Introdução ao MATLAB para Engenharia

Afrânio Melo
afraeq@gmail.com

2017

MATLAB

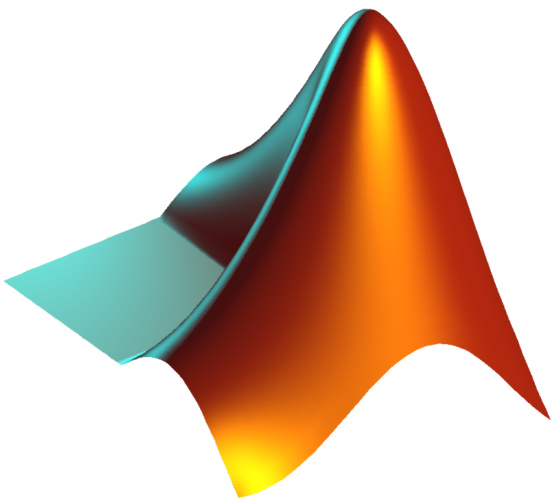


Figura 1: Logo do MATLAB.

Do que se trata?

- O MATLAB (MATrix LABoratory) é um ambiente de programação e computação científica de alto nível, que nos permite implementar algoritmos, manipular matrizes, plotar funções, analisar dados e, principalmente, resolver problemas numéricos complexos.
- O aprendizado de uma ferramenta do tipo, apesar de negligenciado em muitas grades curriculares, é *imprescindível* para qualquer estudante ou profissional que utiliza a Matemática para resolver problemas práticos.

Modo interativo

Uma calculadora científica

- O primeiro contato a ser feito com o MATLAB fica dentro da nossa zona de conforto: utilizá-lo como uma calculadora científica. É o seu primeiro contato com o *modo interativo* do MATLAB!
- Abra sua versão e procure a chamada *linha de comando*. Lá é o lugar onde podemos dar nossas ordens (comandos) ao MATLAB!
- Por exemplo, digite algumas operações básicas, como $2 + 2$, ou $2 * (3 + 4)/5$.
- Obs: se você não quiser que o MATLAB apresente o resultado do comando na tela, termine-o com ponto e vírgula. Ex: 2+2;

Bug!

Erro comum: esquecer o * (sinal de multiplicação).

Erro comum: não fechar os parênteses.

Funções pré-definidas

Bug!

Erro comum: escrever sen( ) no lugar de sin( ).

Erro comum: esquecer-se de que as funções trigonométricas trabalham com *radianos*.

Erro comum: achar que log (logaritmo natural ou neperiano) se refere ao log10 (logaritmo decimal).

Operando sobre valores

- Funções aceitam argumentos (*inputs*), os processam e fornecem resultados (*outputs*).
- Como veremos mais à frente, o MATLAB nos permite escrever nossas próprias funções. No entanto, existem algumas funções que já são parte integrante da linguagem!
- Muitas representam as funções matemáticas com as quais já estamos acostumados. Exemplo: insira sin(2), cos(1), log(1) e log10(0) na sua linha de comando.
- Você pode utilizar o comando *help* para tirar dúvidas sobre a natureza de uma função. Exemplo: insira help log na sua linha de comando.

Variáveis

Variáveis

- Uma *variável* é *um pedaço da memória do computador que guarda algum valor ou informação*.
- Por exemplo, digite o seguinte comando: a = 2.
- Note que provavelmente na sua tela há uma região chamada de *workspace*, que mostra todas as variáveis armazenadas na memória em dado momento.
- Para apagar uma variável, use o comando *clear*. Ex: clear a
- Para apagar todas as variáveis, use o comando *clear all*.

Bug!

Erro comum: sobrescrever uma variável e esquecer-se disso.

Variáveis

Bug!

Erro comum: confundir os conceitos de *igualdade* e *atribuição*.

Operador atribuição

- Quando utilizamos o comando $a = 2$, fazemos uso do operador $=$ (*atribuição*). Apesar do símbolo sugerir, este operador *não* representa uma igualdade!
- O operador $=$ na verdade *atribui* o valor que está do lado direito (no caso, 2) à variável do lado esquerdo (no caso, a). Esta é regra é conhecida no jargão da área como *right-to-left rule*.
- A diferença fica clara quando utilizamos o comando $a = a + 2$. O que ele significa? Se o sinal $=$ fosse uma igualdade, o comando não faria sentido! Pense um pouco e o digite no MATLAB para conferir o resultado.
- Outra maneira de perceber o significado do operador é tentando usar o comando $a + b = 2$. Que mensagem de erro aparece?

Regras para nomes de variáveis

As variáveis são sensíveis a letras maiúsculas e minúsculas	`Itens, itens` e `ITENS` São entendidas como diferentes variáveis.
As variáveis podem possuir até 31 caracteres. Os caracteres além do 31° são ignorados.	`Oquevoceachadestenomedevariavel` Pode ser usado como nome de variável.
O nome da variável deve começar com uma letra, seguida de qualquer número, letra ou sublinhado.	`O_que_voce_acha_deste_nome` e `X51` podem ser utilizados como nome de variáveis.

Variáveis especiais do MATLAB

Variável	*Significado*
`ans`	Variável padrão usada para resultados.
`pi`	Razão entre o perímetro da circunferência e seu diâmetro.
`eps`	Precisão relativa da máquina.
`inf`	Infinito
`NaN nan`	Não numérico
`i j`	$i = j = \sqrt{-1}$
`nargin`	Número de argumentos de entrada de uma função.
`nargout`	Número de argumentos de saída de uma função.
`realmin`	Menor número real positivo utilizável pela máquina.
`realmax`	Maior número real positivo utilizável pela máquina.

Keywords do MATLAB

TABLE 1.2 Keywords Reserved Explicitly for the MATLAB Programming Language

`break`	`global`
`case`	`if`
`catch`	`otherwise`
`continue`	`persistent`
`else`	`return`
`elseif`	`switch`
`end`	`try`
`for`	`while`
`function`	

Precisão e representação numérica

Precisão

- A precisão é a quantidade de casas decimais usadas para representar um número em um computador.
- Todo computador tem uma precisão finita!! Para verificar isto, digite o comando $1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2$. Qual o resultado?
- E para o comando $1 - 5 * 0.2$?
- $1/5$ não pode ser representado de maneira exata na base binária, usada pelo computador (da mesma maneira que $1/3$ não pode ser representado de maneira exata na base decimal). Sucessivas aproximações, portanto, levam a acumulação de erros!

Bug!

Erro comum: esquecer-se de que o computador tem uma precisão finita e que inevitavelmente faz aproximações.

Precisão e representação numérica

Bug!

Erro comum: ignorar as casas decimais não mostradas na tela, quando elas são importantes.

Comando format

- O MATLAB, implicitamente, realiza as operações utilizando toda a precisão da máquina (em geral, 16 casas decimais). No entanto, ele não costuma mostrar todas as casas decimais de um número na tela.
- Para controlar a quantidade de casas decimais que o MATLAB lhe apresenta, utilize o comando *format*. Por exemplo, digite *format long* ou *format short* e verifique o efeito que eles têm na representação de seus números.

Uso do comando *format*

TABLE 1.1 Examples of the Command Window `format` Options

Option	Display number > 1	Display $0 <$ number < 1
`short`	444.4444	0.0044
`long`	4.444444444444445e+002	0.004444444444444
`short e`	4.4444e+002	4.4444e-003
`long e`	4.444444444444445e+002	4.444444444444444e-003
`short g`	444.44	0.0044444
`long g`	444.444444444444	0.0044444444444444
`short eng`	444.4444e+000	4.4444e-003
`long eng`	444.444444444444e+000	4.444444444444444e-003
`rational`	4000/9	1/225
`hex`	407bc71c71c71c72	3f723456789abcdf
`bank`	444.44	0.00

Vetores e matrizes

O maior poder do MATLAB

- A característica mais marcante do MATLAB, inclusive refletida em seu próprio nome, é a maneira intuitiva e conveniente com que lida com os conceitos de *vetores* e *matrizes*, provenientes da álgebra linear.
- O MATLAB permite que vetores e matrizes sejam definidos como variáveis, a partir da linha de comando. Cada vetor ou matriz é composto por números, chamados de *elementos*.

Vetores e matrizes

Definindo vetores

- Para definirmos vetores, fazemos uso de colchetes. Por exemplo, o comando:

$$a = [0, 2, 5]$$

define um vetor-linha, que pode ser interpretado como uma matriz de apenas 1 linha e 3 colunas (1X3). O comando:

$$a = [0; 2; 5]$$

define um vetor-coluna, que pode ser interpretado como uma matriz de 3 linhas e apenas 1 coluna (3X1).

Vetores e matrizes

Definindo matrizes

- Percebemos dos dois exemplos anteriores que a *vírgula* separa elementos *em uma mesma linha* e o *ponto e vírgula* separa elementos *em uma mesma coluna*.
- Uma matriz quadrada 3X3, portanto, pode ser definida da seguinte maneira:

$$\text{a} = [1,\ 2,\ 3;\ 4,\ 5,\ 6;\ 7,\ 8,\ 9]$$

- Elementos de uma mesma linha também podem ser separados apenas por espaços. O exemplo acima é equivalente a:

$$\text{a} = [1\ 2\ 3;\ 4\ 5\ 6;\ 7\ 8\ 9]$$

- Brinque de definir matrizes e se acostume com a sintaxe!
- Em versões mais recentes do MATLAB, você pode visualizar o conteúdo de uma matriz na forma de uma planilha ao clicar duas vezes sobre seu nome no *workspace*.

Vetores e matrizes

Notação *colon*

- Existem várias outras maneiras de definir vetores e matrizes. Uma delas é a partir da notação *colon*, que tem a forma:

$$a = primeiro\_valor : passo : ultimo\_valor$$

- Aqui temos um vetor-linha a que começa com *primeiro_valor* e tem seus elementos crescendo de *passo* em *passo* até o *ultimo_valor*.
- Por exemplo, o comando $a = 0 : 0.5 : 3$ geraria o seguinte vetor-linha:

$$a = [0,\ 0.5,\ 1,\ 1.5,\ 2,\ 2.5,\ 3]$$

- Se o valor para o passo é deixado em branco, o MATLAB o determina como sendo 1. Por exemplo, o comando $a = 0:3$ geraria:

$$a = [0,\ 1,\ 2,\ 3]$$

Bug!

Erro comum: escolher um tamanho de passo que, a partir do primeiro elemento, não chegue exatamente no último valor desejado (o vetor sempre termina no maior valor que seja menor ou igual ao *ultimo_valor* especificado no comando).

Vetores e matrizes

Definindo vetores e matrizes a partir de funções especiais

- Existem funções feitas especialmente para definir matrizes. Exemplos:
  - *linspace(valor_inicial, valor_final, numero_de_valores)*: parecido com a notação *colon*, mas ao invés de especificar o tamanho das subdivisões existentes no intervalo do vetor criado, especifica o número de elementos presentes.
  - *logspace(valor_inicial, valor_final, numero_de_valores)*: similar ao *linspace*, mas os elementos são gerados no espaço logarítmico ao invés do linear. Os valores inicial e final devem ser as potências de 10 dos elementos que se deseja criar.
  - *zeros(n,m)*: define uma matriz preenchida de zeros, de dimensão $n \times m$.
  - *ones(n,m)*: define uma matriz preenchida de uns, de dimensão $n \times m$.
  - *eye(n)*: define uma matriz identidade de dimensão $n \times n$.
  - *magic(n)*: define uma matriz de dimensão n cuja soma dos elementos em cada coluna, cada linha e em cada diagonal são iguais.
  - *diag(a)*: define uma matriz diagonal com os elementos do vetor a.
  - *rand(n,m)*: define uma matriz de números aleatórios entre 0 e 1, de dimensão $n \times m$.
  - *randn(n,m)*: define uma matriz de números aleatórios com distribuição normal entre 0 e 1, de de dimensão $n \times m$.
  - *repmat(x,r,c)*: cria uma matriz feita de cópias de x, onde r é o número de cópias de x replicadas como linhas, e c é o número de cópias de x replicadas como colunas.

Vetores e matrizes

Operações

- Operações com vetores e matrizes no MATLAB seguem as regras da álgebra linear. Defina algumas matrizes e teste! Tente a soma e a multiplicação entre matrizes, ou a multiplicação por um escalar.
- Algumas operações são válidas apenas para vetores e matrizes, como a *transposição* (operador linha, '). Por exemplo, defina uma matriz A na linha de comando e digite A'.
- Podemos também querer operar elemento a elemento, ou seja, operar cada elemento ij de uma matriz com o correspondente elemento ij de outra. Para isso, precedemos o operador por um ponto.
- Defina, por exemplo, duas matrizes de mesma dimensão A e B, e tente fazer as seguintes operações: A*B, A .* B, A^2, A.^ 2, A.^B e A^B.

Bug!

Erro comum: esquecer-se do *ponto* quando se deseja fazer operações elemento a elemento. Este deslize é bastante recorrente, em especial quando escrevemos funções. Isto ficará claro mais adiante.

Erro comum: utilizar um vetor-linha quando o contexto pede um vetor-coluna, ou vice-versa. Para transformar um em outro, utilize o operador transposição '.

Vetores e matrizes

Bug!

Erro comum: tentar concatenar matrizes que não têm dimensões compatíveis.

Concatenação de matrizes

- É possível também juntar vetores e matrizes de modo a formar um só.
- Se quisermos juntar matrizes linha a linha, de modo a criar uma nova matriz com um maior número de colunas, usamos a forma $C = [A,B]$.
  Ex: A = [1 2; 5 6]; B = [3 4; 7 8]; C = [A,B] = [1 2 3 4; 5 6 7 8]
- Se quisermos juntar matrizes coluna a coluna, de modo a criar uma nova matriz com um maior número de linhas, usamos a forma $C = [A;B]$.
  Ex: A = [1 2; 3 4]; B = [5 6; 7 8]; C = [A;B] = [1 2; 3 4; 5 6; 7 8]

Vetores e matrizes

Funções de matrizes

- Algumas funções podem aceitar como argumento e/ou fornecer como resultado uma ou mais matrizes.
- A maioria das funções pré-definidas no MATLAB aceitam, além de escalares, matrizes como argumento, operando sobre elas elemento-a-elemento!
- Por exemplo, defina uma matriz A qualquer e calcule sin(A).

Vetores e matrizes

Funções especiais para matrizes

- Certas funções, apesar de também aceitarem escalares como argumentos, são definidas especialmente para operar sobre matrizes. Algumas delas são:
  - *det(A)*: calcula o determinante de A.
  - *inv(A)*: calcula a inversa de A.
  - *rank(A)*: calcula o número de linhas ou colunas linearmente independentes de A.
  - *size(A)*: retorna os números de linhas e colunas de A.
  - *length(A)*: retorna o número de colunas de A (se A for um vetor-coluna, calcula seu número de linhas).
  - *sum(A)*: calcula a soma dos elementos de cada coluna de A.
  - *mean(A)*: calcula a média dos elementos de cada coluna de A.
  - *var(A)*: calcula a variância dos elementos de cada coluna de A.
  - *corrcoef(A)*: calcula os coeficientes de correlação entre os elementos de A.
  - *[Amin, locmin] = min(A)*: encontra os menores elementos ao longo de diferentes colunas de uma matriz.
  - *[Amax, locmax] = max(A)*: encontra os maiores elementos ao longo de diferentes colunas de uma matriz.
  - *[V,D] = eig(A)*: calcula os vetores e valores característicos de A.
  - *[U,S,V] = svd(A)*: calcula os vetores e valores singulares de A.

Seleção de elementos

E quando não quero a matriz inteira?

- Muitas vezes estamos interessados apenas em certas porções das matrizes: um determinado elemento, uma ou duas colunas específicas ou apenas a penúltima linha. Nessas situações utilizamos a *seleção de elementos*.

Selecionando elementos de um vetor

- Para selecionar o segundo elemento do vetor:

$$a = [10\ 20\ 30]$$

basta escrever *a(2)*

$$a(2) = 20$$

Para selecionar seu último elemento, basta escrever *a(end)*

$$a(end) = 30$$

Seleção de elementos

Selecionando elementos de uma matriz

- Para selecionar o elemento da linha 2, coluna 3, da matriz:

$$a = [10\ 20\ 30;\ 40\ 50\ 60;\ 70\ 80\ 90]$$

basta escrever *a(2,3)*

$$a(2,3) = 60$$

Seleção de elementos

Selecionando elementos de uma matriz

- Para selecionar a segunda linha da matriz:

$$a = [10\ 20\ 30;\ 40\ 50\ 60;\ 70\ 80\ 90]$$

basta escrever *a(2,:)*

$$a(2,:) = [40\ 50\ 60]$$

Os dois pontos em *a(2,:)* indicam *todas as colunas da matriz*.

- Para selecionar a primeira coluna da mesma matriz, basta escrever *a(:,1)*

$$a(:,1) = [10;\ 40;\ 70]$$

Os dois pontos em *a(:,1)* indicam *todas as linhas da matriz*.

Seleção de elementos

Selecionando elementos de uma matriz

- Para selecionar os elementos das duas primeiras linhas da terceira coluna da matriz:

$$a = [10\ 20\ 30;\ 40\ 50\ 60;\ 70\ 80\ 90]$$

basta escrever *a(1:2,3)*

$$a(1:2,3) = [30;\ 60]$$

Repare o uso da notação *colon* no primeiro índice da matriz, indicando que estamos interessados em selecionar as linhas de 1 até 2 da matriz *a* !

- Para selecionar os elementos das duas primeiras linhas das duas últimas colunas da mesma matriz basta escrever *a(1:2,2:3)*

$$a(1:2,2:3) = [20\ 30;\ 50\ 60]$$

Seleção de elementos

Selecionando elementos de uma matriz

- Para selecionar os elementos da diagonal da matriz:

$$a = [10\ 20\ 30;\ 40\ 50\ 60;\ 70\ 80\ 90]$$

basta escrever *diag(a)*

$$diag(a) = [10;\ 50;\ 90]$$

Lembre-se de que a palavra *diag* também pode ter outro significado em MATLAB, já aqui apresentado: criar uma matriz diagonal com elementos de um dado vetor. Este é um exemplo em que um mesmo identificador é usado para representar mais de uma função! A diferença está no contexto em o identificador é aplicado. Um bom exemplo para entender isto é tentar prever e analisar o resultado da operação *diag(diag(a))*.

Seleção de elementos

Selecionando elementos de uma matriz

- Pergunta: qual seria o comando para selecionar as duas últimas colunas da matriz:

$$a = [10\ 20\ 30;\ 40\ 50\ 60;\ 70\ 80\ 90]$$

Seleção de elementos

Selecionando elementos de uma matriz

- Resposta: *a(:,end-1:end)*

$$a = [20\ 30;\ 50\ 60;\ 80\ 90]$$

Strings

O que são?

