options {‘2-point’, ‘3-point’, ‘cs’} and needs to be estimated using one of the quasi-Newton strategies. Finite-difference options {‘2-point’, ‘3-point’, ‘cs’} and <a href="#">HessianUpdateStrategy</a> are available only for ‘trust-constr’ method.
<b>hessp</b>	[callable, optional] Hessian of objective function times an arbitrary vector p. Only for Newton-CG, trust-ncg, trust-krylov, trust-constr. Only one of hessp or hess needs to be given. If hess is provided, then hessp will be ignored. hessp must compute the Hessian times an arbitrary vector:  <code>hessp(x, p, *args) -&gt; ndarray shape (n,)</code> where x is a (n,) ndarray, p is an arbitrary vector with dimension (n,) and args is a tuple with the fixed parameters.
<b>bounds</b>	[sequence or <a href="#">Bounds</a> , optional] Bounds on variables for L-BFGS-B, TNC, SLSQP and trust-constr methods. There are two ways to specify the bounds: <ol style="list-style-type: none"><li>1. Instance of <a href="#">Bounds</a> class.</li></ol>

2. Sequence of `(min, max)` pairs for each element in `x`. `None` is used to specify no bound.

**constraints**

[{Constraint, dict} or List of {Constraint, dict}, optional] Constraints definition (only for COBYLA, SLSQP and trust-constr). Constraints for ‘trust-constr’ are defined as a single object or a list of objects specifying constraints to the optimization problem. Available constraints are:

- `LinearConstraint`
- `NonlinearConstraint`

Constraints for COBYLA, SLSQP are defined as a list of dictionaries. Each dictionary with fields:

<code>type</code>	[str] Constraint type: ‘eq’ for equality, ‘ineq’ for inequality.
<code>fun</code>	[callable] The function defining the constraint.
<code>jac</code>	[callable, optional] The Jacobian of <code>fun</code> (only for SLSQP).
<code>args</code>	[sequence, optional] Extra arguments to be passed to the function and Jacobian.

Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative. Note that COBYLA only supports inequality constraints.

**tol****options**

[float, optional] Tolerance for termination. For detailed control, use solver-specific options.

[dict, optional] A dictionary of solver options. All methods accept the following generic options:

<code>maxiter</code>	[int] Maximum number of iterations to perform.
<code>disp</code>	[bool] Set to True to print convergence messages.

For method-specific options, see `show_options`.

**callback**

[callable, optional] Called after each iteration. For ‘trust-constr’ it is a callable with the signature:

```
callback(xk, OptimizeResult state) -> bool
```

where `xk` is the current parameter vector. and `state` is an `OptimizeResult` object, with the same fields as the ones from the return. If `callback` returns True the algorithm execution is terminated. For all the other methods, the signature is:

```
callback(xk)
```

where `xk` is the current parameter vector.

**Returns****res**

[`OptimizeResult`] The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See `OptimizeResult` for a description of other attributes.

**See also:****`minimize_scalar`**

Interface to minimization algorithms for scalar univariate functions

**`show_options`**

Additional options accepted by the solvers

**Notes**

This section describes the available solvers that can be selected by the ‘method’ parameter. The default method is `BFGS`.

**Unconstrained minimization**

Method [Nelder-Mead](#) uses the Simplex algorithm [1], [2]. This algorithm is robust in many applications. However, if numerical computation of derivative can be trusted, other algorithms using the first and/or second derivatives information might be preferred for their better performance in general.

Method [Powell](#) is a modification of Powell's method [3], [4] which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken.

Method [CG](#) uses a nonlinear conjugate gradient algorithm by Polak and Ribiére, a variant of the Fletcher-Reeves method described in [5] pp. 120-122. Only the first derivatives are used.

Method [BFGS](#) uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5] pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess\_inv* in the *OptimizeResult* object.

Method [Newton-CG](#) uses a Newton-CG algorithm [5] pp. 168 (also known as the truncated Newton method). It uses a CG method to compute the search direction. See also *TNC* method for a box-constrained minimization with a similar algorithm. Suitable for large-scale problems.

Method [dogleg](#) uses the dog-leg trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method [trust-ncg](#) uses the Newton conjugate gradient trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems.