- *Strings* são vetores compostos por *caracteres*.
- Para definir *strings*, precisamos usar aspas simples. Por exemplo, digite o seguinte comando:

$$s = \text{'eu s2 matlab'}$$

- Podemos aqui aplicar as mesmas técnicas de extração de elementos e concatenação vistas. Por exemplo, o comando *s(4:12)* resulta em:

$$\text{'s2 matlab'}$$

Ou se definimos $t = \text{' muito mesmo'}$ e digitamos $u = [s,t]$, teremos:

$$u = \text{'eu s2 matlab muito mesmo'}$$

Bug!

Erro comum: tentar usar aspas duplas ao invés de aspas simples para definir strings.

Strings

Agrupando strings

- Podemos criar matrizes feitas de strings com o comando *char*. Por exemplo, digite o comando *dias = char('segunda', 'terca', 'quarta')* e analise o resultado.
- Digite também os comandos *dias(2)*, *dias(2,:)*, *dias(2,1)*, *dias(2,5)*, *dias(2,6)*, *dias(2,8)*. Compare os resultados.
- A partir da análise dos resultados anteriores, perceba a presença de *caracteres em branco*, adicionados ao final de 'terca' e 'quarta' pelo comando *char* para tornar compatíveis os tamanhos das strings concatenadas (ambas passam a ter o mesmo tamanho que 'segunda').

Strings

Bug!

Erro comum: esquecer de que as strings que estamos manipulando podem ter caracteres em branco.

Comandos úteis

- Alguns comandos úteis para strings são:
  - *deblank(s)*: remove os caracteres em branco do final de uma string.
  - *strtrim(s)*: remove os caracteres em branco do começo e do final de uma string.
  - $L =$ *strcmp(s, t)*: compara as strings s e t. Se $s = t$, teremos $L = 1$; caso contrário, $L = 0$.
- Por exemplo, analise o resultado do seguinte conjunto de comandos:

A = ' lelek lek lek ';
B = 'lelek lek lek';
C1 = strcmp(A,B)
C2 = strcmp(strtrim(A), strtrim(B))

Strings

De números para strings

- Algumas vezes precisamos converter variáveis que estão armazenadas na forma numérica para strings.
- O comando que realiza tal tarefa é o *num2str*, que tem a seguinte sintaxe:

  $z = num2str(num, N)$.

  no qual *num* é um número ou matriz, ou uma expressão que resulte em um número ou matriz e N é a precisão desejada na conversão. Por exemplo, defina $num = 1000*pi$ e analise o resultado dos seguintes comandos: *num2str(num,1)*, *num2str(num,3)*, *num2str(num,4)*, *num2str(num,5)*, *num2str(num,8)*.
- Para converter um inteiro em uma string, use $z = int2str(num)$, em que *num* é um inteiro.

Bug!

Erro comum: tentar usar valores numéricos quando o contexto pede uma string.

Matrizes celulares

Misturando tudo

- *Matrizes celulares* são compostas por células que, por sua vez, podem armazenar todos os tipos de dados que vimos (escalares, matrizes e strings).
- Definimos matrizes celulares de maneira semelhante à que definimos matrizes comuns, mas utilizando chaves { } ao invés de colchetes [ ].
- Por exemplo, digite os seguintes comandos:

  C = {2,3}
  D = {2, 3; 4, 5}
  E = {'palavra', 1:0.5:20 ; 8, [1 2; 3 4]}
  F = {}

- No *workspace*, clique duas vezes no nome das matrizes celulares que você definiu, para visualizá-las na forma de planilha.

Matrizes celulares

Extração de elementos

- Podemos extrair elementos de matrizes celulares de duas formas: com parênteses ( ) ou chaves { }.
- Ao usarmos *parênteses*, estamos selecionando uma *porção da matriz celular, mas não os elementos que estão dentro de cada célula*.
- Ao usarmos *chaves*, estamos selecionando o *conteúdo em si de cada célula*.
- Exemplo: defina a seguinte matriz celular:

  C = {'one', 'two', 'three';
  1, 2, 3};

  E teste os seguintes comandos:

  C {2,1}
  C (2,1)
  C {1:2,1:2}
  C (1:2,1:2)
  C {2,1}+4
  C (2,1)+4

Matrizes celulares

Combinando matrizes celulares

- Podemos combinar matrizes celulares também de duas maneiras: por *concatenação* ou por *aninhamento (nesting)*.
- A *concatenação* de matrizes celulares é idêntica à de matrizes comuns. Usamos o operador colchetes [ ]. Por exemplo, digite os seguintes comandos:

  C1 = {1, 2, 3};
  C2 = {'A', 'B', 'C'};
  C3 = {10, 20, 30};
  C4 = [C1; C2; C3]
- O *aninhamento* ocorre quando uma célula de uma matriz celular é, por sua vez, *outra matriz celular*. Usamos o operador chaves { }. Por exemplo:

  C5 = {C1; C2; C3}
- Verifique os resultados, visualizando as matrizes celulares na forma de planilha.

Gráficos

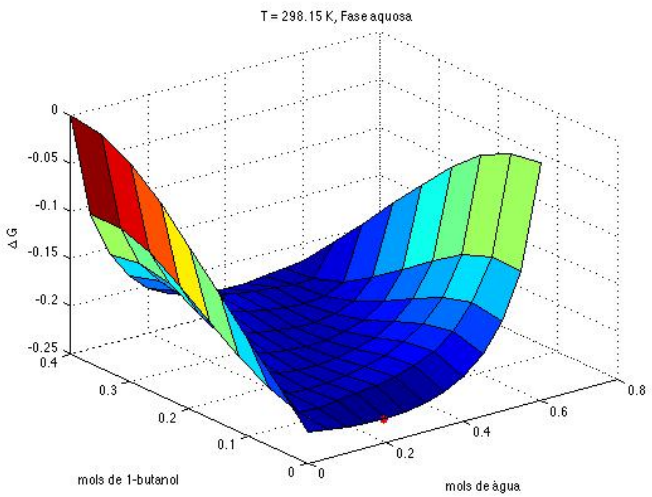


Figura 2: Energia de Gibbs de um sistema com 0,4 mols de butanol e 0,6 de água, a 298,15 K, em função dos números de mols dos componentes na fase aquosa. Plotado por meio do uso da função *surf*.

Plotando

- Outro recurso muito poderoso do MATLAB é sua capacidade de gerar vários tipos de gráficos, a partir de certos comandos.
- Na maioria dos comandos utilizados, a lógica é a mesma:
  - *definimos um domínio discreto* por meio da criação de uma *malha de pontos*;
  - *especificamos os valores da curva* a ser plotada, nestes pontos;
  - e *plotamos*!

Gráficos 2D - função *plot*

Função *plot*

- A maneira mais simples de plotar dados no MATLAB é usando a função *plot*.
- Por exemplo, para plotar a função matemática $y = sen(x)$ no intervalo $x = [-\pi, \pi]$, a sequência é a seguinte:
  - *x = -pi:0.1:pi;* Com esse comando, utilizamos a notação colon para criar um vetor que armazena os valores de x nos quais a função deve ser plotada. Esta é a etapa de *definição do domínio*.
  - y = sin(x); Cria um vetor y, da mesma dimensão de x, contendo os senos de cada um dos elementos de x. Nesta etapa, *especificamos os valores* y que a curva assume para cada x.
  - plot(x,y) e a mágica acontece!

Bug!

Erro comum: tentar usar a função *plot* com dois vetores que não tenham a mesma dimensão.

Erro comum: usar intervalos grandes demais entre os elementos do vetor do domínio, o que faz com que não haja a *impressão* de que a curva seja contínua.

Gráficos 2D - função *plot*

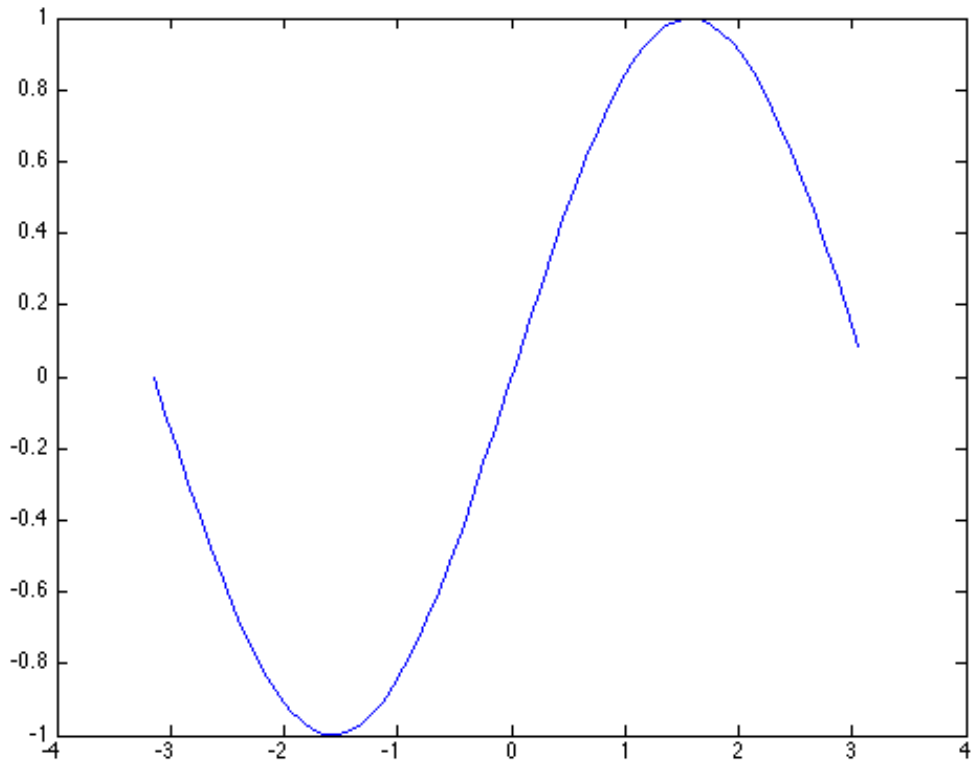


Figura 3: Função matemática $y = sen(x)$ plotada no intervalo $x = [-\pi, \pi]$ com o uso da função *plot*.

Customizando a plotagem

Bug!

Erro comum: esquecer de usar as aspas na string contendo as opções de customização.

Opções de customização

- Podemos personalizar nossos gráficos por meio de especificações sobre diferentes cores, marcadores ou tipos de curva.
- Isso é feito através de uma *string* que contenha as especificações desejadas e que deve ser fornecida como terceiro argumento para a função *plot*.
- Para plotar a curva anterior com asteriscos vermelhos, o comando seria *plot(x,y,'*r')*. Teste!

Customizando a plotagem

Cores de linhas		*Marcadores*		*Tipo de linha*	
símbolo	cor	Símbolo	Marcador	Símbolo	Tipo de linha
b	azul	.	ponto	-	linha contínua
g	verde	o	círculo	:	linha pontilhada
r	vermelho	x	x	-.	traços e pontos
c	ciano	+	+	--	linha tracejada
m	magenta	*	estrela		
y	amarelo	s	quadrado		
k	preto	d	losango		
w	branco	<	triângulo para a esquerda		
		>	triângulo para a direita		
		p	pentagrama		
		h	hexagrama		

Formatando os gráficos

Opções de formatação

- Após criado o gráfico, podemos usar funções específicas para modificá-lo, como por exemplo:
  - Título: *title('Titulo do grafico')*
  - Título do eixo x: *xlabel('Titulo do eixo x')*
  - Título do eixo y: *ylabel('Titulo do eixo y')*
  - Legendas das curvas: *legend('grafico1','grafico2', ...)*
  - Textos avulsos: *text(x,y,'Texto a ser colocado')* Aqui, x e y são as coordenadas do ponto onde será colocado o texto.
  - Faixas dos eixos: *axis([xinicial xfinal yinicial yfinal])*
- Essas opções estão disponíveis também nas janela dos próprios gráficos.

Bug!

Erro comum: tentar usar as opções de formatação *antes* de construir o gráfico.

Erro comum: na função *axis*, confundir-se com a ordem das especificações ou esquecer-se dos colchetes.

Gráficos 2D - mais de uma curva

Bug!

Erro comum: esquecer de usar a função hold (comandos hold on ou hold off) nas situações apropriadas.

Várias curvas em uma figura só

- Podemos plotar várias curvas em apenas uma figura de duas maneiras:
  - Usando apenas um comando: *plot(x, sin(x), 'b', x, cos(x), 'r')*
  - Ou usando o comando hold on:

```
plot(x,sin(x),'b')
hold on
plot(x,cos(x),'r')
```

- Para desligar o efeito do *hold on*, utilize o *hold off*.

Gráficos 2D - mais de uma curva

Figuras separadas em janelas separadas

- Para plotar várias figuras separadas, utilize a função *figure*. Por exemplo, após uma figura já estar plotada, você pode preparar outra janela de gráfico com o comando *figure(2)*; uma terceira janela com o comando *figure(3)*, e assim sucessivamente. Teste!

Figuras separadas em uma mesma janela

- É possível plotar vários gráficos separados, em uma mesma janela, utilizando a função *subplot*.
- A sintaxe é: *subplot(i,j,k)*. Os dois primeiros argumentos dividem a janela em seções (linhas e colunas) e o terceiro indica em que seção o gráfico deve ser plotado.
- Experimente por exemplo digitar os comandos *subplot(2,2,1)*, *subplot(2,2,3)*, *subplot(2,2,5)*.
- As expressões que plotarão as curvas devem vir após os comandos *subplot*.

Gráficos 2D - mais de uma curva

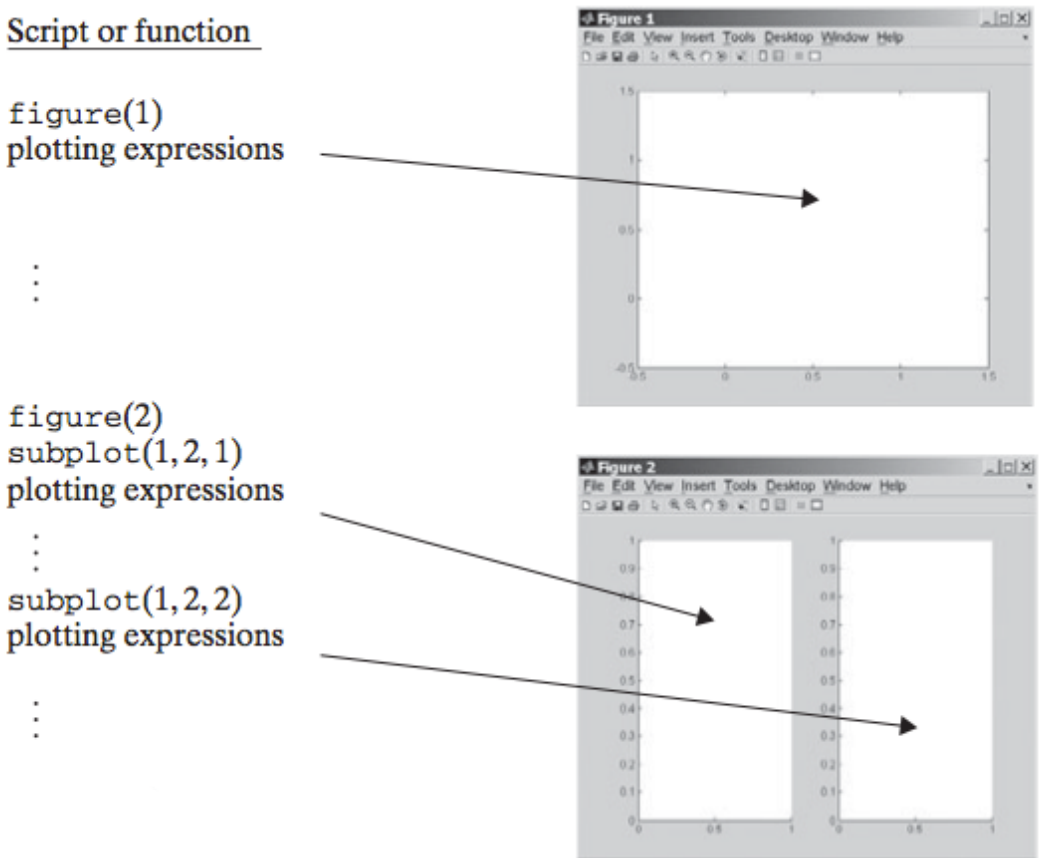


Figura 4: Ilustração do uso do comando *subplot*.