Method [trust-krylov](#) uses the Newton GLTR trust-region algorithm [14], [15] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems. On indefinite problems it requires usually less iterations than the *trust-ncg* method and is recommended for medium and large-scale problems.

Method [trust-exact](#) is a trust-region method for unconstrained minimization in which quadratic subproblems are solved almost exactly [13]. This algorithm requires the gradient and the Hessian (which is *not* required to be positive definite). It is, in many situations, the Newton method to converge in fewer iterations and the most recommended for small and medium-size problems.

### Bound-Constrained minimization

Method [L-BFGS-B](#) uses the L-BFGS-B algorithm [6], [7] for bound constrained minimization.

Method [TNC](#) uses a truncated Newton algorithm [5], [8] to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

### Constrained Minimization

Method [COBYLA](#) uses the Constrained Optimization BY Linear Approximation (COBYLA) method [9], [10], [11]. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm. The constraints functions ‘fun’ may return either a single number or an array or list of numbers.

Method [SLSQP](#) uses Sequential Least Squares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

Method [trust-constr](#) is a trust-region algorithm for constrained optimization. It switches between two implementations depending on the problem definition. It is the most versatile constrained minimization algorithm implemented in SciPy and the most appropriate for large-scale problems. For equality constrained problems it is an

implementation of Byrd-Omojokun Trust-Region SQP method described in [17] and in [5], p. 549. When inequality constraints are imposed as well, it switches to the trust-region interior point method described in [16]. This interior point algorithm, in turn, solves inequality constraints by introducing slack variables and solving a sequence of equality-constrained barrier problems for progressively smaller values of the barrier parameter. The previously described equality constrained SQP method is used to solve the subproblems with increasing levels of accuracy as the iterate gets closer to a solution.

### Finite-Difference Options

For Method `trust-constr` the gradient and the Hessian may be approximated using three finite-difference schemes: {'2-point', '3-point', 'cs'}. The scheme 'cs' is, potentially, the most accurate but it requires the function to correctly handles complex inputs and to be differentiable in the complex plane. The scheme '3-point' is more accurate than '2-point' but requires twice as much operations.

### Custom minimizers

It may be useful to pass a custom minimization method, for example when using a frontend to this method such as `scipy.optimize.basinhopping` or a different library. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, x0, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as `callback`, `hess`, etc.), except the `options` dict, which has its contents also passed as `method` parameters pair by pair. Also, if `jac` has been passed as a bool type, `jac` and `fun` are mangled so that `fun` returns just the function values and `jac` is converted to a function returning the Jacobian. The method shall return an `OptimizeResult` object.

The provided `method` callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the `scipy.optimize` tutorial.

New in version 0.11.0.

## References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]

## Examples

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in `rosen` (resp. `rosen_der`, `rosen_hess`) in the `scipy.optimize`.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the *Nelder-Mead* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
```

Now using the *BFGS* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                 options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
      Current function value: 0.000000
```

(continues on next page)

(continued from previous page)

```

Iterations: 26
Function evaluations: 31
Gradient evaluations: 31
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377], ...
   # may vary
   [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
   [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
   [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
   [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,  1.53713523]])
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from [5]). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x: x[0] - 2 * x[1] + 2},
...          {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
...          {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
...                 constraints=cons)
```

It should converge to the theoretical solution (1.4, 1.7).

The `minimize` function supports the following methods:

### `minimize(method='Nelder-Mead')`

```
scipy.optimize.minimize(fun, x0, args=(), method='Nelder-Mead', tol=None, callback=None, options={'func': None, 'maxiter': None, 'maxfev': None, 'disp': False, 'return_all': False, 'initial_simplex': None, 'xatol': 0.0001, 'fatol': 0.0001, 'adaptive': False})
```

Minimization of scalar function of one or more variables using the Nelder-Mead algorithm.

#### See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

#### *Options*

<code>disp</code>	[bool]	Set to True to print convergence messages.
<code>maxiter</code> , <code>maxfev</code>		

[int] Maximum allowed number of iterations and function evaluations. Will default to  $N*200$ , where  $N$  is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.

**initial\_simplex**

[array\_like of shape (N + 1, N)] Initial simplex. If given, overrides *x0*. *initial\_simplex*[j, :] should contain the coordinates of the j-th vertex of the N+1 vertices in the simplex, where N is the dimension.