Gráficos 2D - mais de uma curva

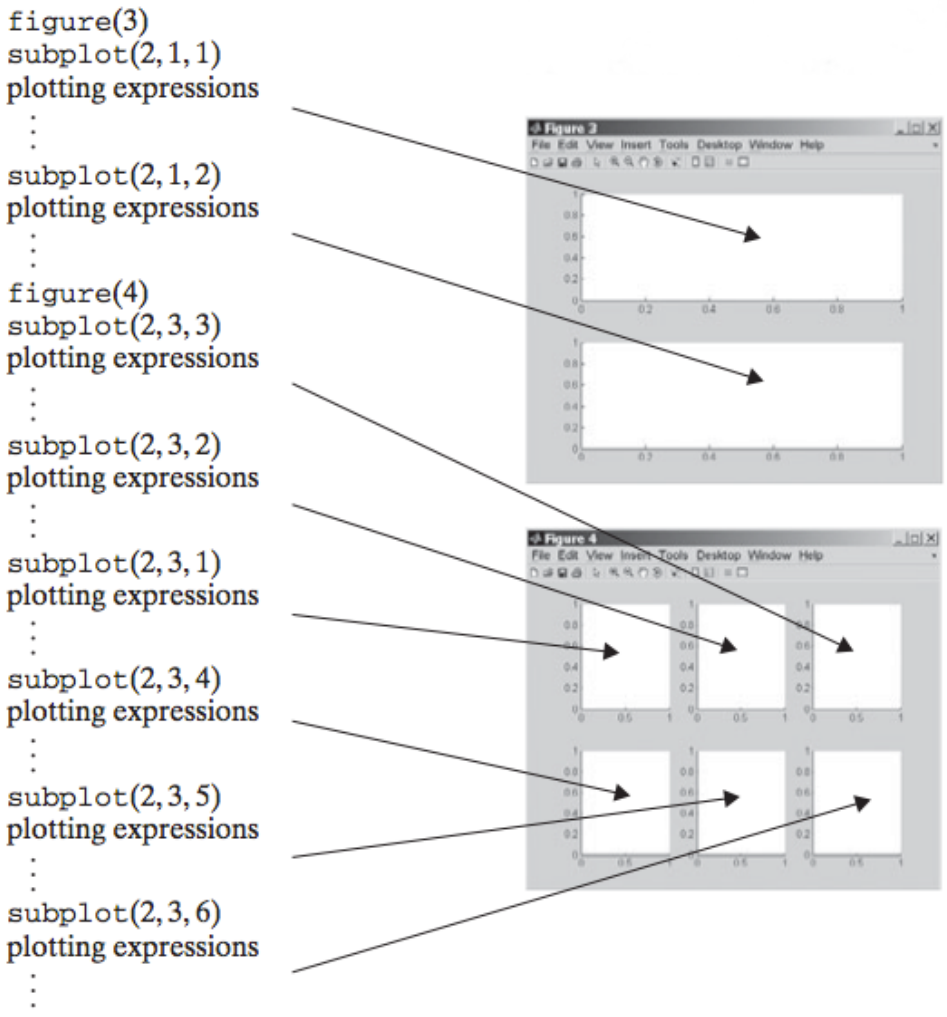
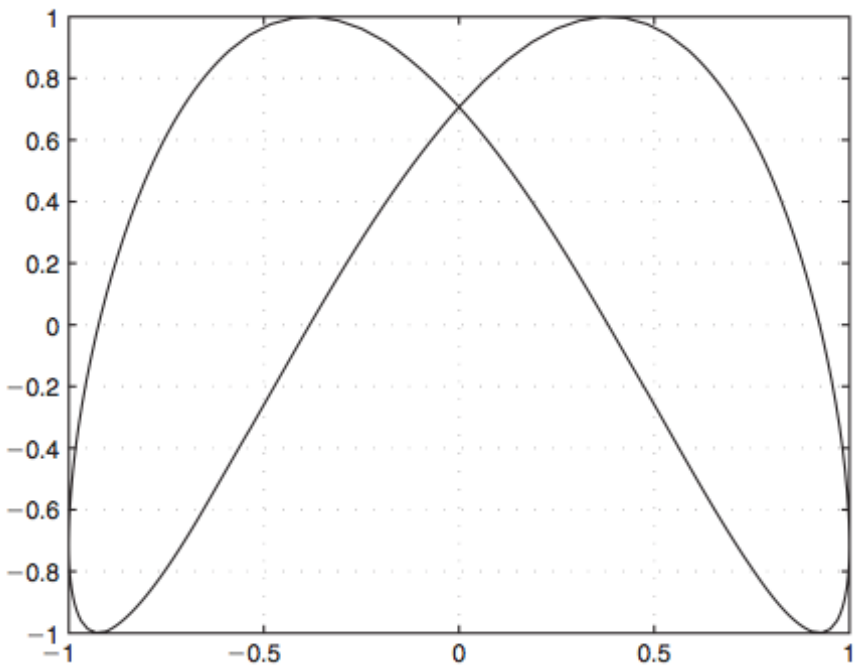


Figura 5: Ilustração do uso do comando *subplot*.

Gráficos 2D - comandos *box*, *grid* e *axis*

TABLE 6.3 Illustration of box, grid, and axis

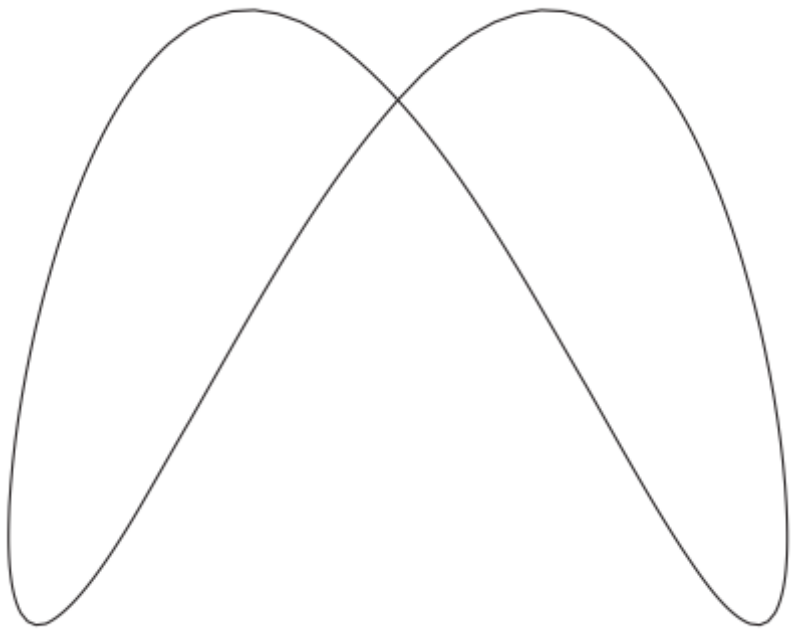
Function	Script	Graph

box on
grid on
	th = linspace(0, 2*pi, 101);
	x = sin(th);
	y = sin(2*th+pi/4);
	plot(x, y, 'k-')
	box on
	grid on
		

Gráficos 2D - comandos *box*, *grid* e *axis*

```
box off
grid off
axis off
```

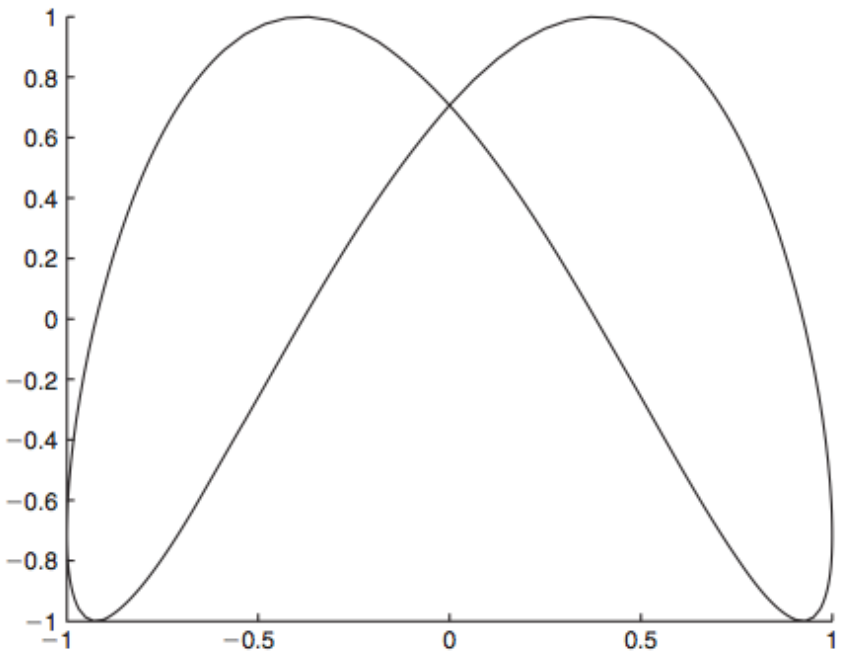
```
th = linspace(0, 2*pi, 101);
x = sin(th);
y = sin(2*th+pi/4);
plot(x, y, 'k-')
box off
grid off
axis off
```



Gráficos 2D - comandos *box*, *grid* e *axis*

```
box off
grid off
axis on
```

```
th = linspace(0, 2*pi, 101);
x = sin(th);
y = sin(2*th+pi/4);
plot(x, y, 'k-')
box off
grid off
```



Gráficos 2D do tipo torta

Gráficos de torta

- Úteis principalmente na área de estatística, podemos plotar em MATLAB os chamados *gráficos de torta*.
- A sintaxe é: *pie(d, expl, label)*, onde *d* é um vetor com os dados a serem plotados, *expl* é um vetor opcional consistindo de 1's ou 0's que indicam se o setor em questão deve ou não estar separado da torta e *label*, também opcional, é uma matriz com as strings a serem usadas como legendas de cada setor.
- A título de ilustração, defina o conjunto de dados *dat = [39,10,1]* e construa gráficos com os seguintes comandos:

  pie(dat)
  pie3(dat)
  pie(dat)
  pie3 (dat,[0 1 1])

Gráficos 2D do tipo torta

Gráficos de torta

- É comum que combinemos o comando *num2str* de modo a exibir as porcentagens realtivas dos dados no gráfico. Por exemplo, digite os seguintes comandos:

```
dat = [45, 43, 7, 5];
aecio = ['Aécio Neves ' num2str(dat(1)) '%'];
dilma = ['Dilma Rousseff ' num2str(dat(2)) '%'];
branco_nulo = ['Branco/Nulo ' num2str(dat(3)) '%'];
nao_sabe = ['Não sabe/não respondeu ' num2str(dat(4)) '%'];
label = {aecio, dilma, branco_nulo, nao_sabe}
pie (dat, [1 1 1 1], label)
title('PESQUISA ELEITORAL - IBOPE - 15/10')
```

Gráficos 2D do tipo torta

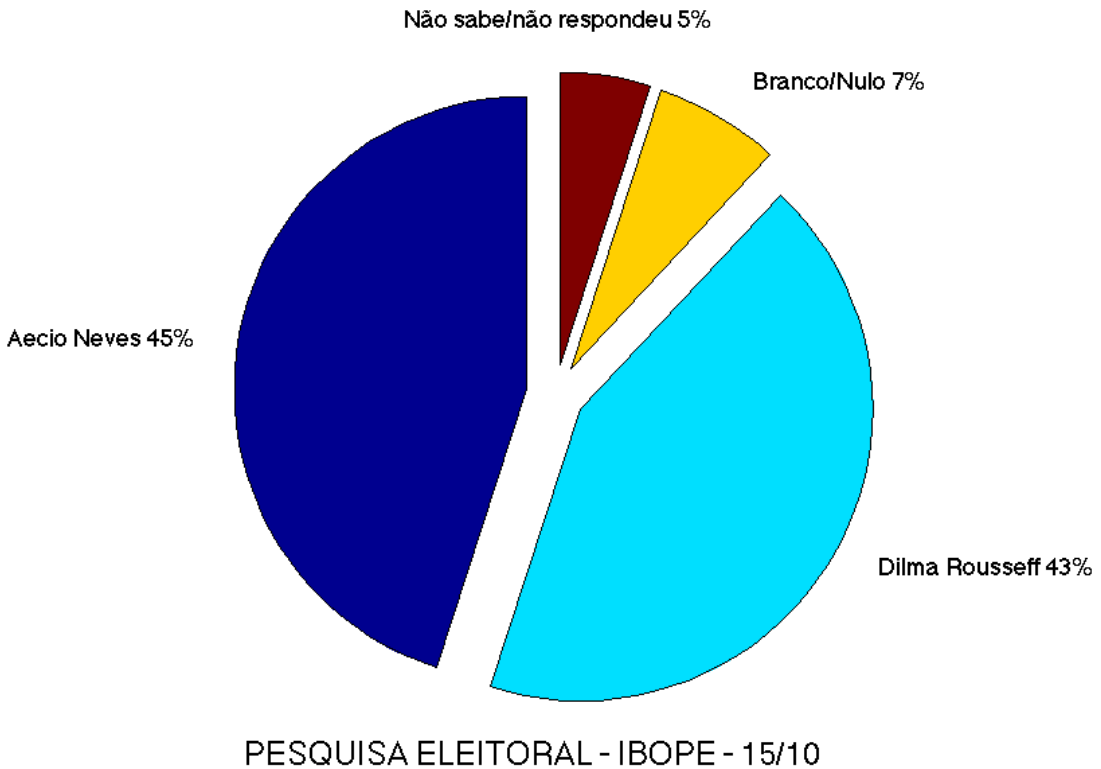


Figura 6: Gráfico de torta gerado pelos comandos do slide anterior.

Gráficos 2D

Outros tipos de gráficos bidimensionais

- Algumas funções adicionais que plotam gráficos 2D são:
  - *semilogx*: semelhante à *plot*, traça a curva estando o eixo x em escala logarítmica.
  - *semilogy*: análoga ao *semilogy*, mas com o eixo y.
  - *loglog*: semelhante aos anteriores, mas com os dois eixos em escala logarítmica.
  - *bar*: gráficos de barras.
  - *errorbar*: gráficos com barras de erros.

  Entre muitos outros!

Gráficos 3D

Visualizando superfícies

- Gráficos tridimensionais são úteis principalmente na visualização de superfícies de funções de duas variáveis.
- A título de ilustração, plotaremos a função $f(x, y) = x^2 + y^2$, uma parábola bidimensional.

A função *meshgrid*

- De modo a especificar uma malha bidimensional para representar o domínio da função matemática a ser plotada, usamos a função *meshgrid*.
- A meshgrid *aceita como argumento* dois vetores que definem o domínio e *retorna* duas matrizes que representam os pontos da malha bidimensional. Ex: x = -4 : 0.1 : 4; y = -4 : 0.1 : 4; [X,Y] = meshgrid(x,y);

Bug!

Erro comum: tentar plotar a função sem especificar a malha bidimensional.

Gráficos 3D

A função *surf*

- Para *definirmos os valores* da superfície a ser plotada, basta criarmos uma matriz que relaciona os pontos da malha bidimensional na forma da função desejada: `Z = Y.^2+X.^2;`
- Agora, é só digitar *surf(X,Y,Z)* e a mágica acontece!

A função *mesh*

- A função *mesh*, é análoga à *surf*, mas ao invés de gerar uma superfície, gera um gráfico de rede.

Bug!

Erro comum: não utilizar, quando necessárias, operações elemento-a-elemento na definição da função matemática a ser plotada.

Gráficos 3D

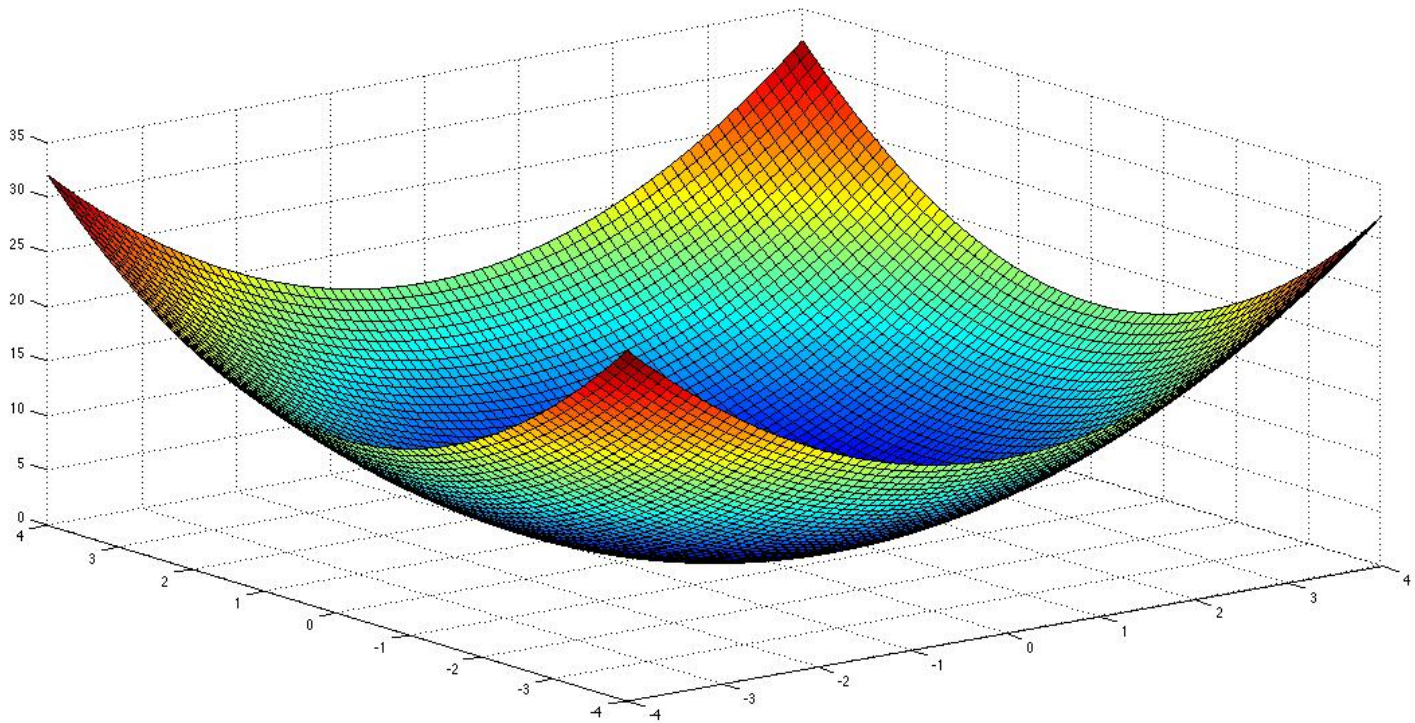


Figura 7: Função $f(x, y) = x^2 + y^2$ plotada com o uso do comando *surf*.

Gráficos 3D

Outros tipos de gráficos tridimensionais

- Outras funções que plotam gráficos 3D são:
  - *plot3*: gráfico de linha tridimensionais.
  - *pcolor*: gera um mapa de cores projetado na malha bidimensional.
  - *contour*: gera curvas de nível projetadas na malha bidimensional.
  - *contour3*: gera curvas de nível no espaço tridimensional.

O que é programação?

Definições

- *Computadores* são máquinas projetadas para realizar operações aritméticas e lógicas seguindo uma série de instruções pré-definidas.
- Um conjunto de instruções destinadas a serem seguidas por um computador é conhecido como *programa*.
- Portanto, a *programação* pode ser definida como *a arte de se escrever códigos que contém instruções para determinadas tarefas a serem realizadas por um computador*.

Figura 8: Computador.

Programando

Comunicação com o computador

- Computadores só entendem *linguagem binária*. Ou seja, todas as instruções que desejarmos passar a um computador devem estar expressas em termos de 0 ou 1.
- Exemplo de uma instrução entendida por um computador: 000101010101.
- O problema é que seres humanos (normais) não entendem binário! Como efetuar a comunicação entre o homem e o computador?
- Com o objetivo de resolver este problema, foram criadas as *linguagens de programação*.

Programando

Linguagens de programação

- As primeiras linguagens de programação que surgiram foram as de *baixo nível de abstração* (ou apenas de *baixo nível*). Estas apresentam sintaxes relacionadas diretamente às instruções que devem ser seguidas por um processador (apresentando pouco comprometimento em relação à legibilidade por humanos). Um exemplo é a Assembly.
- Mais tarde, foram aparecendo linguagens de *alto nível de abstração* (ou apenas *de alto nível*), que têm mais relação com os objetivos gerais do programador do que com as tarefas específicas do processador (sendo muito mais práticas, porém menos eficientes). Exemplos são C, Fortran, Python ou Matlab.