**xtol**

[float, optional] Absolute error in *xopt* between iterations that is acceptable for convergence.

**fatol**

[number, optional] Absolute error in *func(xopt)* between iterations that is acceptable for convergence.

**adaptive**

[bool, optional] Adapt algorithm parameters to dimensionality of problem. Useful for high-dimensional minimization [1].

## References

[1]

**minimize(method='Powell')**

```
scipy.optimize.minimize(fun, x0, args=(), method='Powell', tol=None, callback=None, options={'func': None, 'xtol': 0.0001, 'ftol': 0.0001, 'maxiter': None, 'maxfev': None, 'disp': False, 'direc': None, 'return_all': False})
```

Minimization of scalar function of one or more variables using the modified Powell algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options***disp**

[bool] Set to True to print convergence messages.

**xtol**

[float] Relative error in solution *xopt* acceptable for convergence.

**ftol**

[float] Relative error in *fun(xopt)* acceptable for convergence.

**maxiter, maxfev**

[int] Maximum allowed number of iterations and function evaluations. Will default to  $N*1000$ , where  $N$  is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.

**direc**

[ndarray] Initial set of direction vectors for the Powell method.

**minimize(method='CG')**

```
scipy.optimize.minimize(fun, x0, args=(), method='CG', jac=None, tol=None, callback=None, options={'gtol': 1e-05, 'norm': inf, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False})
```

Minimization of scalar function of one or more variables using the conjugate gradient algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options***disp**

[bool] Set to True to print convergence messages.

**maxiter**

[int] Maximum number of iterations to perform.

**gtol**

[float] Gradient norm must be less than *gtol* before successful termination.

**norm**

[float] Order of norm (Inf is max, -Inf is min).

**eps**

[float or ndarray] If *jac* is approximated, use this value for the step size.

**minimize(method='BFGS')**

```
scipy.optimize.minimize(fun, x0, args=(), method='BFGS', jac=None, tol=None, callback=None, options={'gtol': 1e-05, 'norm': inf, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False})
```

Minimization of scalar function of one or more variables using the BFGS algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

<b>disp</b>	[bool] Set to True to print convergence messages.
<b>maxiter</b>	[int] Maximum number of iterations to perform.
<b>gtol</b>	[float] Gradient norm must be less than <i>gtol</i> before successful termination.
<b>norm</b>	[float] Order of norm (Inf is max, -Inf is min).
<b>eps</b>	[float or ndarray] If <i>jac</i> is approximated, use this value for the step size.

**minimize(method='Newton-CG')**

```
scipy.optimize.minimize(fun, x0, args=(), method='Newton-CG', jac=None, hess=None, hessp=None, tol=None, callback=None, options={'xtol': 1e-05, 'eps': 1.4901161193847656e-08, 'maxiter': None, 'disp': False, 'return_all': False})
```

Minimization of scalar function of one or more variables using the Newton-CG algorithm.

Note that the *jac* parameter (Jacobian) is required.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

<b>disp</b>	[bool] Set to True to print convergence messages.
<b>xtol</b>	[float] Average relative error in solution <i>xopt</i> acceptable for convergence.
<b>maxiter</b>	[int] Maximum number of iterations to perform.
<b>eps</b>	[float or ndarray] If <i>jac</i> is approximated, use this value for the step size.

**minimize(method='L-BFGS-B')**

```
scipy.optimize.minimize(fun, x0, args=(), method='L-BFGS-B', jac=None, bounds=None, tol=None, callback=None, options={'disp': None, 'maxcor': 10, 'ftol': 2.220446049250313e-09, 'gtol': 1e-05, 'eps': 1e-08, 'maxfun': 15000, 'maxiter': 15000, 'iprint': -1, 'maxls': 20})
```

Minimize a scalar function of one or more variables using the L-BFGS-B algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

<b>disp</b>	[None or int] If <i>disp</i> is <i>None</i> (the default), then the supplied version of <i>iprint</i> is used. If <i>disp</i> is not <i>None</i> , then it overrides the supplied version of <i>iprint</i> with the behaviour you outlined.
<b>maxcor</b>	[int] The maximum number of variable metric corrections used to define the limited memory matrix. (The limited memory BFGS method does not store the full hessian but uses this many terms in an approximation to it.)
<b>ftol</b>	[float] The iteration stops when $(f^k - f^{k+1}) / \max\{ f^k ,  f^{k+1} , 1\} \leq ftol$ .