Rodando o programa

Linguagens compiladas X Linguagens interpretadas

- Imagine que você escreveu um programa em alguma linguagem de alto nível. Como fazer para rodá-lo?
- Em primeiro lugar, é preciso traduzir o código da *linguagem de programação* (que *você* entende) para a *linguagem binária* (que a *máquina* entende). Isto pode ser feito de duas maneiras:
  - *Compilação*: neste caso, traduz-se todo o programa de uma só vez para o código binário, gerando um arquivo chamado *executável* (no Windows, um arquivo com extensão *.exe*). Esse executável então pode ser rodado. Exemplos de linguagens compiladas: C/C++, Fortran.
  - *Interpretação*: aqui, o programa é traduzido e executado linha por linha por algum software que age como *interpretador*. O desempenho é menor em relação aos programas compilados. Exemplos de linguagens interpretadas: Matlab, Octave, Python.

Figura 9: Ilustração da necessidade da compilação ou interpretação.

Erros de programação

Os famosos bugs!

- É preciso ter em mente que existem três tipos de erros de programação:
  - *Erro de sintaxe*: este erro ocorre quando escrevemos algo que *não está definido* na linguagem, ou seja, que o interpretador não entende. A *execução do programa é interrompida e uma mensagem de erro emitida*.
  - *Erro de lógica*: este erro aparece quando o programa funciona, mas não do jeito planejado, fornecendo *resultados espúrios* (talvez por alguma fórmula errada, etc) . É o mais difícil de ser encontrado.
  - *Erro de estilo*: é o menos grave de todos, no sentido de que não prejudica a execução do programa. Muitos nem consideram este como um tipo de erro. Ocorre quando o programador desenvolve o seu código de maneira bagunçada e ilegível, o que dificulta sua *manutenção, desenvolvimento* e *extensão*.

Figura 10: Fuja destes insetos!

Scripts

Scripts: conjuntos de comandos

- Programas em MATLAB são constituídos por *scripts*.
- Um *script* nada mais é do que um arquivo de texto contendo um conjunto de comandos a serem executados sequencialmente pelo MATLAB.
- Esse arquivo, cuja extensão é *.m*, pode ser escrito com qualquer editor de texto (por exemplo, o famoso programa *Bloco de Notas*). Em geral, usamos o editor disponível no próprio MATLAB.

Scripts

Criando scripts

- Para criar um novo script, clique no botão "*New Script*" ou semelhante (em versões mais recentes do MATLAB, presente na aba "*Home*").
- Uma nova janela ou aba do editor do MATLAB será aberta e você poderá escrever o conjunto de comandos que constituirá seu programa.
- Você pode salvar seu script (por meio do botão na própria janela do MATLAB ou do atalho ctrl+S).
- Para rodar o script, basta digitar seu nome na linha de comando ou clicar no botão "*Run*" (em versões mais recentes, presente na aba "*Editor*").

Atividade

- Escreva e rode um script que plote o gráfico da função $f(x) = 1/x$ entre -1 e 1.

Scripts

Navegando pelo computador

- É importante perceber que podemos navegar pelo sistema de arquivos do computador através do MATLAB.
- Fazemos isso por meio de uma seção da tela intitulada "*Current Folder*" ou com uma barra de endereços localizada na parte superior da tela. Teste!
- Para rodar um script, *ele deve estar salvo na mesma pasta em que o MATLAB se encontra no momento da execução*.

Comentários

Escrevendo para você e não para o computador

- Comentários são linhas que escrevemos no programa, mas que são ignoradas pelo MATLAB.
- Têm como objetivo a descrição do código, de modo a deixá-lo claro para alguém que eventualmente vá lê-lo.
- Comentários em MATLAB devem começar com o sinal de porcentagem %.

Dicas para programação em MATLAB

Ao escrever um script...

- ...*minimize o número de linhas no seu código*! Em geral, você vai querer que seu programa tenha o menor tamanho possível. Só tome cuidado para que ele não fique tão pequeno de modo que não possa ser entendido rapidamente por qualquer usuário médio de MATLAB. Aqui deve haver um compromisso entre *concisão* e *legibilidade*.
- ... *suprima o ponto e vírgula do final dos seus comandos, em locais estratégicos.* Quando certos resultados intermediários são mostrados na tela, o desenvolvedor consegue acompanhar o que seu programa está fazendo e a identificação de erros torna-se mais fácil. Esta é uma das técnicas mais simples e intuitivas de *debug*. Só tome cuidado para não ficar exibindo muitos resultados intermediários, já que isto aumenta bastante o tempo de execução do código!

Dicas para programação em MATLAB

Ao escrever um script...

- ... *use a ajuda do MATLAB*. Isto ajuda a minimizar erros de sintaxe e poupa seu tempo. Seja amigo do comando *help*.
- ... *comente seu código*! Comentários sucintos e explicativos tornam o código mais claro para alguém que vá lê-lo em outra ocasião (seja você mesma ou outra pessoa). Só tome cuidado para não exagerar e fazer com que o excesso de comentários atrapalhe a leitura!
- ... *valide seu código!* O fato de você estar encontrando *um* ou mesmo *alguns* resultados corretos não significa que seu programa esteja todo certo. Teste-o por meios independentes e no maior espectro de situações possível.

I/O Básico

Entrada e saída de dados

- I/O significa *input/output*, ou seja, entrada e saída de dados.
- A função mais simples para output no MATLAB é o *disp*, que imprime uma string na tela. Exemplo: *disp('Que bonita a sua roupa')*
- A função mais simples para input no MATLAB é a *input*. Ela, além de imprimir uma string na tela, funciona por meio da definição de uma variável, que deve por sua vez ser fornecida pelo usuário. Exemplo: $x = input('Insira\ o\ valor\ da\ variável\ x')$
- Caso a variável a ser fornecida seja uma string, deve-se colocar *'s'* como um segundo argumento na função *input*. Exemplo: *str = input('Insira o valor da string str','s')*

Atividade

- Teste você mesmo! Escreva um pequeno script que peça ao usuário um valor e o exiba na tela.

I/O Básico

A função *fprintf*

- A função *fprintf* é semelhante à *disp*, mas nos possibilita imprimir na tela, dentro de strings, valores de variáveis.
- Caso queiramos, por exemplo, imprimir na tela as variáveis *x1* (número inteiro) e *x2* (número real, de ponto flutuante), dentro de uma frase, o comando ficaria:

  fprintf('os numeros achados foram o inteiro %i e o real %f',x1,x2)

- O comando acima exibe na tela a string (que, é lógico, está entre aspas). Dentro da string colocamos os chamados *especificadores*, que indicam a posição em que as variáveis *x1* e *x2* (os outros dois argumentos que demos à função *fprintf*) são exibidas.
- Cada especificador diz respeito a um tipo de dado diferente (decimal inteiro, decimal de ponto flutuante, octal, etc).

Especificadores do comando *fprintf*

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`
`p`	Pointer address	`b8000000`
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

Estruturas de controle

Importância

- O coração da arte de se programar encontra-se nas *estruturas de controle*.
- Digitar uma linha para cada operação feita ou decisão tomada pelo computador seria cansativo, concorda?
- Com o uso das estruturas de controle, podemos com poucas linhas de código ordenar ao computador o que ele realmente foi designado para fazer, ou seja, *tarefas repetitivas e automáticas*.
- Dividem-se em dois tipos.
  - Estruturas condicionais;
  - Estruturas iterativas.

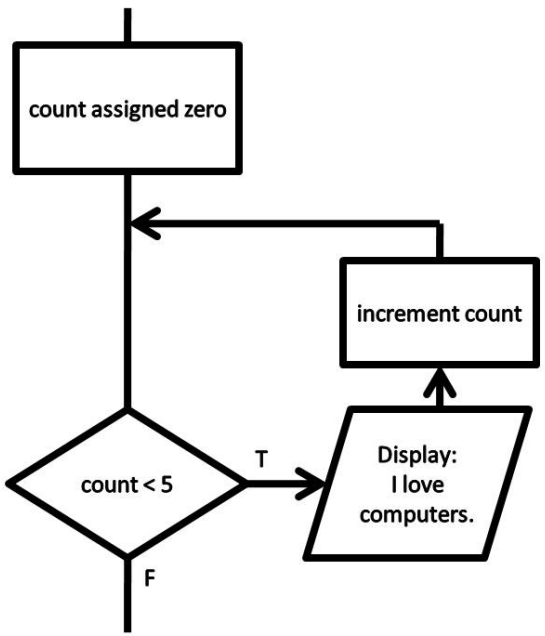


Figura 11: Exemplo de um algoritmo que utiliza estruturas condicionais e iterativas.

Estruturas condicionais: expressões lógicas

Expressões lógicas

- As estruturas condicionais avaliam *expressões lógicas* para decidir se executam ou não algum comando ou conjunto de comandos.
- Expressões lógicas só podem resultar em dois valores: *verdadeiro* (1) ou *falso* (0).
- A ideia é simples: se o resultado da expressão for verdadeiro, o(s) comando(s) são executado(s). Se for falso, não são executado(s).
- As expressões lógicas são construídas de acordo com a *álgebra booleana*.

Expressões lógicas

symbol	meaning	example
==	equal	`if x == y`
~=	not equal	`if x ~= y`
>	greater than	`if x > y`
>=	greater than or equal	`if x >= y`
<	less than	`if x < y`
<=	less than or equal	`if x <= y`
&	AND	`if x == 1 & y > 2`
|	OR	`if x == 1 | y > 2`
~	NOT	`x = ~y`

Estruturas Condicionais: o bloco *if*

if

- Executa um ou mais comandos sob determinada condição, representada por uma expressão lógica. A sintaxe é:

```
if (condicao)
        lista de comandos;
elseif (outra condicao)
        outra lista de comandos;
else
        ultima lista de comandos;
end
```

Exemplo 1

- Escreva um programa que peça ao usuário um valor de temperatura, em °C, e indique em qual estado físico, ao nível do mar, se encontra a água.

Bug!

Erro comum:	esquecer de fechar o bloco com o *end*.
Erro comum:	utilizar o operador atribuição (=) ao invés do operador lógico de igualdade (==) na condição avaliada.
Erro comum:	esquecer-se de que o bloco *if* é interrompido assim que uma condição verdadeira é encontrada.

Estruturas iterativas: o bloco while

Bug!

Erro comum: entrar em um loop infinito (a condição nunca deixa de ser satisfeita).

while

- Executa, enquanto uma dada *condição* permanece satisfeita, um ou mais comandos. A sintaxe é:

```
while (condicao)
        lista de comandos;
end
```

Exemplo 2

- Mostre a lista dos números naturais em tela, desde o 5000, em ordem decrescente.

Estruturas iterativas: o bloco for

for

- Executa um ou mais comandos enquanto um *contador* percorre valores dos elementos de um vetor especificado. A sintaxe é:

```
for i = [vetor]
        lista de comandos;
end
```

Exemplo 3

- Calcule a soma:

$$\sum_{i=1}^{6} 3^{i+1}$$

Bug!

Erro comum: ao calcular uma soma, não inicializar seu valor como sendo zero.

Funções definidas pelo usuário

Bug!

Erro comum: tentar declarar funções dentro do corpo de scripts comuns. É possível, no entanto, declarar funções dentro de outras funções.

Nossas próprias funções

- Funções são *scripts* especiais, que criam um *workspace* próprio (local e independente).
- Apesar da imensa gama de funções pré-definidas no MATLAB, se quisermos fazer um trabalho sério de programação, precisaremos definir nossas próprias funções!
- Funções usualmente são *chamadas* quando necessitadas e, com base em um ou mais *argumentos* fornecidos a elas, retornam um ou mais *resultados*.

Funções definidas pelo usuário

Sintaxe para declaração (criação da função)

- Toda função deve ter como começo uma linha no seguinte formato:

  $$function\ output = nome\ (input1,\ input2,\ ...)$$

  em que *output* é a variável de saída (valor que a função vai retornar), *input1* ,*input2* são as variáveis de entrada (valores que a função vai ler, também chamados de *argumentos*) e *nome* é o nome (dã!) da função.
- Dentro do corpo da função, podem haver quaisquer comandos que desejemos ver cumpridos.
- Em algum ponto dentro do corpo da função, devemos especificar o valor da variável *output*, para que esta seja fornecida como resultado ao usuário.
- Deve haver apenas um *output*, que pode ser um vetor. Caso seja necessário retornar vários resultados, eles devem ser atribuídos aos elementos do vetor *output*.
- As funções podem ou não terminar com o marcador *end*.

Funções definidas pelo usuário

Sintaxe para uso

- Uma vez criada a função e devidamente salva como arquivo .m, podemos então utilizá-la. Para isso, a invocamos na linha de comando, em scripts ou em outras funções.
- A forma de invocar a função é simples: basta digitar seu nome e, entre parênteses, fornecer os argumentos necessários. Já havíamos aprendido a fazer isso lá no começo, quando usamos as funções pré-definidas do MATLAB! Ex: cos(x).
- Podemos atribuir o resultado retornado pela função a uma variável. Ex: y = cos(x). Se nada for especificado, o resultado irá para a variável ans.

Funções definidas pelo usuário

Exemplo 4

- Implemente uma função que calcule o produto escalar entre dois vetores. Lembre-se de que, para realizar o produto escalar, os vetores precisam ter a mesma dimensão!

Funções definidas pelo usuário

Escopo

- O corpo de uma função é um mundo à parte, independente do resto do programa! Como ela tem seu próprio *workspace*, as variáveis que estão fora de uma função não são "enxergadas" de dentro dela, e vice-versa (as variáveis definidas dentro de uma função não são "enxergadas" do lado de fora).
- A porção do código de onde a variável pode ser enxergada é chamada de *escopo* da variável.
- Variáveis definidas dentro de uma função, portanto, *têm como escopo a própria função*. Ou seja, só são enxergadas dentro da função.

Bug!

Erro comum: esquecer-se de que as variáveis definidas dentro de uma função têm escopo diferenciado.

Funcões definidas pelo usuário

Usamos funções para...

- ... isolar operações complicadas;
- ... reduzir a complexidade do programa como um todo, tornando-o mais legível e fácil de manter;
- ... evitar códigos duplicados;
- ... promover a reusabilidade;
- ... limitar o efeito de mudanças em seções específicas de um programa;
- ... facilitar o *debug*.

Funcões definidas pelo usuário

Cabeçalho

- Um bom hábito é sempre escrever *cabeçalhos* em sua função (blocos de comentários, localizados logo após a primeira linha do arquivo, explicando brevemente seu propósito e maneira de usar).
- O cabeçalho de uma função pode ser acessado com o uso do comando *help*. Ex: *help sin*.

Funcões definidas pelo usuário

Funções anônimas

- Caso queiramos definir uma simples função matemática, não precisamos criar um arquivo separado para isto. Podemos definir uma *função anônima*.
- Funções anônimas tem a seguinte sintaxe:

$$\textbf{functionhandle} = @(\text{argumento})(\text{expressão})$$

- A melhor forma de se entender é com um exemplo. Para se definir a função $|cos(\beta x)|$, sendo $\beta = \pi/3$, e avaliá-la no valor $x = 4,1$, basta digitar na linha de comando:

```
bet = pi/3;
cx = @(x) (abs(cos(bet*x)));
cx(4.1)
```

Cálculo simbólico

Como se fizéssemos à mão

- O MATLAB foi criado tendo em vista a resolução de problemas *numéricos*, cujas soluções são naturalmente alcançadas via métodos computacionais.
- No entanto, podemos também realizar operações analíticas (aquelas do cálculo diferencial e integral) utilizando a *toolbox "Symbolic Math"*.

Bug!

Erro comum: confundir operações analíticas e numéricas.

Variáveis e expressões

Variáveis e expressões

- Para se definir *variáveis simbólicas*, basta usar o comando *syms*. Ex:

  syms x

- Com a variável na memória, agora podemos definir *expressões* com ela:

  num $=$ 12*x^2 + 24*x -4;
  denom $=$ 2*x^2 + 2*x - 6;
  f $=$ num/denom

- Podemos simplificar a expressão:

  simplify(f)

- Se quisermos que a expressão apareça bonitinha na tela, basta digitar:

  pretty(f)

- Também há a opção de plotar:

  ezplot(f)

Limites

Comandos para cálculo de limites

- Limites bidirecionais:

```
syms x t
f(x) = sin(x/x)
g(x,t) = (sin(x + t) - sin(x))
limit(f)
limit(g, t, 0)
```

- Limites à esquerda e à direita:

```
syms x
limit(1/x, x, 0, 'right')
limit(1/x, x, 0, 'left')
```

Derivadas

Comandos para cálculo de derivadas

- Funcões unidimensionais:

  syms x
  f(x) = sin(x^2)
  df = diff(f)

- Funcões multidimensionais:

  syms x t
  df = diff(sin($x * t^2$),t)

- Derivadas de ordem superior:

  syms x y
  diff(x*cos($x * y$), y, 4)

Integrais

Comandos para cálculo de integrais

- Integrais indefinidas:

  syms x z
  int($x/(1+z^2)$, z)

- Integrais definidas:

  syms x t
  int(x*log($1+x$), 0, 1)
  int(2*x, sin(t), 1)

O comando *rsums*

- Este comando, que tem a mesma sintaxe do *int*, faz uma demonstração do cálculo da integral por meio da soma de Riemman. Experimente, por exemplo:

  rsums (exp (-5*x^2))

Polinômios

Função roots

- Para extrair as raízes de um polinômio, utilizamos a função *roots*. Esta função aceita como argumento um vetor com os coeficientes do polinômio, dispostos na forma $[a_n, a_{n-1}, ..., a_0]$.
- Por exemplo, para extrair as raízes do polinômio $x^2 - 2x + 5$, o comando usado é:

  roots([1 -2 5]).

Sistemas de equações lineares

Operador barra invertida

- Uma maneira extremamente simples de resolver um sistema linear $Ax = b$ é utilizando o operador barra invertida. Por exemplo, seja o sistema:

$$\begin{aligned} 3.5x + \quad 2y \quad\quad\quad &= 5 \\ -1.5x + \quad 2.8y + 1.9z &= -1 \\ -2.5y + \quad 3z &= 2 \end{aligned}$$

- Para resolvê-lo, definimos a matriz A = [3.5, 2, 0; -1.5, 2.8, 1.9; 0, -2.5, 3], o vetor b = [5;-1;2], e digitamos A\b.

Equações não-lineares

Função fzero

- A função *fzero* é útil para achar os zeros de uma função de uma variável. Seus argumentos são um *function handle* da função (já devidamente salva como arquivo .m, ou declarada como função anônima) e uma estimativa inicial para os cálculos:

  fzero(@minha_funcao,chute_inicial)

- Por exemplo: *fzero(@f,3);*
- Podemos também fornecer uma função diretamente dentro de uma string, desde que seu argumento seja x. Exemplo: *fzero('sin(x)',3)*
- Se a função a ser fornecida está na forma de função anônima, não há a necessidade do @.