<b>gtol</b>	[float] The iteration will stop when $\max\{ \text{proj } g_i  \mid i = 1, \dots, n\} \leq gtol$ where $g_i$ is the $i$ -th component of the projected gradient.
<b>eps</b>	[float] Step size used for numerical approximation of the jacobian.
<b>maxfun</b>	[int] Maximum number of function evaluations.
<b>maxiter</b>	[int] Maximum number of iterations.
<b>maxls</b>	[int, optional] Maximum number of line search steps (per iteration). Default is 20.

## Notes

The option *ftol* is exposed via the `scipy.optimize.minimize` interface, but calling `scipy.optimize.fmin_l_bfgs_b` directly exposes *factr*. The relationship between the two is  $ftol = factr * \text{numpy.finfo(float).eps}$ . I.e., *factr* multiplies the default machine floating-point precision to arrive at *ftol*.

### `minimize(method='TNC')`

```
scipy.optimize.minimize(fun, x0, args=(), method='TNC', jac=None, bounds=None, tol=None, call-
    back=None, options={'eps': 1e-08, 'scale': None, 'offset': None, 'msg_num':
        None, 'maxCGit': -1, 'maxiter': None, 'eta': -1, 'stepmx': 0, 'accuracy': 0, 'min-
        fev': 0, 'ftol': -1, 'xtol': -1, 'gtol': -1, 'rescale': -1, 'disp': False})
```

Minimize a scalar function of one or more variables using a truncated Newton (TNC) algorithm.

#### See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

#### Options

<b>eps</b>	[float] Step size used for numerical approximation of the jacobian.
<b>scale</b>	[list of floats] Scaling factors to apply to each variable. If None, the factors are up-low for interval bounded variables and 1+lx] fo the others. Defaults to None
<b>offset</b>	[float] Value to subtract from each variable. If None, the offsets are (up+low)/2 for interval bounded variables and x for the others.
<b>disp</b>	[bool] Set to True to print convergence messages.
<b>maxCGit</b>	[int] Maximum number of hessian*vector evaluations per main iteration. If maxCGit == 0, the direction chosen is -gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -1.
<b>maxiter</b>	[int] Maximum number of function evaluation. if None, <i>maxiter</i> is set to max(100, 10*len(x0)). Defaults to None.
<b>eta</b>	[float] Severity of the line search. if < 0 or > 1, set to 0.25. Defaults to -1.
<b>stepmx</b>	[float] Maximum step for the line search. May be increased during call. If too small, it will be set to 10.0. Defaults to 0.
<b>accuracy</b>	[float] Relative precision for finite difference calculations. If <= machine_precision, set to sqrt(machine_precision). Defaults to 0.
<b>minfev</b>	[float] Minimum function value estimate. Defaults to 0.
<b>ftol</b>	[float] Precision goal for the value of f in the stopping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -1.
<b>xtol</b>	[float] Precision goal for the value of x in the stopping criterion (after applying x scaling factors). If xtol < 0.0, xtol is set to sqrt(machine_precision). Defaults to -1.
<b>gtol</b>	[float] Precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors). If gtol < 0.0, gtol is set to 1e-2 * sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.
<b>rescale</b>	[float] Scaling factor (in log10) used to trigger f value rescaling. If 0, rescale at each iteration. If a large value, never rescale. If < 0, rescale is set to 1.3.

**minimize(method='COBYLA')**

```
scipy.optimize.minimize(fun, x0, args=(), method='COBYLA', constraints=(),
                       tol=None, callback=None, options={'rhobeg': 1.0, 'maxiter': 1000, 'disp': False, 'catol': 0.0002})
```

Minimize a scalar function of one or more variables using the Constrained Optimization BY Linear Approximation (COBYLA) algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

<b>rhobeg</b>	[float] Reasonable initial changes to the variables.
<b>tol</b>	[float] Final accuracy in the optimization (not precisely guaranteed). This is a lower bound on the size of the trust region.
<b>disp</b>	[bool] Set to True to print convergence messages. If False, <i>verbosity</i> is ignored as set to 0.
<b>maxiter</b>	[int] Maximum number of function evaluations.
<b>catol</b>	[float] Tolerance (absolute) for constraint violations

**minimize(method='SLSQP')**

```
scipy.optimize.minimize(fun, x0, args=(), method='SLSQP', jac=None, bounds=None, constraints=(),
                       tol=None, callback=None, options={'func': None, 'maxiter': 100, 'ftol': 1e-06,
                                                       'iprint': 1, 'disp': False, 'eps': 1.4901161193847656e-08})
```

Minimize a scalar function of one or more variables using Sequential Least SQuares Programming (SLSQP).

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

<b>ftol</b>	[float] Precision goal for the value of f in the stopping criterion.
<b>eps</b>	[float] Step size used for numerical approximation of the Jacobian.
<b>disp</b>	[bool] Set to True to print convergence messages. If False, <i>verbosity</i> is ignored and set to 0.
<b>maxiter</b>	[int] Maximum number of iterations.

**minimize(method='trust-constr')**

```
scipy.optimize.minimize(fun, x0, args=(), method='trust-constr', hess=None, hessp=None,
                       bounds=None, constraints=(), tol=None, callback=None, options={'grad': None, 'gtol': 1e-08, 'gtol': 1e-08, 'barrier_tol': 1e-08, 'sparse_jacobian': None, 'maxiter': 1000, 'verbose': 0, 'finite_diff_rel_step': None, 'initial_constr_penalty': 1.0, 'initial_tr_radius': 1.0, 'initial_barrier_parameter': 0.1, 'initial_barrier_tolerance': 0.1, 'factorization_method': None, 'disp': False})
```

Minimize a scalar function subject to constraints.

*Parameters*

<b>gtol</b>	[float, optional] Tolerance for termination by the norm of the Lagrangian gradient. The algorithm will terminate when both the infinity norm (i.e. max abs value) of the Lagrangian gradient and the constraint violation are smaller than <code>gtol</code> . Default is 1e-8.
<b>xtol</b>	[float, optional] Tolerance for termination by the change of the independent variable. The algorithm will terminate when <code>tr_radius &lt; xtol</code> , where <code>tr_radius</code> is the radius of the trust region used in the algorithm. Default is 1e-8.

**barrier\_tol**

[float, optional] Threshold on the barrier parameter for the algorithm termination. When inequality constraints are present the algorithm will terminate only when the barrier parameter is less than *barrier\_tol*. Default is 1e-8.

**sparse\_jacobian**

[{bool, None}, optional] Determines how to represent Jacobians of the constraints. If bool, then Jacobians of all the constraints will be converted to the corresponding format. If None (default), then Jacobians won't be converted, but the algorithm can proceed only if they all have the same format.

**initial\_tr\_radius: float, optional**

Initial trust radius. The trust radius gives the maximum distance between solution points in consecutive iterations. It reflects the trust the algorithm puts in the local approximation of the optimization problem. For an accurate local approximation the trust-region should be large and for an approximation valid only close to the current point it should be a small one. The trust radius is automatically updated throughout the optimization process, with *initial\_tr\_radius* being its initial value. Default is 1 (recommended in [1], p. 19).

**initial\_constr\_penalty**

[float, optional] Initial constraints penalty parameter. The penalty parameter is used for balancing the requirements of decreasing the objective function and satisfying the constraints. It is used for defining the merit function:  $\text{merit\_function}(x) = \text{fun}(x) + \text{constr\_penalty} * \text{constr\_norm\_l2}(x)$ , where *constr\_norm\_l2(x)* is the l2 norm of a vector containing all the constraints. The merit function is used for accepting or rejecting trial points and *constr\_penalty* weights the two conflicting goals of reducing objective function and constraints. The penalty is automatically updated throughout the optimization process, with *initial\_constr\_penalty* being its initial value. Default is 1 (recommended in [1], p 19).