Sistemas de equações não-lineares

Função fsolve

- O *fsolve* consegue resolver sistemas de equações não lineares na forma:

$$
\begin{aligned}
f_1(x_1, x_2, x_3, ..., x_n) &= 0 \\
f_2(x_1, x_2, x_3, ..., x_n) &= 0 \\
f_3(x_1, x_2, x_3, ..., x_n) &= 0 \\
&..... \\
f_n(x_1, x_2, x_3, ..., x_n) &= 0
\end{aligned}
$$

Sistemas de equações não-lineares

Função fsolve

- Para usar o *fsolve*, precisamos definir uma função que aceite como *input* um vetor de tamanho n contendo os elementos $x_1, ..., x_n$ e que forneça como *output* outro vetor, também de tamanho n, contendo os valores das funções $f_1(x_1, ...x_n), ..., f_n(x_1, ..., x_n)$. Além disso, precisamos especificar um vetor de estimativas iniciais para os valores de x.
- E a sintaxe para utilizar o fsolve é muito parecida com o fzero:

fsolve (@minha_funcao,chute_inicial)

Exemplo 5

- Resolva o seguinte sistema de equações:

$$\frac{x}{1 + e^{-27y}(x/3 - 1)} - 5 = 0$$
$$\frac{x}{1 + e^{-39y}(x/3 - 1)} - 6 = 0$$

Equações diferenciais ordinárias

Qual solver de EDO usar?

- Como a gama de comportamentos das *equações diferenciais* é bem mais rica que das algébricas, para resolvê-las o MATLAB dispõe de vários solvers, cada um adequado para uma classe de problemas.
- No nosso caso usaremos o *ode45*, para resolver *problemas de valor inicial*, que podem ser postos na forma:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, x_3, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, x_3, ..., x_n)$$
$$.....$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, x_3, ..., x_n)$$

Equações diferenciais ordinárias

ode45

- A sintaxe do *ode45* é:

  [t,x] = ode45 (@f, intervalo_de_integracao, condicao_inicial)

- Precisamos, portanto, fornecer ao *ode45* uma função que representa $f(t, x)$, de maneira semelhante como fazíamos com as equações algébricas.
- O resultado (output) do ode45 são dois vetores (no caso de sistemas, o segundo argumento é uma matriz) contendo os pontos correspondentes das variáveis independente e dependente.

Bug!

Erro comum: não definir corretamente a função a ser fornecida para o ode45. Ela deve necessariamente aceitar dois argumentos (t e x, nessa ordem); mesmo que matematicamente sua função não dependa de t ou que x seja um vetor (caso de sistemas de equações).

Equações diferenciais ordinárias

Exemplo 6

- Resolva a equação:

$$\frac{dx}{dt} = 3e^{-t}, \; x(0) = 0$$

Equações diferenciais ordinárias

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

Equações diferenciais parciais

Qual solver de EDP usar?

- Se a gama de comportamentos das equações diferenciais ordinárias já é grande, a de *equações diferenciais parciais* é ainda maior!
- A título de ilustração, vamos aprender a resolver equações *parabólicas* e *elípticas* utilizando o solver *pdepe*.

pdepe

- O *pdepe* resolve equações na forma:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t}=x^{-m}\frac{\partial}{\partial x}\left(x^{m}f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right)+s\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

em $t_0 \leqslant t \leqslant t_f$ e $a \leqslant x \leqslant b$. As condições inicial e de contorno têm a forma:

$$u(x,t_0)=u_0(x)$$

$$p(x,t,u)+q(x,t)f(x,t,u,\frac{\partial u}{\partial x})=0$$

Equações diferenciais parciais

pdepe

- Sua sintaxe é:

$$sol = pdepe\ (m,\ @pdeID,\ @pdeIC,\ @pdeBC,\ xmesh,\ tspan)$$

Em que:

- *m*: parâmetro correspondente às coordenadas do problema (se m $= 0$, cartesianas; m $=$ 1, cilíndricas; m $= 2$, esféricas);
- @pdeID: um handle para uma função que defina c, f e s;
- @pdeIC: um handle para uma função que defina as condições iniciais;
- @pdeBC: um handle para uma função que defina as condições de contorno;
- *xmesh*: um vetor $[x_0,\ x_1,\ ...,\ x_n]$ especificando os pontos em que a solução numérica é requerida para cada valor de *tspan*;
- *tspan*: um vetor $[t_0,\ t_1,\ ...,\ t_f]$ especificando os pontos em que a solução numérica é requerida para cada valor de *xmesh*;
- *sol*: uma matriz *sol(i,j)* contendo a solução do problema, em que *i* corresponde aos resultados na malha temporal e *j* na malha espacial.

Equações diferenciais parciais

Exemplo 7

- Resolva a equação:

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

sujeita às condições inicial e de contorno:

$$u(x,0) = sen\,\pi x$$

$$u(0,t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0$$

A equação é válida para $0 \leqslant x \leqslant 1$ e $t \geqslant 0$.

Otimização

Encontrando a melhor opção!

- Dado um conjunto de possibilidades, a *otimização* é a escolha do(s) elemento(s) desse conjunto que melhor atende(m) a um critério específico. Esse critério pode ser o mínimo gasto de energia, um lucro máximo, menor dano ambiental, etc.
- Do ponto de vista matemático, problemas de otimização são formulados como problemas de minimização de funções. Por exemplo, sendo uma função $f(x_1, x_2, ...x_n)$, o ponto de ótimo é definido como o conjunto $(x_1, x_2, ...x_n)$ que torna mínimo o valor da função f.

Otimização

Otimização local ou global?

- Existem dois tipos de otimização: *local* e *global*.
- Métodos de otimização local são convenientes para:
  - funções com um único mínimo;
  - funções com vários mínimos, quando o interesse está em qualquer um desses mínimos e não apenas no menor de todos eles.
- Já os métodos de otimização global são necessários quando se deseja encontrar o mínimo global de uma função que possui múltiplos mínimos.
- Métodos de otimização local estão disponíveis na *toolbox "Optimization"*. Métodos de otimização global estão disponíveis na *toolbox "Global Optimization"*.

Otimização local

fminsearch

- Ilustraremos a resolução de problemas de otimização local com a função *fminsearch*. Sua sintaxe é:

  fminsearch(@minha_funcao,chute_inicial)

Exemplo 8

- Encontre o mínimo da seguinte função, conhecida como função de Rosenbrock:

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

  Em seguida, plote a superfície da função e o mínimo encontrado.

Otimização global

particleswarm

- Ilustraremos a resolução de problemas de otimização global com a função *particleswwarm*. Sua sintaxe é:

  particleswarm(@minha_funcao, nvars, lim_inf, lim_sup)

  sendo:
  - *nvars* o número de variáveis de que a *@minha_funcao* depende;
  - *lim_inf* e *lim_sup* vtores contendo os limites inferior e superior de busca das variáveis.

Exemplo 9

- Encontre o mínimo da seguinte função, conhecida como função "caixa de ovos":

$$f(x,y) = -(y+47)\sin\left(\sqrt{\left|\frac{x}{2}+(y+47)\right|}\right) - x\sin\left(\sqrt{|x-(y+47)|}\right)$$

  Em seguida, plote a superfície da função e o mínimo encontrado. A região de busca deve ser [-512, 512], tanto para x quanto para y.

Reator batelada

Modelo

- Sejam as reações sequenciais A $\xrightarrow{k_1}$ B $\xrightarrow{k_2}$ C ocorrendo em um reator químico do tipo batelada. A aplicação de balanços materiais com hipóteses apropriadas pode levar ao seguinte conjunto de equações:

$$\frac{dC_a}{dt} = -k_1 C_a$$

$$\frac{dC_b}{dt} = k_1 C_a - k_2 C_b$$

$$\frac{dC_c}{dt} = k_2 C_b$$

  sendo $k_1 = 1h^{-1}$, $k_2 = 2h^{-1}$. Na condição inicial, temos $C_{a0} = 5mol/L$ e $C_{b0} = C_{c0} = 0$.
- Mostrar em um gráfico o comportamento das concentrações ao longo de 5 horas.

Reator batelada

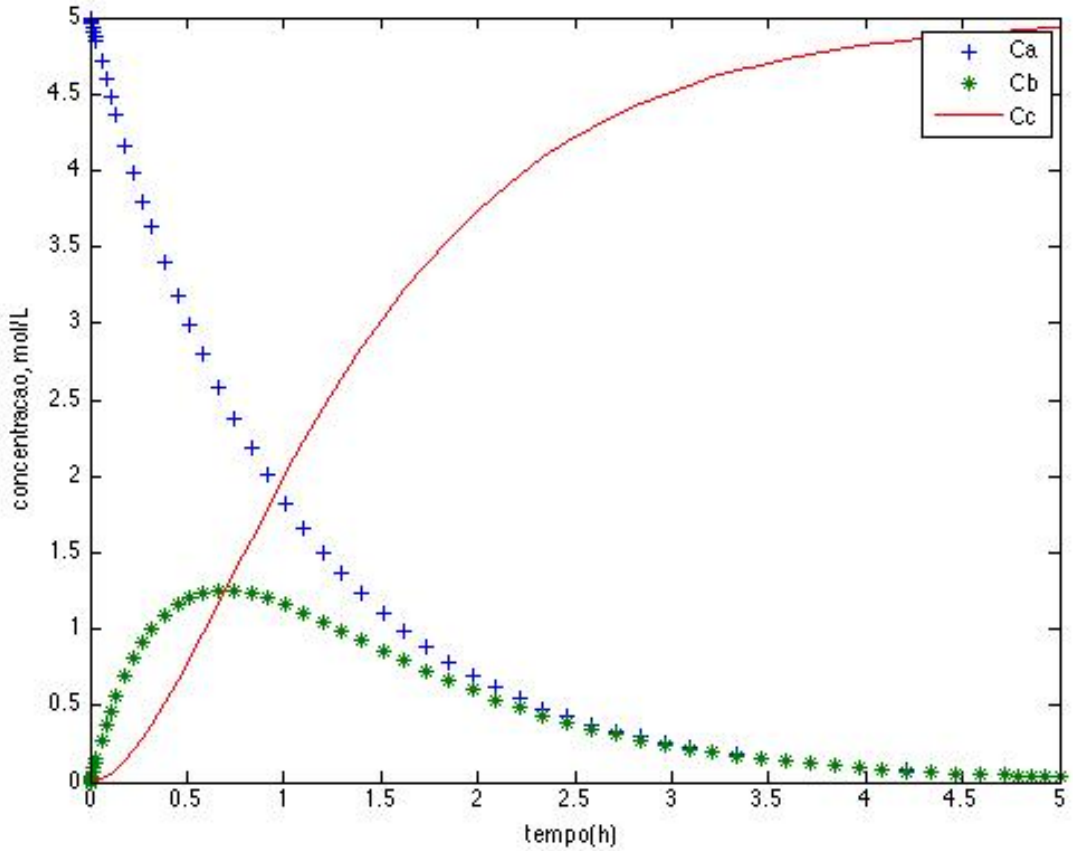


Figura 12: Curvas de concentração em função do tempo para o caso do reator batelada.

Sólido semi-inifinito

Descrição

- O problema de encontrar a distribuição transiente de temperaturas $T(x,t)$ em um sólido semi-infinito pode ser resolvido analiticamente. Sua solução é:

$$\theta(\eta,\tau) = \operatorname{erfc}[\eta/(2\tau)] - \exp(\eta+\tau^2)\operatorname{erfc}[\eta/(2\tau)+\tau]$$

com as variáveis adimensionais:

$$\theta(\eta,\tau) = \frac{T(\eta,\tau) - T_i}{T_\infty - T_i}$$

$$\tau = \frac{h}{k}\sqrt{\alpha t}$$

$$\eta = \frac{hx}{k}$$

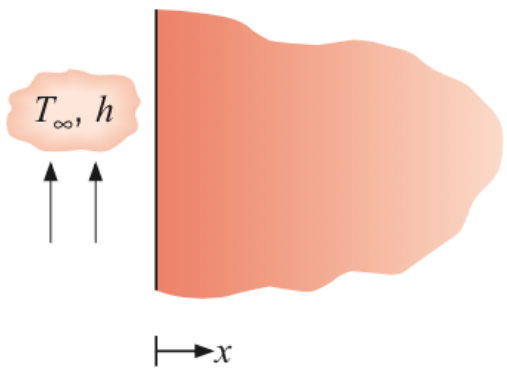


Figura 13: Sólido semi-infinito.

Sólido semi-inifinito

Descrição

- Sendo:
  - T_i a temperatura uniforme inicial do sólido,
  - T_∞ a temperatura ambiente do ar,
  - h o coeficiente de troca convectiva,
  - k a condutividade térmica do sólido,
  - α a difusividade térmica do sólido.
- erfc é a função *erro complementar*, muito comum na matemática e disponível no MATLAB.

Objetivos

- Analisar a distribuição de temperaturas, plotando:
  - um gráfico 3D $\theta \times \tau \times \eta$,
  - um gráfico 2D $\theta \times \tau$ para valores selecionados de η.

Sólido semi-infinito

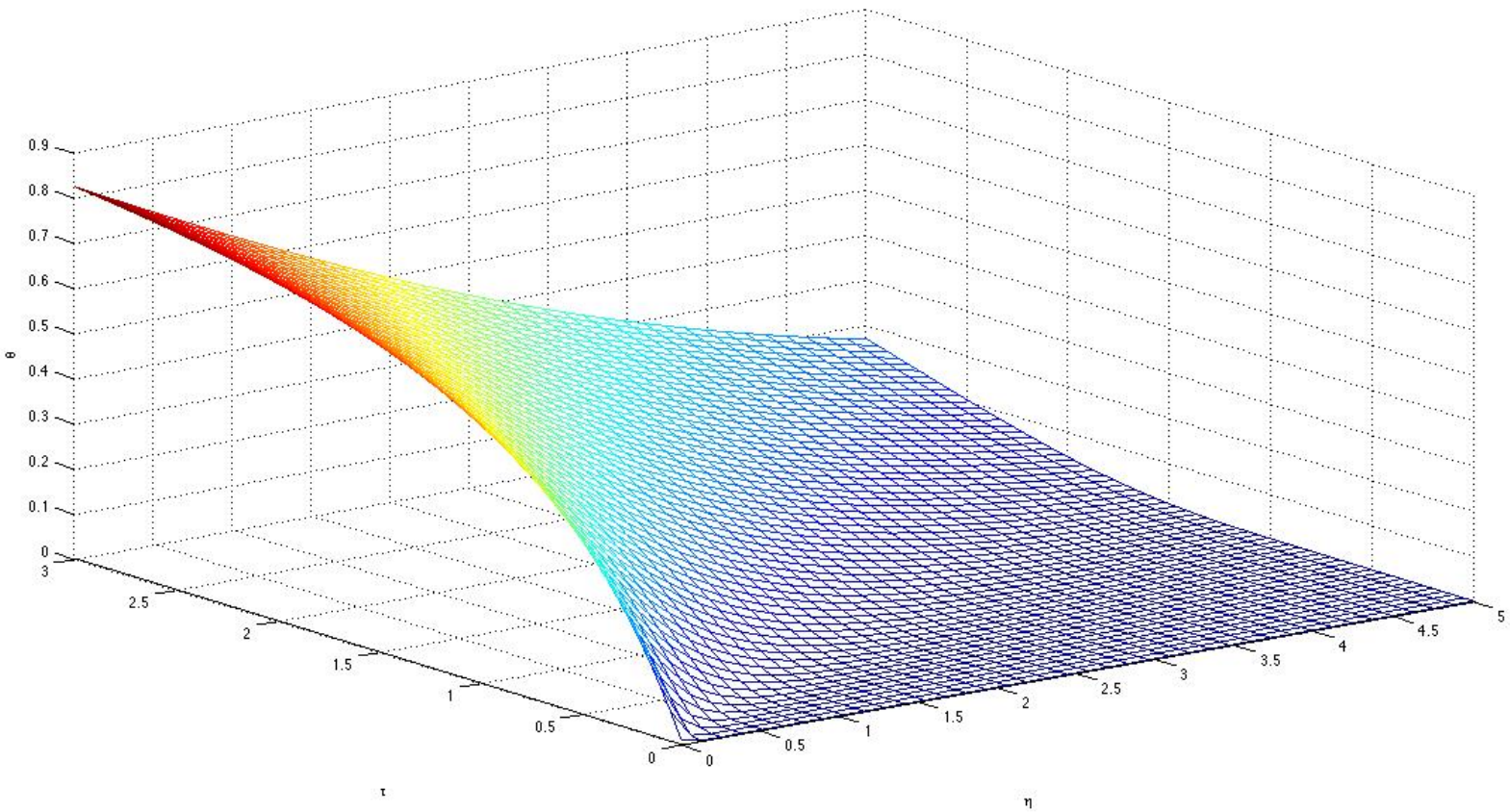


Figura 14: Temperatura em um sólido semi-infinito em função da posição adimensional η e do tempo adimensional τ.

Sólido semi-infinito

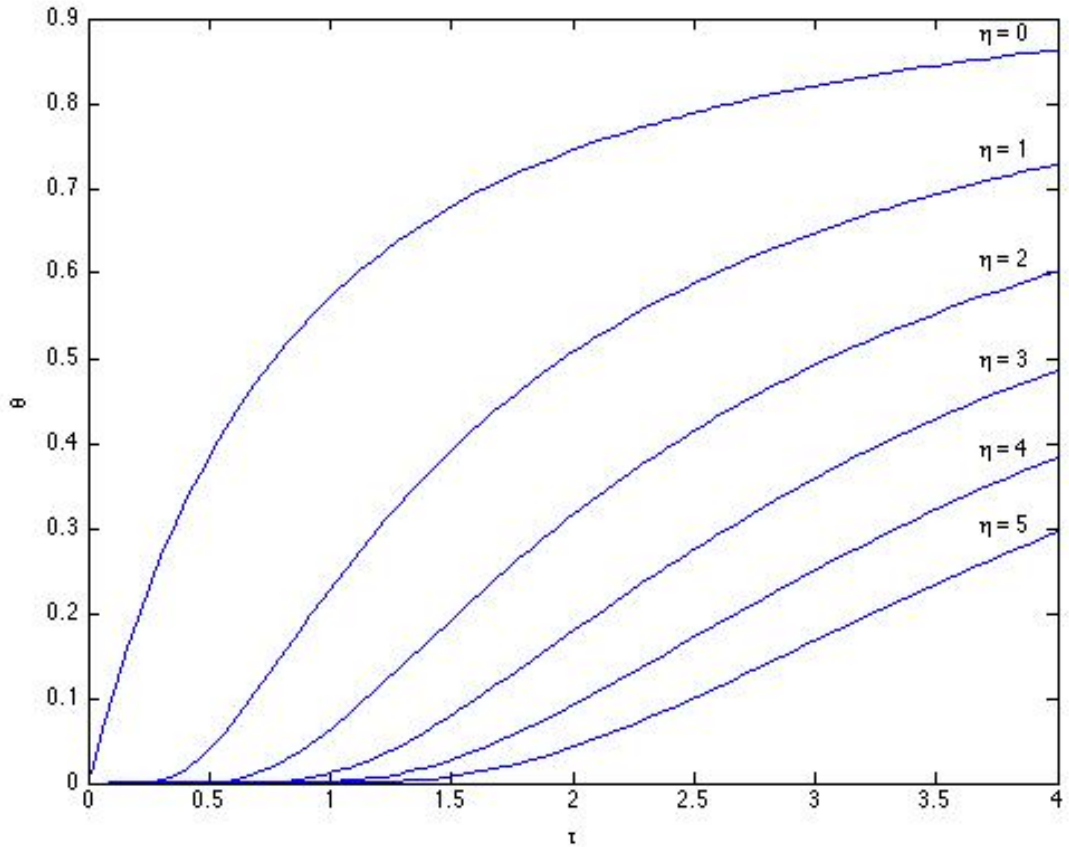


Figura 15: Temperatura em um sólido semi-infinito em função do tempo adimensional τ para diversas posições adimensionais η.

Aceleração de uma camada líquida

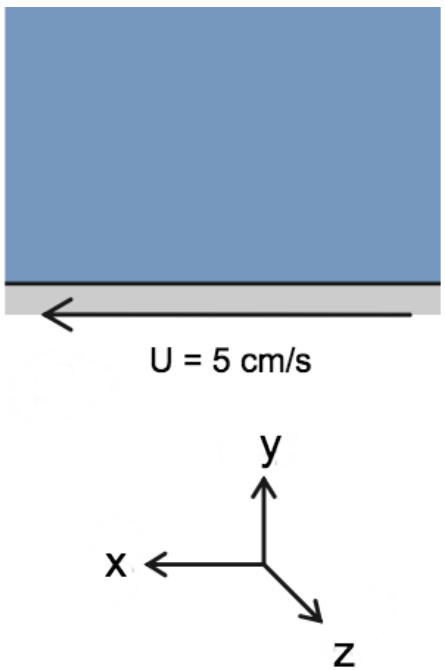


Figura 16: Aceleração de uma camada líquida.

Descrição

- Considere uma camada de líquido, inicialmente em repouso, de espessura $h = 10\,\text{cm}$ e viscosidade cinemática $\nu_{vis} = 1,0\,\text{cm}^2/\text{s}$, que se estende ao infinito no plano x-z, e é limitada por uma placa rígida em $y = 0$ e uma superfície livre em $y = h$. Em $t = 0$, a placa começa a se mover na direção x com velocidade $U = 5,0\,\text{cm/s}$.

Aceleração de uma camada líquida

Modelo

- Aplicando as equações de Navier-Stokes ao exemplo, chegamos a:

$$\frac{\partial u}{\partial t} = \nu_{vis} \frac{\partial^2 u}{\partial y^2}$$

com as condições inicial e de contorno:

$$u(y, 0) = 0$$

$$u(0, t) = U$$

$$\nu_{vis} \left(\frac{du}{dy} \right)_{y=h} = 0$$

Objetivo

- Traçar perfis de velocidade horizontal do fluido, y (cm) x u (cm/s), para vários intervalos de tempo de $t = 0$ até 100 s.

Aceleração de uma camada líquida

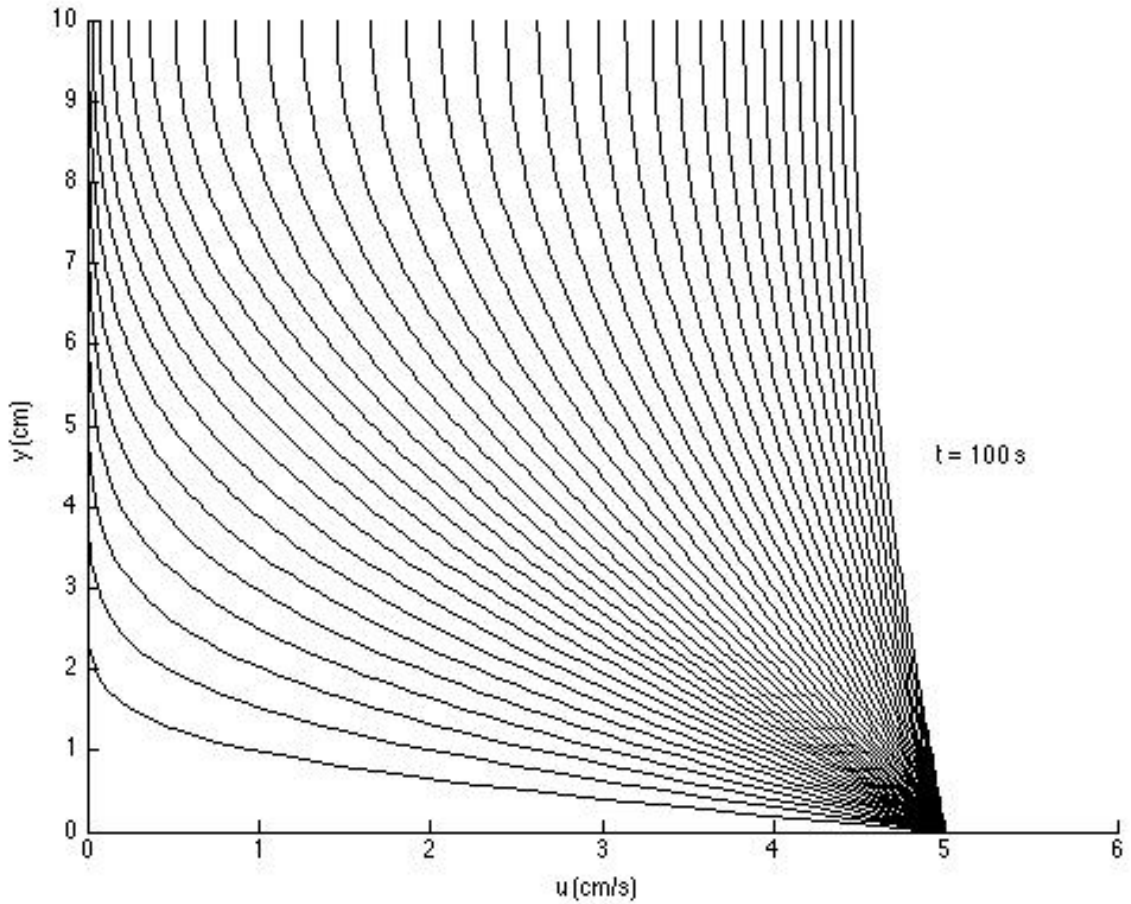


Figura 17: Velocidade horizontal de uma camada de fluido de espessura 10 cm que é repentinamente acelerada a uma velocidade de $U = 5,0$ cm/s.

Tanque de nível

Modelo

- Considere o seguinte modelo, que descreve como varia a altura de um tanque com válvula de saída em função do tempo:

$$\frac{dh(t)}{dt} = \alpha - \beta\sqrt{h(t)}$$

E sua versão linearizada em torno do estado estacionário ($h^* = \alpha^2/\beta^2$):

$$\frac{dh(t)}{dt} = -\frac{\beta^2}{2\alpha}(h(t) - \frac{\alpha^2}{\beta^2})$$

sendo $\alpha = F_e/A$ e $\beta = c/A$. Aqui temos $A = V/h$ como sendo a área da seção reta do tanque e c a constante da válvula.

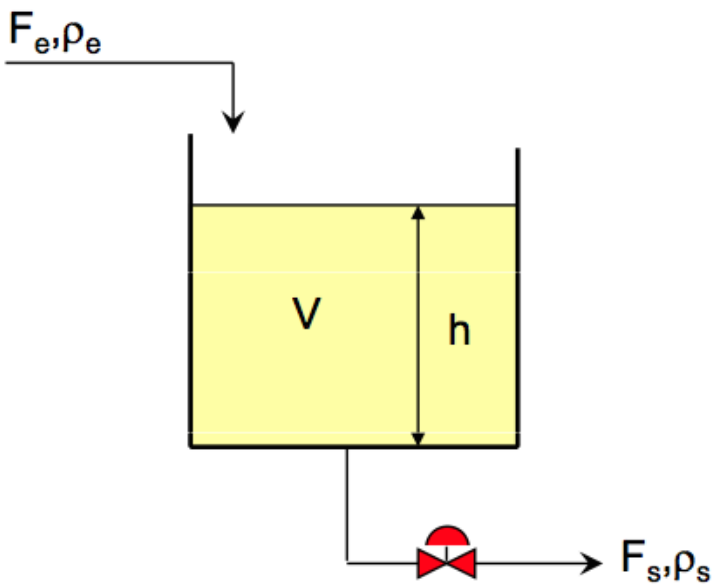


Figura 18: Tanque de nível com válvula de saída. Fonte: Argimiro Secchi.

Tanque de nível

Objetivos

- Nossos objetivos são:
  - comparar as duas formas de dh/dt para diversos valores de h, de modo a verificar em que região a aproximação linear é satisfatória;
  - plotar $t \times h(t)$ para os dois casos de modo a analisar o comportamento dinâmico dos dois modelos e observar a qualidade da aproximação.
- Usaremos como valores para os parâmetros $\alpha = 1$ e $\beta = 0,5$ (daí, resulta $h^* = \alpha^2/\beta^2 = 4$).

Tanque de nível

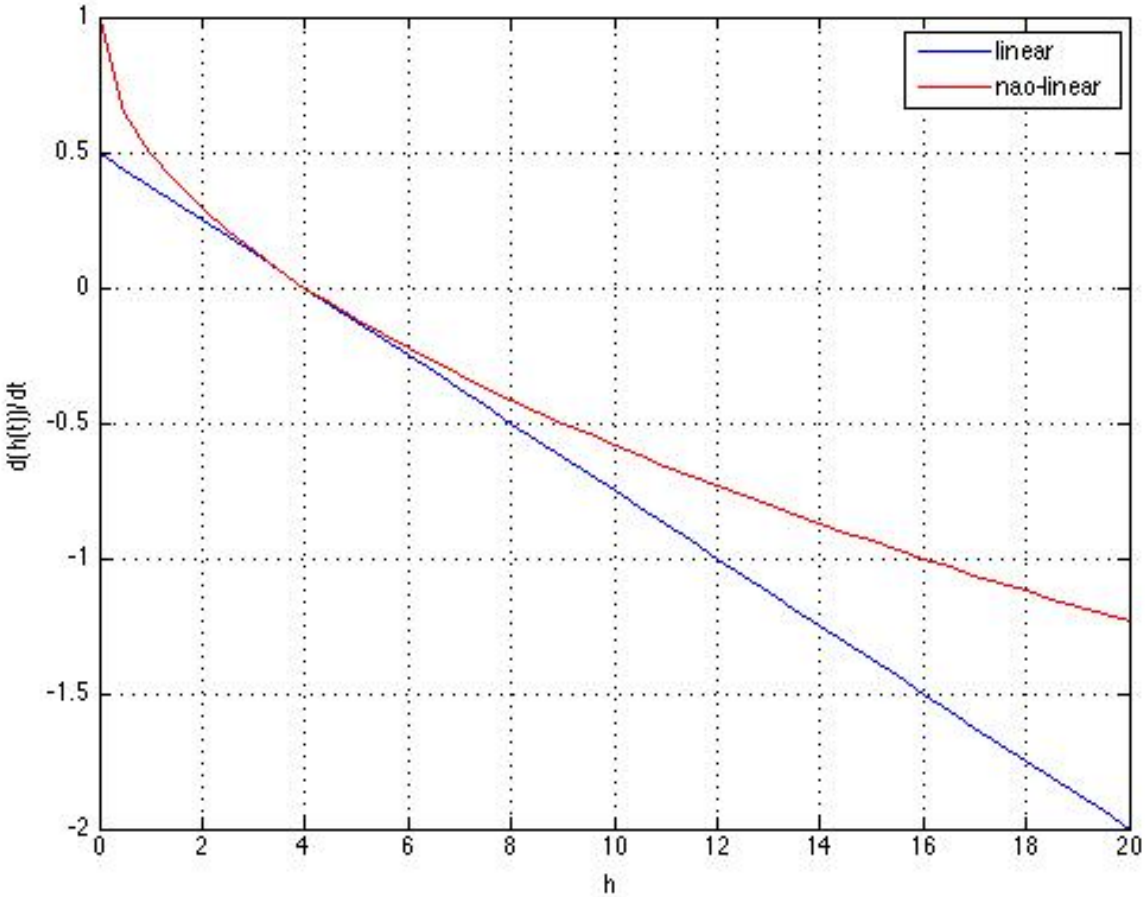


Figura 19: Altura x sua derivada para os dois modelos.

Tanque de nível

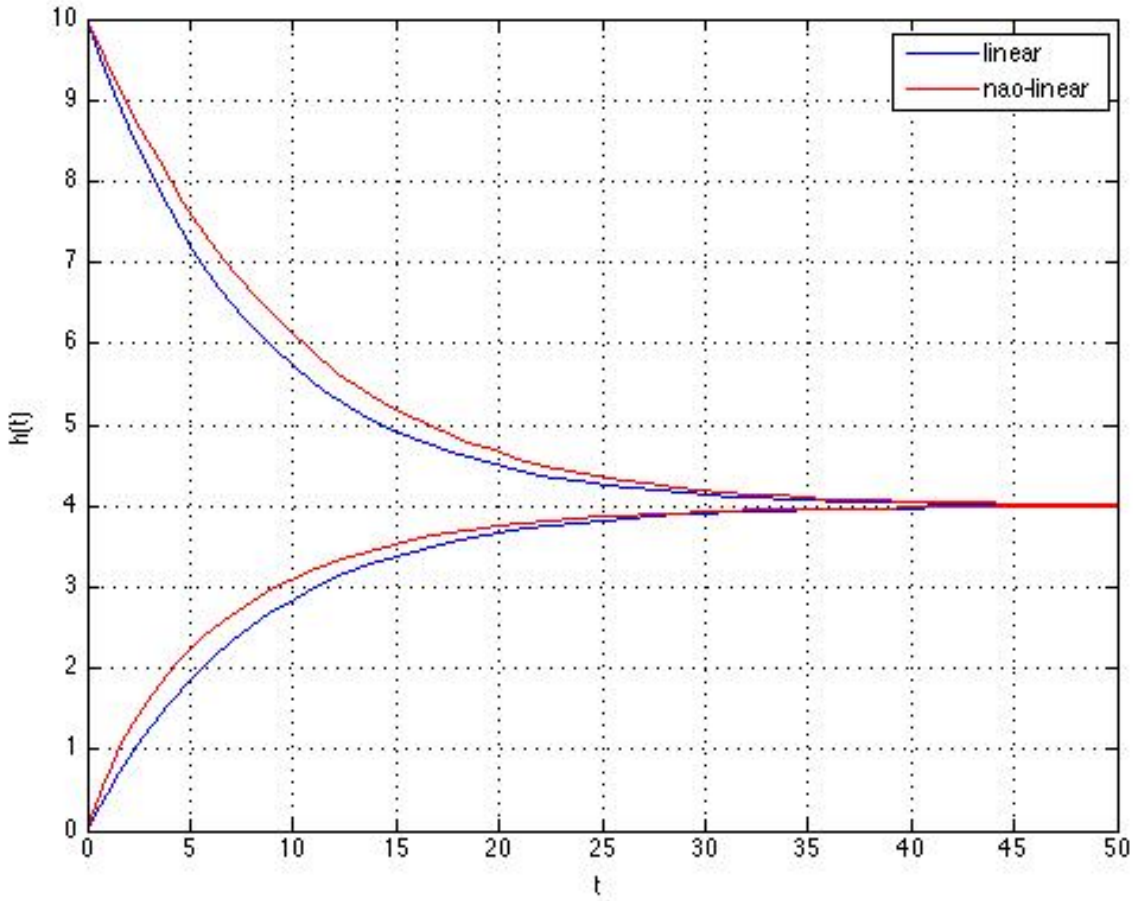


Figura 20: Comportamento dinâmico da altura para duas condições iniciais diferentes usando os dois modelos.

Ajuste de dados de pressão de vapor

Correlações empíricas

- Ao lado encontram-se dados experimentais da pressão de vapor do benzeno em função da temperatura. Ajuste-os às seguintes correlações:
  - Polinômio de quarto grau:

$$P = a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4$$

  - Equação de Clausius-Clapeyron:

$$log_{10} P = A - \frac{B}{T + 273,15}$$

  - Equação de Antoine:

$$log_{10} P = A - \frac{B}{T + C}$$

Temperature, T (°C)	Pressure, P (mm Hg)
-36.7	1
-19.6	5
-11.5	10
-2.6	20
+7.6	40
15.4	60
26.1	100
42.2	200
60.6	400
80.1	760

Figura 21: Dados experimentais de pressão de vapor para o benzeno.