**initial\_barrier\_parameter, initial\_barrier\_tolerance: float, optional**

Initial barrier parameter and initial tolerance for the barrier subproblem. Both are used only when inequality constraints are present. For dealing with optimization problems  $\min_x f(x)$  subject to inequality constraints  $c(x) \leq 0$  the algorithm introduces slack variables, solving the problem  $\min_{(x,s)} f(x) + \text{barrier\_parameter} * \sum(\ln(s))$  subject to the equality constraints  $c(x) + s = 0$  instead of the original problem. This subproblem is solved for increasing values of *barrier\_parameter* and with decreasing tolerances for the termination, starting with *initial\_barrier\_parameter* for the barrier parameter and *initial\_barrier\_tolerance* for the barrier subproblem barrier. Default is 0.1 for both values (recommended in [1] p. 19).

**factorization\_method**

[string or None, optional] Method to factorize the Jacobian of the constraints. Use None (default) for the auto selection or one of:

- ‘NormalEquation’ (requires scikit-sparse)
- ‘AugmentedSystem’
- ‘QRFactorization’
- ‘SVDFactorization’

The methods ‘NormalEquation’ and ‘AugmentedSystem’ can be used only with sparse constraints. The projections required by the algorithm will be computed using, respectively, the normal equation and the augmented system approaches explained in [1]. ‘NormalEquation’ computes the Cholesky factorization of  $A^T A$  and ‘AugmentedSystem’ performs the LU factorization of an augmented system. They usually provide similar results. ‘AugmentedSystem’ is used by default for sparse matrices.

The methods ‘QRFactorization’ and ‘SVDFactorization’ can be used only with dense constraints. They compute the required projections using, respectively, QR and SVD factorizations. The ‘SVDFactorization’ method can cope with Jacobian matrices with deficient row

rank and will be used whenever other factorization methods fail (which may imply the conversion of sparse matrices to a dense format when required). By default ‘QRFactorization’ is used for dense matrices.

<b>finite_diff_rel_step</b>	[None or array_like, optional] Relative step size for the finite difference approximation.
<b>maxiter</b>	[int, optional] Maximum number of algorithm iterations. Default is 1000.
<b>verbose</b>	[{0, 1, 2}, optional] Level of algorithm’s verbosity: <ul style="list-style-type: none"> <li>• 0 (default) : work silently.</li> <li>• 1 : display a termination report.</li> <li>• 2 : display progress during iterations.</li> <li>• 3 : display progress during iterations (more complete report).</li> </ul>
<b>disp</b>	[bool, optional] If True (default) then <i>verbose</i> will be set to 1 if it was 0.

#### Returns

‘OptimizeResult’ with the fields documented below. Note the following:

1. All values corresponding to the constraints are ordered as they were passed to the solver. And values corresponding to *bounds* constraints are put *after* other constraints.
2. All numbers of function, Jacobian or Hessian evaluations correspond to numbers of actual Python function calls. It means, for example, that if a Jacobian is estimated by finite differences then the number of Jacobian evaluations will be zero and the number of function evaluations will be incremented by all calls during the finite difference estimation.

<b>x</b>	[ndarray, shape (n,)] Solution found.
<b>optimality</b>	[float] Infinity norm of the Lagrangian gradient at the solution.
<b>constr_violation</b>	[float] Maximum constraint violation at the solution.
<b>fun</b>	[float] Objective function at the solution.
<b>grad</b>	[ndarray, shape (n,)] Gradient of the objective function at the solution.
<b>lagrangian_grad</b>	[ndarray, shape (n,)] Gradient of the Lagrangian function at the solution.
<b>nit</b>	[int] Total number of iterations.
<b>nfev</b>	[integer] Number of the objective function evaluations.
<b>ngev</b>	[integer] Number of the objective function gradient evaluations.
<b>nhev</b>	[integer] Number of the objective function Hessian evaluations.
<b>cg_niter</b>	[int] Total number of the conjugate gradient method iterations.
<b>method</b>	[{'equality_constrained_sqp', 'tr_interior_point'}] Optimization method used.
<b>constr</b>	[list of ndarray] List of constraint values at the solution.
<b>jac</b>	[list of {ndarray, sparse matrix}] List of the Jacobian matrices of the constraints at the solution.
<b>v</b>	[list of ndarray] List of the Lagrange multipliers for the constraints at the solution. For an inequality constraint a positive multiplier means that the upper bound is active, a negative multiplier means that the lower bound is active and if a multiplier is zero it means the constraint is not active.
<b>constr_nfev</b>	[list of int] Number of constraint evaluations for each of the constraints.
<b>constr_njев</b>	[list of int] Number of Jacobian matrix evaluations for each of the constraints.
<b>constr_nhev</b>	[list of int] Number of Hessian evaluations for each of the constraints.
<b>tr_radius</b>	[float] Radius of the trust region at the last iteration.
<b>constr_penalty</b>	[float] Penalty parameter at the last iteration, see <i>initial_constr_penalty</i> .