Ajuste de dados de pressão de vapor

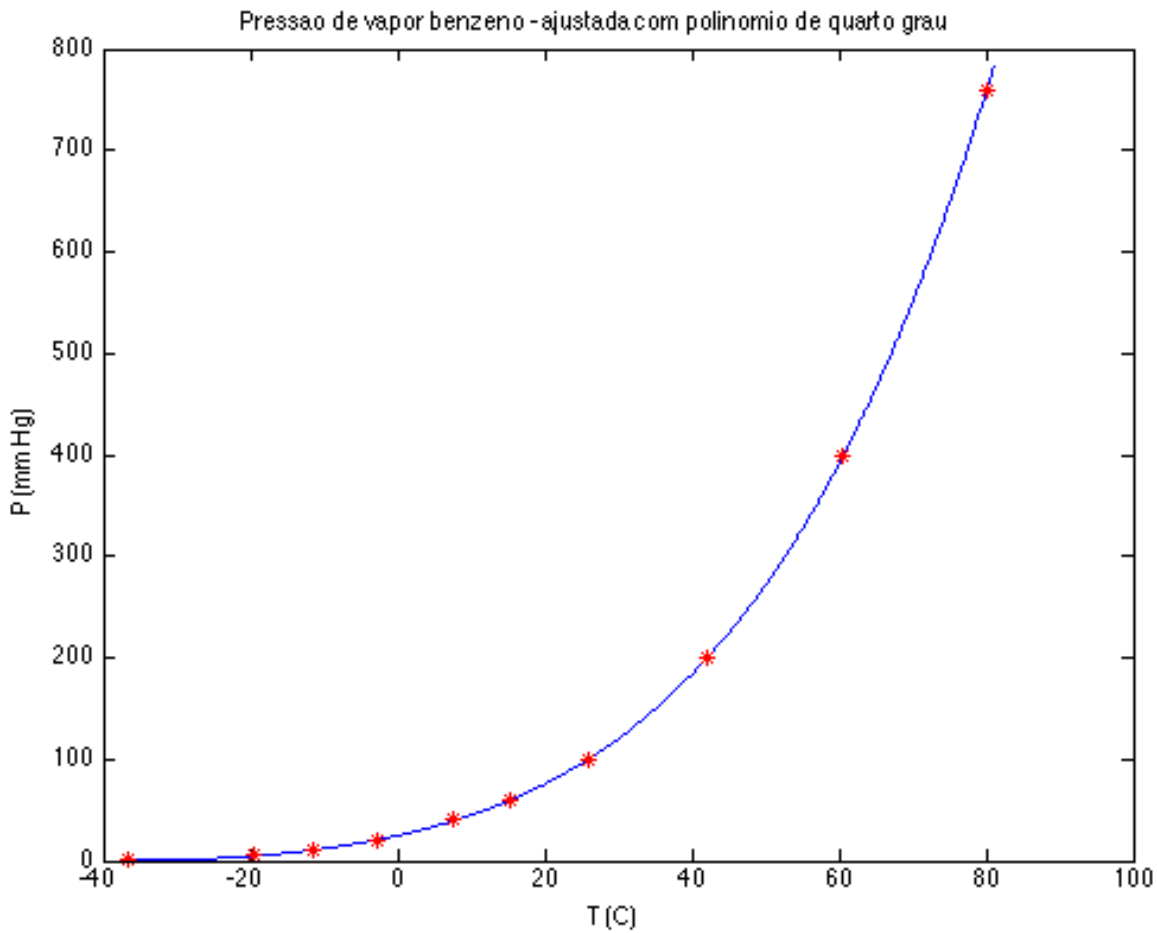


Figura 22: Ajuste dos dados de pressão de vapor do benzeno ao polinômio de quarto grau.

Ajuste de dados de pressão de vapor

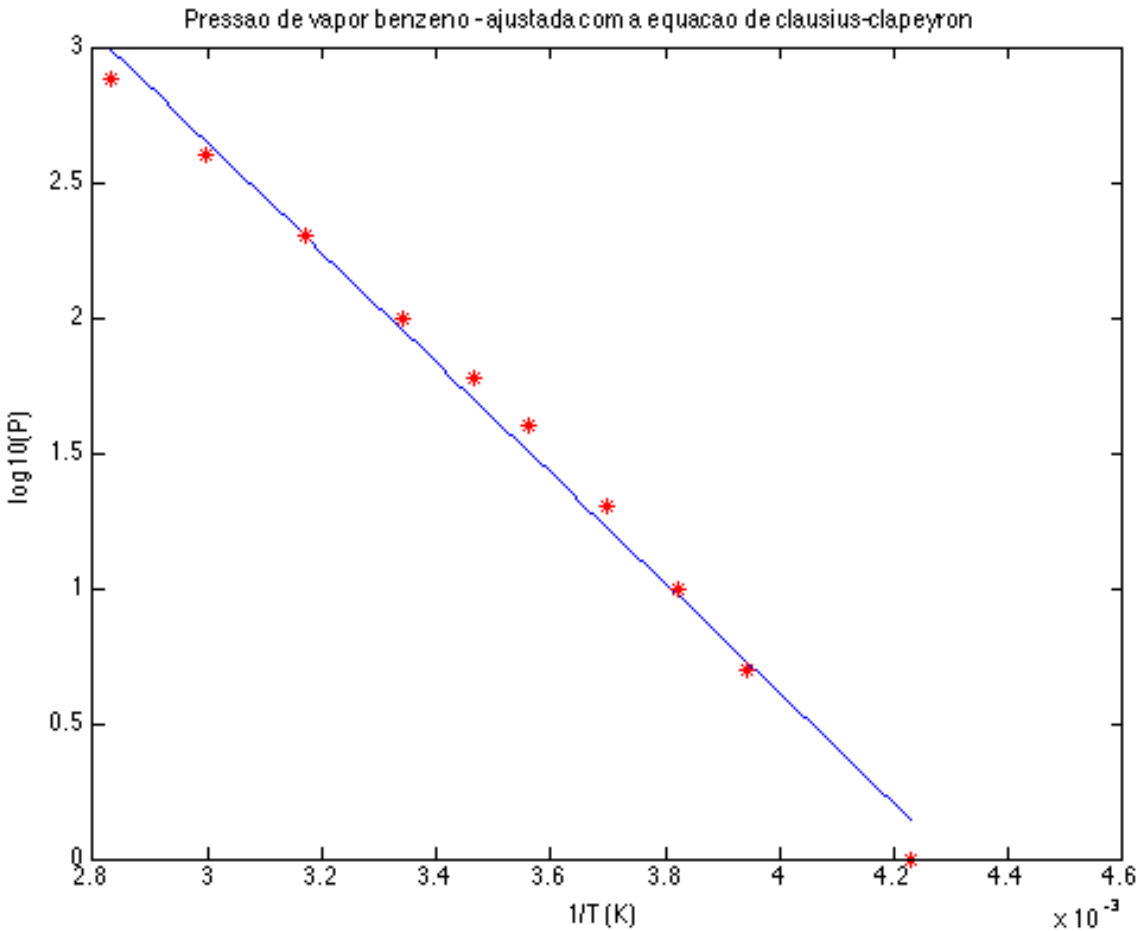


Figura 23: Ajuste dos dados de pressão de vapor do benzeno a equação de Clausius-Clapeyron.

Ajuste de dados de pressão de vapor

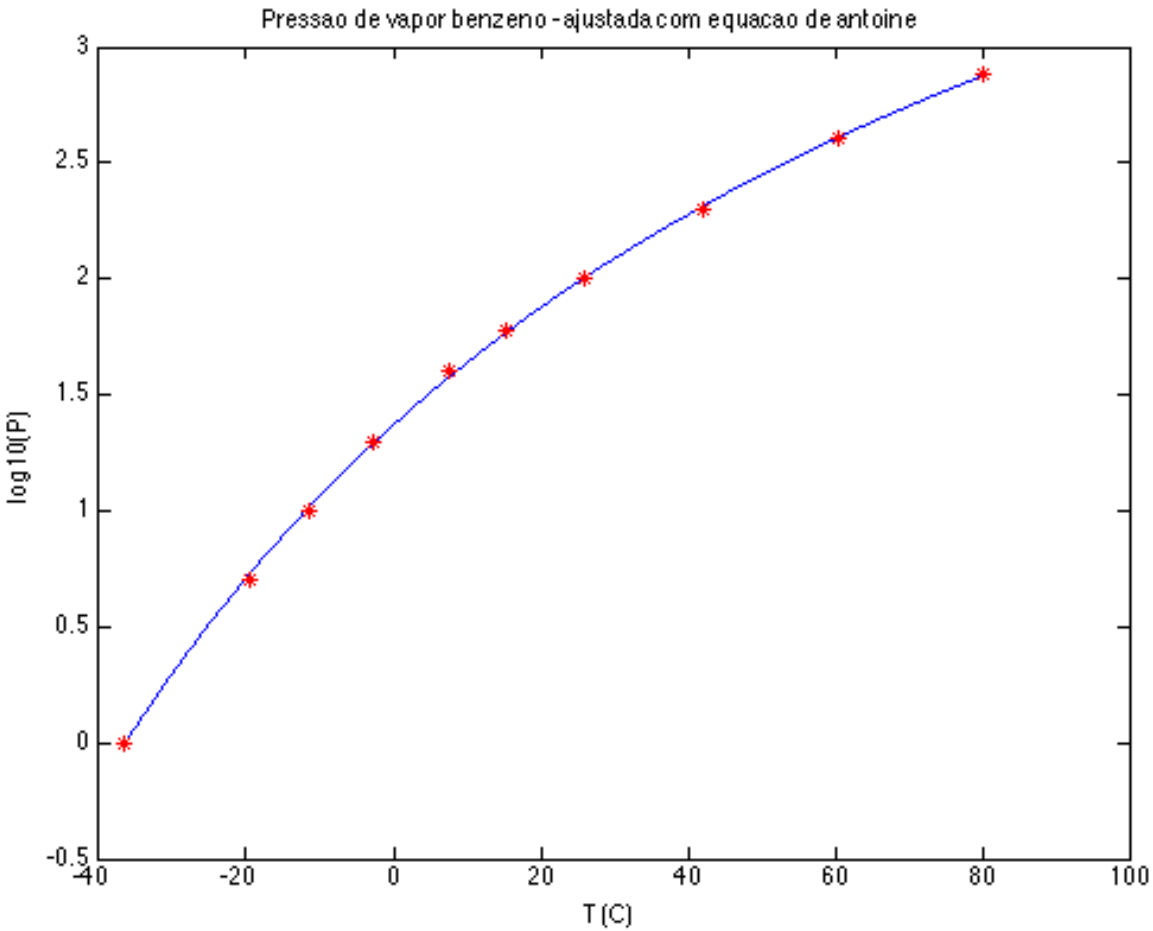


Figura 24: Ajuste dos dados de pressão de vapor do benzeno à equação de Antoine.

Cálculos de flash

Problema

- O problema de *flash* aqui estudado pode ser posto da seguinte maneira: dadas a composição global (z_i), temperatura (T) e pressão (P) de um sistema composto por duas fases líquidas, determinar as composições das fases em equilíbrio x_i e x_i^* e a razão entre as quantidades das fases (β).
- O acoplamento das equações de balanço material às relações de equilíbrio termodinâmico levam à chamada equação de *Rashford-Rice*:

$$\sum_i \frac{z_i}{\beta + K_i(1-\beta)} - 1 = 0$$

onde $K_i = x_i/x_i^* = \gamma_i^*/\gamma_i$ é o fator de equilíbrio e β a razão entre as quantidades das fases. Os valores de γ são fornecidos por um modelo de *coeficiente de atividade*.

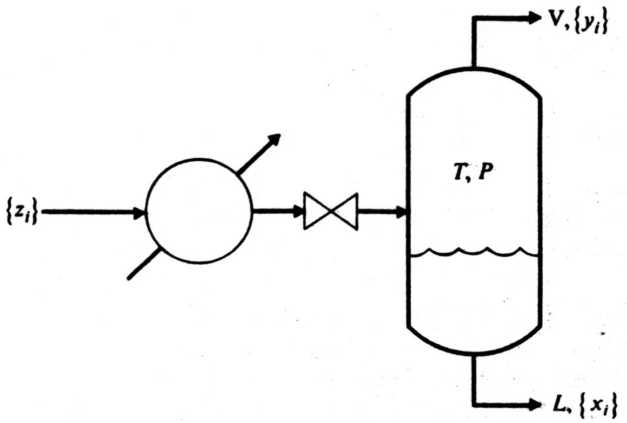


Figura 25: Fluxograma de um processo de separação em *flash*.

Flash - Método da substituição sucessiva

Resolvendo o flash

- Conhecidos todos os x_i, podemos obter x_i^* via balanço material, calcular os K_i e resolver a equação para β por um método numérico qualquer.
- Problema: não temos x_i! Necessidade de iteração.

Método da substituição sucessiva - Algoritmo (WALAS, 1985)

- 1 - Estimar a composição de uma das fases e o β;
- 2 - Calcular a composição da outra fase por meio do balanço material;
- 3 - Calcular todos os γ_i e K_i;
- 4 - Resolver a equação de Rashford-Rice e encontrar β;
- 5 - Achar x_i, os termos individuais do somatório. Retornar ao passo 3.

Flash - Minimização da energia de Gibbs

Formulação do problema

$$\underset{\mathbf{n}}{\text{Minimizar}} \quad G(\mathbf{n})$$

$$\text{sujeito a:} \quad n_i = \sum_{j=1}^{N_f} n_{ij} \qquad i = 1, \ldots, N_c.$$

$$0 \leq n_{ij} \leq n_i, \; i = 1, \ldots, N_c, \qquad j = 1, \ldots, N_f.$$

Função objetivo

- Minimizar $G(\mathbf{n})$ é equivalente a minimizar a seguinte função:

$$\Delta g(\mathbf{n}) = \sum_{j=1}^{N_f} \sum_{i=1}^{N_c} \frac{n_{ij}}{n} ln(x_{ij}\gamma_{ij})$$

chamada de energia de Gibbs adimensional.

Flash - Minimização da energia de Gibbs

Eliminando as restrições

- Podemos substituir a restrição de igualdade linear $n_i = \sum_{j=1}^{N_f} n_{ij}$ em Δg, de modo a diminuir N_c graus de liberdade do problema. A função objetivo resultante é:

$$\Delta g(\mathbf{n}) = \sum_{j=1}^{N_f-1} \sum_{i=1}^{N_c} \frac{n_{ij}}{n} ln(x_{ij}\gamma_{ij}) + \sum_{i=1}^{N_c} \frac{n_{i,Nf}}{n} ln(x_{i,N_f}\gamma_{i,N_f})$$

sendo:

$$x_{ij} = \frac{n_{ij}}{\sum_{i=1}^{N_c} n_{ij}} \quad , \quad n_{i,Nf} = n_i - \sum_{j=1}^{N_f-1} n_{ij}$$

- As únicas restrições restantes são os limites de busca das $N_c(N_f - 1)$ variáveis de otimização.

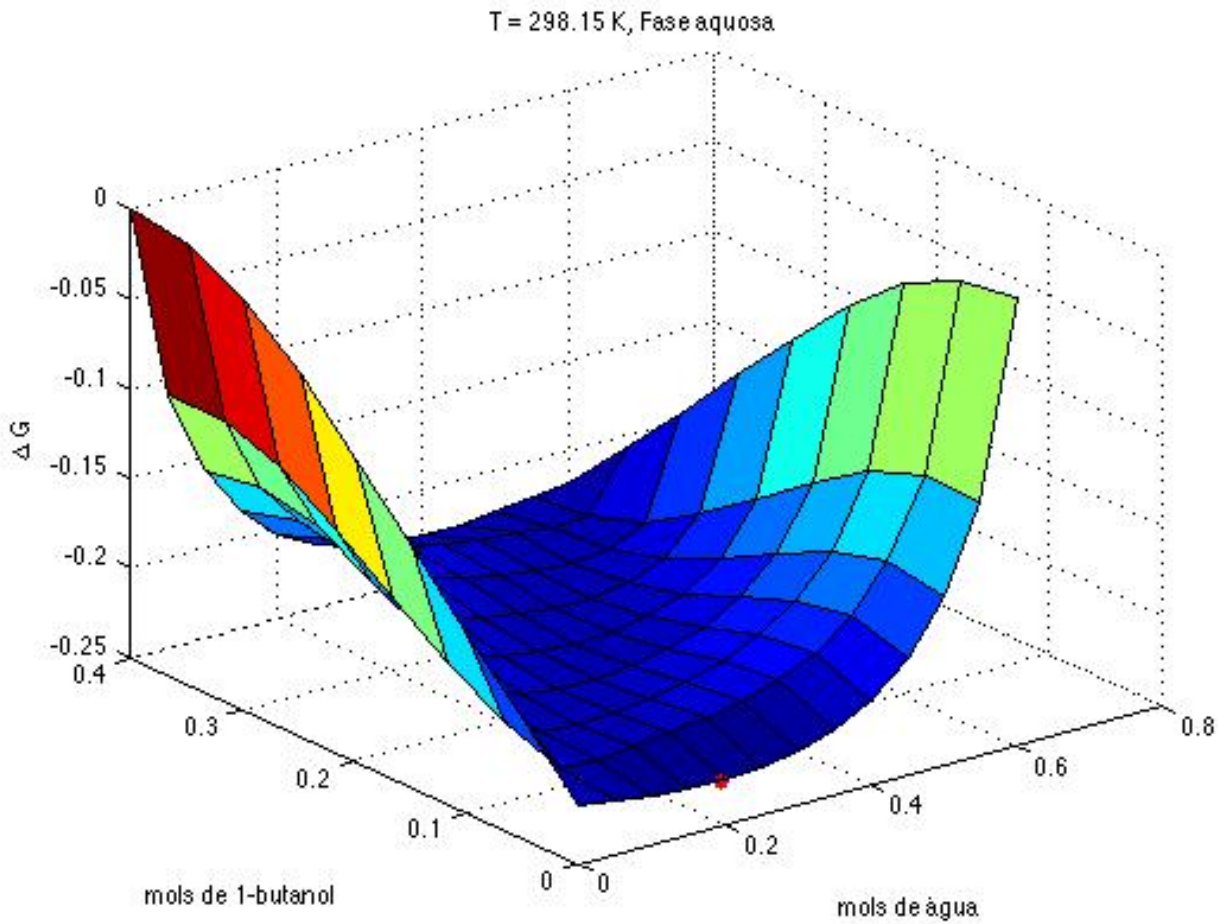


Figura 26: Energia de Gibbs adimensional de um sistema com 0,4 mols de 1-butanol e 0,6 mols de água a 298.15 K em função das quantidades dos componentes na fase aquosa.