**barrier\_tolerance**

[float] Tolerance for the barrier subproblem at the last iteration. Only for problems with inequality constraints.

**barrier\_parameter**

[float] Barrier parameter at the last iteration. Only for problems with inequality constraints.

**execution\_time**

[float] Total execution time.

**message**

[str] Termination message.

**status**

[{0, 1, 2, 3}] Termination status:

- 0 : The maximum number of function evaluations is exceeded.
- 1 :  $gtol$  termination condition is satisfied.
- 2 :  $xtol$  termination condition is satisfied.
- 3 :  $callback$  function requested termination.

**cg\_stop\_cond**

[int] Reason for CG subproblem termination at the last iteration:

- 0 : CG subproblem not evaluated.
- 1 : Iteration limit was reached.
- 2 : Reached the trust-region boundary.
- 3 : Negative curvature detected.
- 4 : Tolerance was satisfied.

## References

[1]

**minimize(method='dogleg')**

```
scipy.optimize.minimize(fun, x0, args=(), method='dogleg', jac=None, hess=None, tol=None, call-
back=None, options={})
```

Minimization of scalar function of one or more variables using the dog-leg trust-region algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options***initial\_trust\_radius**

[float] Initial trust-region radius.

**max\_trust\_radius**

[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

**eta**

[float] Trust region related acceptance stringency for proposed steps.

**gtol**

[float] Gradient norm must be less than  $gtol$  before successful termination.

**minimize(method='trust-ncg')**

```
scipy.optimize.minimize(fun, x0, args=(), method='trust-ncg', jac=None, hess=None, hessp=None,
tol=None, callback=None, options={})
```

Minimization of scalar function of one or more variables using the Newton conjugate gradient trust-region algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

**initial\_trust\_radius**

[float] Initial trust-region radius.

**max\_trust\_radius**

[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

**eta** [float] Trust region related acceptance stringency for proposed steps.

**gtol** [float] Gradient norm must be less than *gtol* before successful termination.

**minimize(method='trust-krylov')**

```
scipy.optimize.minimize(fun, x0, args=(), method='trust-krylov', jac=None, hess=None,
                       tol=None, callback=None, options={'inexact': True})
```

Minimization of a scalar function of one or more variables using a nearly exact trust-region algorithm that only requires matrix vector products with the hessian matrix.

New in version 1.0.0.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options*

**inexact** [bool, optional] Accuracy to solve subproblems. If True requires less nonlinear iterations, but more vector products.

**minimize(method='trust-exact')**

```
scipy.optimize.minimize(fun, x0, args=(), method='trust-exact', jac=None, hess=None, tol=None, call-
                        back=None, options={})
```

Minimization of scalar function of one or more variables using a nearly exact trust-region algorithm.

**See also:**

For documentation for the rest of the parameters, see [scipy.optimize.minimize](#)

*Options***initial\_tr\_radius**

[float] Initial trust-region radius.

**max\_tr\_radius**

[float] Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.

**eta** [float] Trust region related acceptance stringency for proposed steps.

**gtol** [float] Gradient norm must be less than *gtol* before successful termination.

Constraints are passed to *minimize* function as a single object or as a list of objects from the following classes:

<code>NonlinearConstraint(fun, lb, ub[, jac, ...])</code>	Nonlinear constraint on the variables.
<code>LinearConstraint(A, lb, ub[, keep_feasible])</code>	Linear constraint on the variables.

**scipy.optimize.NonlinearConstraint**

```
class scipy.optimize.NonlinearConstraint(fun, lb, ub, jac='2-point',
                                         hess=<scipy.optimize._hessian_update_strategy.BFGS
                                         object>, keep_feasible=False, f-
                                         nite_diff_rel_step=None, f-
                                         nite_diff_jac_sparsity=None)
```

Nonlinear constraint on the variables.