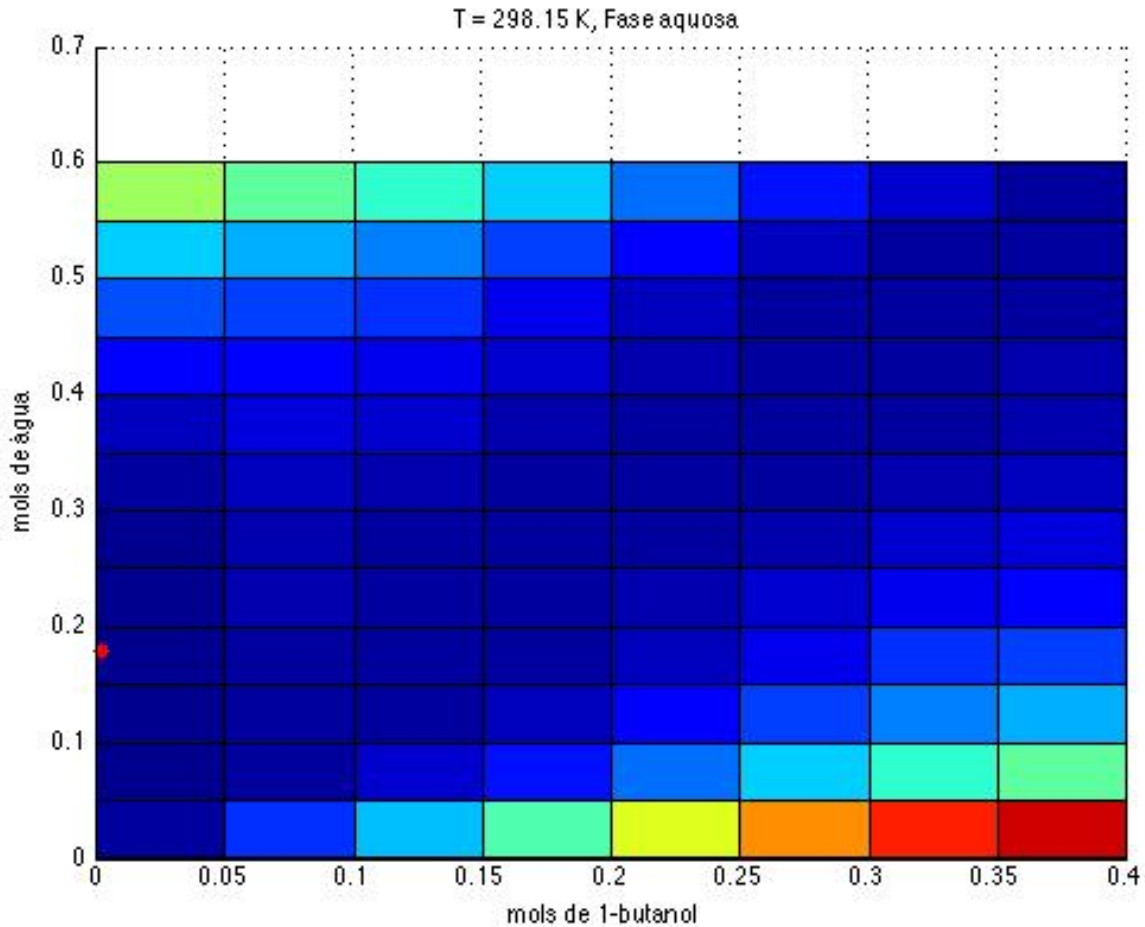


Figura 27: Mapa de cores da energia de Gibbs adimensional de um sistema com 0,4 mols de 1-butanol e 0,6 mols de água a 298.15 K em função das quantidades dos componentes na fase aquosa.

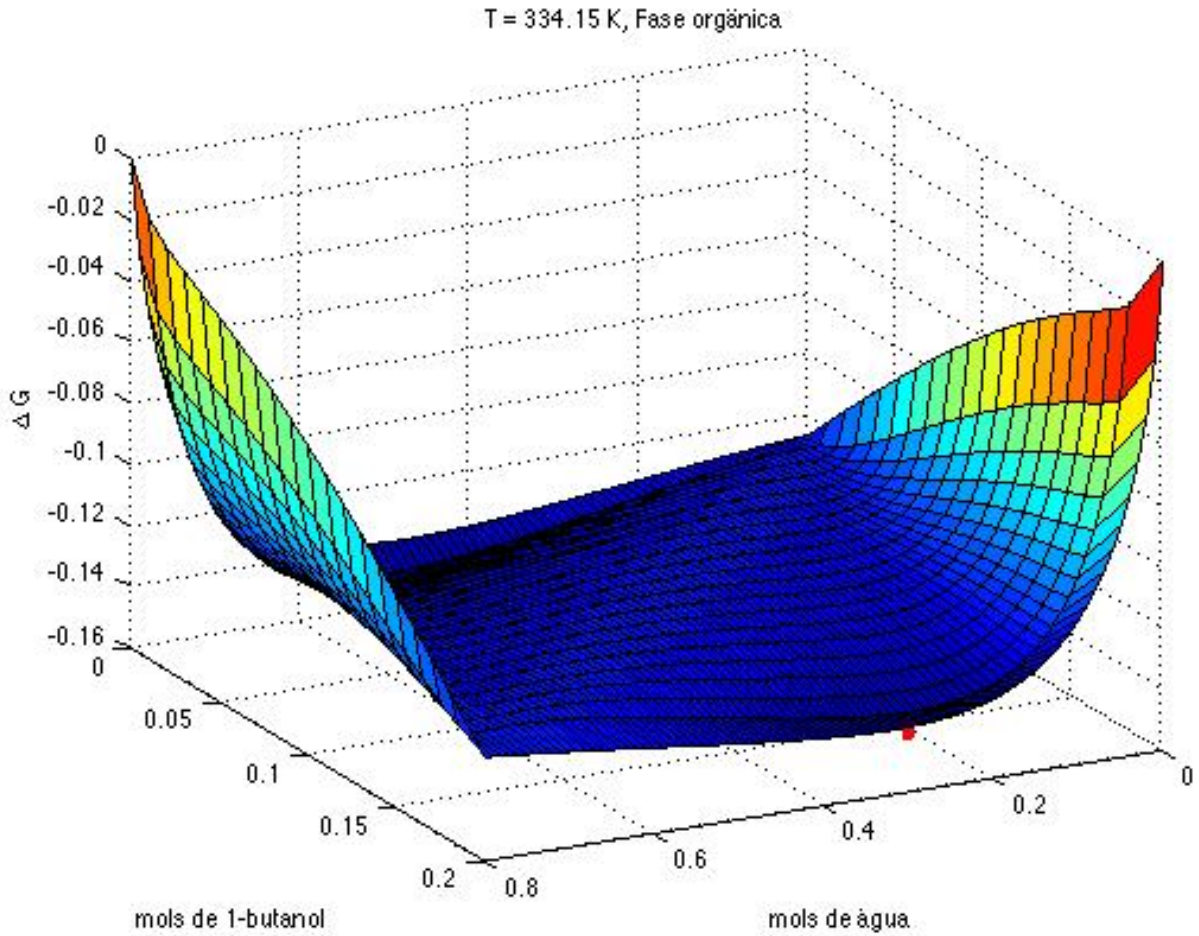


Figura 28: Energia de Gibbs adimensional de um sistema com 0,2 mols de 1-butanol e 0,8 mols de água a 334.15 K em função das quantidades dos componentes na fase orgânica.

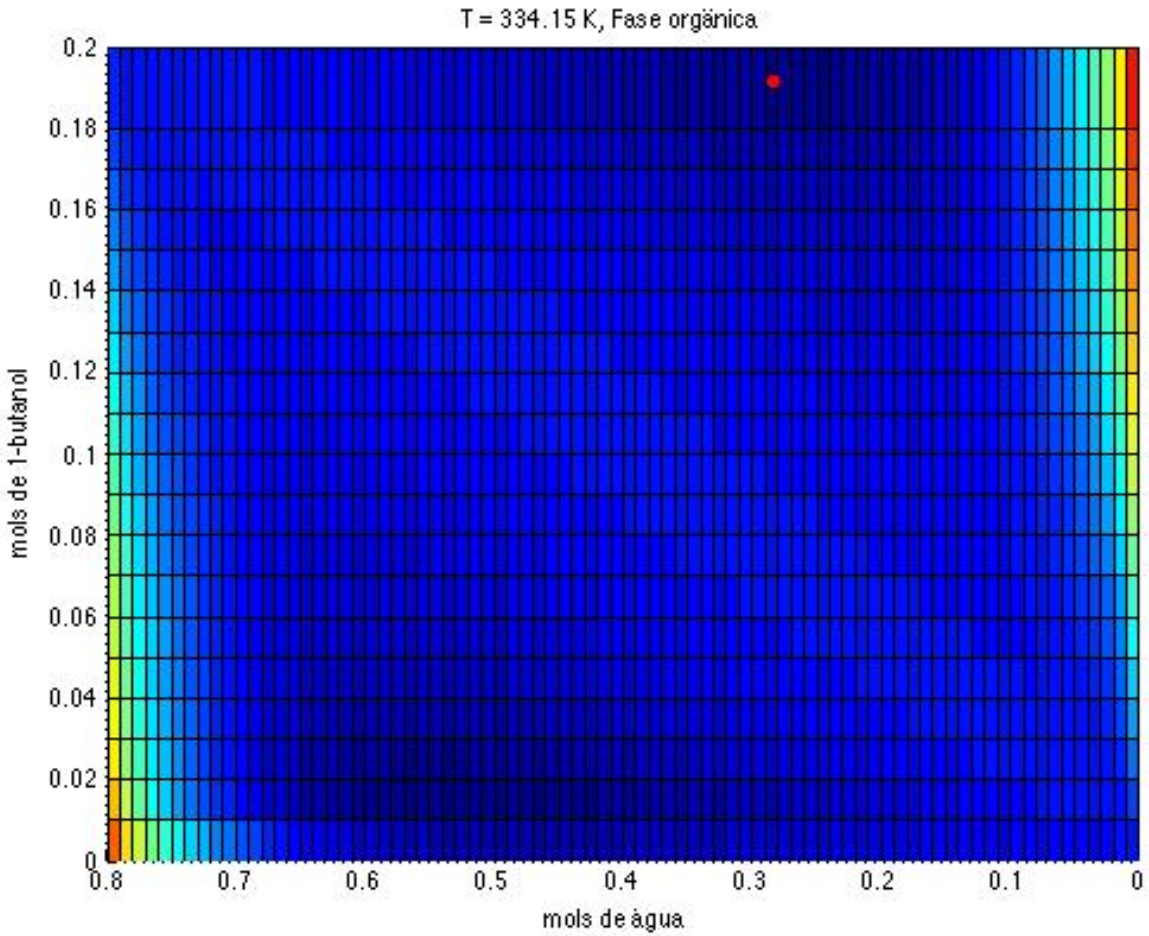


Figura 29: Mapa de cores da energia de Gibbs adimensional de um sistema com 0,2 mols de 1-butanol e 0,8 mols de água a 334.15 K em função das quantidades dos componentes na fase orgânica.

Reator PFR com dispersão axial

Modelo

- Seja o seguinte modelo, que descreve como varia a concentração de um componente ao longo de um reator *plug flow* com dispersão axial:

$$\frac{\partial C(t,z)}{\partial t} + v_z \frac{\partial C(t,z)}{\partial z} - D \frac{\partial^2 C(t,z)}{\partial z^2} - r(t,z) = 0$$

sujeito às condições inicial e de contorno:

$$C(0,z) = C_0(z)$$

$$C(t,0) - \frac{D}{v_z} \left. \frac{\partial C(t,z)}{\partial z} \right|_{z=0} = C_f(t)$$

$$\left. \frac{\partial C(t,z)}{\partial z} \right|_{z=L} = 0$$

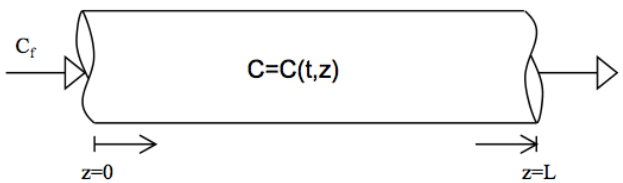


Figura 30: PFR com dispersão axial.
Fonte: Argimiro Secchi.

Reator PFR com dispersão axial

Discretização do domínio z

- Se $r(t,z)$ for não-linear, a equação diferencial parcial pode não ter solução analítica. Portanto, precisaremos usar métodos numéricos para resolvê-la, e o MATLAB é especialista nisso!
- Para aplicarmos o método numérico precisamos discretizar o domínio contínuo z e transformá-lo em um domínio discreto z_i. Ao invés de resolvermos:

$$\frac{\partial C(t,z)}{\partial t} + v_z \frac{\partial C(t,z)}{\partial z} - D\frac{\partial^2 C(t,z)}{\partial z^2} - r(t,z) = 0$$

para $0 < z < L$, resolveremos:

$$\left.\frac{\partial C(t,z)}{\partial t}\right|_{z=z_i} + v_z \left.\frac{\partial C(t,z)}{\partial z}\right|_{z=z_i} - D \left.\frac{\partial^2 C(t,z)}{\partial z^2}\right|_{z=z_i} - r(t,z)|_{z=z_i} = 0$$

para $i = 1, ..., N$.

Reator PFR com dispersão axial

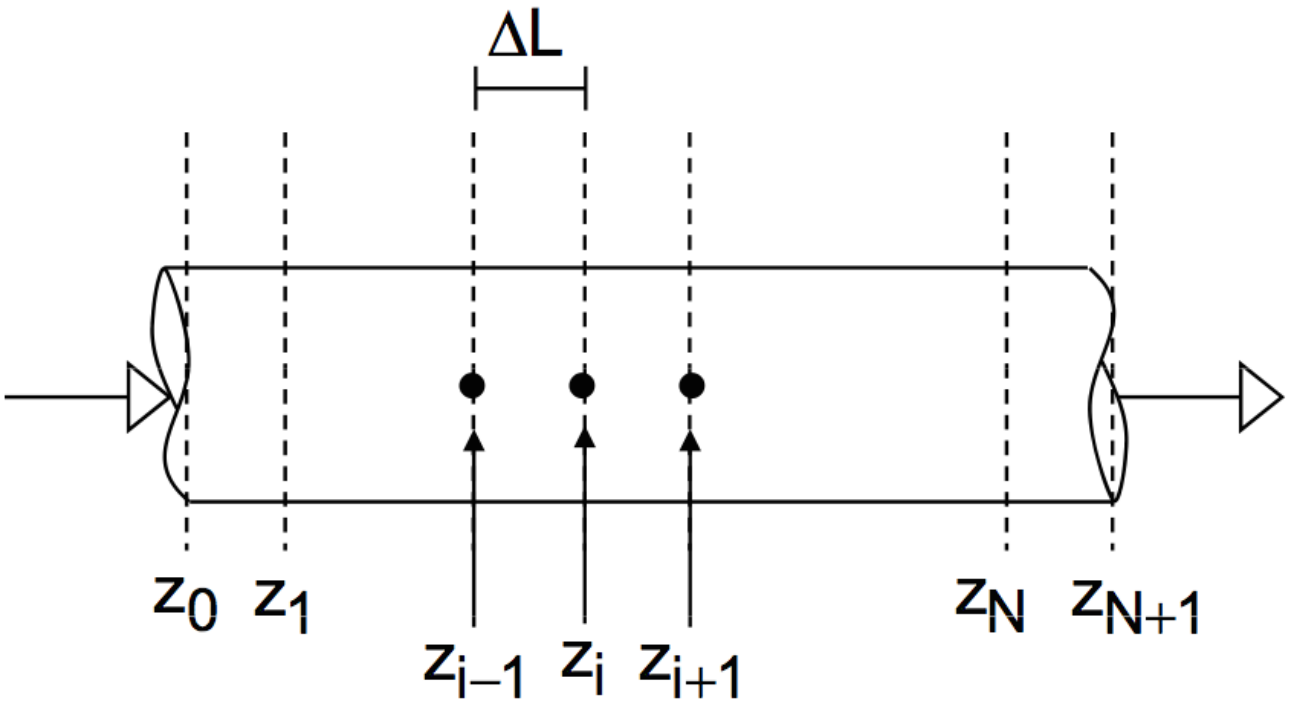


Figura 31: Discretização do domínio z do problema do PFR com dispersão axial. Fonte: Argimiro Secchi.

Reator PFR com dispersão axial

Aproximando as derivadas para $z = i$ (dentro do reator)

- Como a variável espacial foi discretizada, as derivadas parciais no tempo viram derivadas ordinárias:

$$\left.\frac{\partial C}{\partial t}\right|_{z=z_i} \approx \frac{dC_i}{dt}$$

e as derivadas no espaço podem ser aproximadas por diferenças finitas (no caso, *diferenças centrais*):

$$\left.\frac{\partial C}{\partial z}\right|_{z=z_i} \approx \frac{C_{i+1} - C_{i-1}}{2\Delta L}$$

$$\left.\frac{\partial^2 C}{\partial z^2}\right|_{z=z_i} \approx \frac{\left.\frac{\partial C}{\partial z}\right|_{z=z_i+\Delta L/2} - \left.\frac{\partial C}{\partial z}\right|_{z=z_i-\Delta L/2}}{\Delta L} = \frac{C_{i+1} - 2C_i + C_{i-1}}{\Delta L^2}$$

- A EDP se transforma em um sistema de EDO's válidas para $i = 1, ..., N$:

$$\frac{dC_i}{dt} = \frac{D}{\Delta L^2}(C_{i+1} - 2C_i + 2C_{i-1}) - \frac{v_z}{2\Delta L}(C_{i+1} - C_{i-1}) + r_i$$

Reator PFR com dispersão axial

Aproximando as derivadas para $z = 0$ e $z = N + 1$ (condições de contorno)

- Utilizando novamente diferenças finitas (desta vez *para trás*):

$$\left.\frac{\partial C}{\partial z}\right|_{z=0} \approx \frac{C_1 - C_0}{\Delta L}$$

$$\left.\frac{\partial C}{\partial z}\right|_{z=L} \approx \frac{C_{N+1} - C_N}{\Delta L}$$

Substituindo nas expressões das condições de contorno do problema, obtemos finalmente:

$$C_0 = \left(1 + \frac{D}{v_z \Delta L}\right)^{-1} \left(\frac{D}{v_z \Delta L} C_1 + C_f\right)$$

$$C_{N+1} = C_N$$

Reator PFR com dispersão axial

Modelo discretizado

- O modelo final se torna, portanto:

$$\frac{dC_1(t)}{dt} = \frac{D}{\Delta L^2}(C_2 - 2C_1 + C_0) - \frac{v_z}{2\Delta L}(C_2 - C_0) + r_1$$

$$\frac{dC_i(t)}{dt} = \frac{D}{\Delta L^2}(C_{i+1} - 2C_i + C_{i-1}) - \frac{v_z}{2\Delta L}(C_{i+1} - C_{i-1}) + r_i, \quad i = 2, ...N-1$$

$$\frac{dC_N(t)}{dt} = \frac{D}{\Delta L^2}(C_{N+1} - 2C_N + C_{N-1}) - \frac{v_z}{2\Delta L}(C_{N+1} - C_{N-1}) + r_N$$

sujeito às condições inicial e de contorno:

$$C_i(0) = C_{i0}, \quad i = 1...N$$

$$C_0 = \left(1 + \frac{D}{v_z \Delta L}\right)^{-1} \left(\frac{D}{v_z \Delta L} C_1 + C_f\right)$$

$$C_{N+1} = C_N$$

- Esta é a forma do modelo a ser implementada e resolvida no MATLAB!

Reator PFR com dispersão axial

Resolvendo o modelo

- Para resolver as equações, é necessário definir:
  - difusividade D;
  - velocidade axial v_z;
  - forma para a equação da taxa (p. ex, $r = -kC$, primeira ordem);
  - constante da reação k;
  - concentração da alimentação C_f;
  - comprimento do reator L;
  - condição inicial (vetor de concentrações para cada ponto do reator no tempo $t = 0$) ;
  - número de pontos em que o domínio será discretizado;
  - tempo de integração.
- Nosso objetivo é plotar a concentração de saída do reator em função do tempo.

Reator PFR com dispersão axial

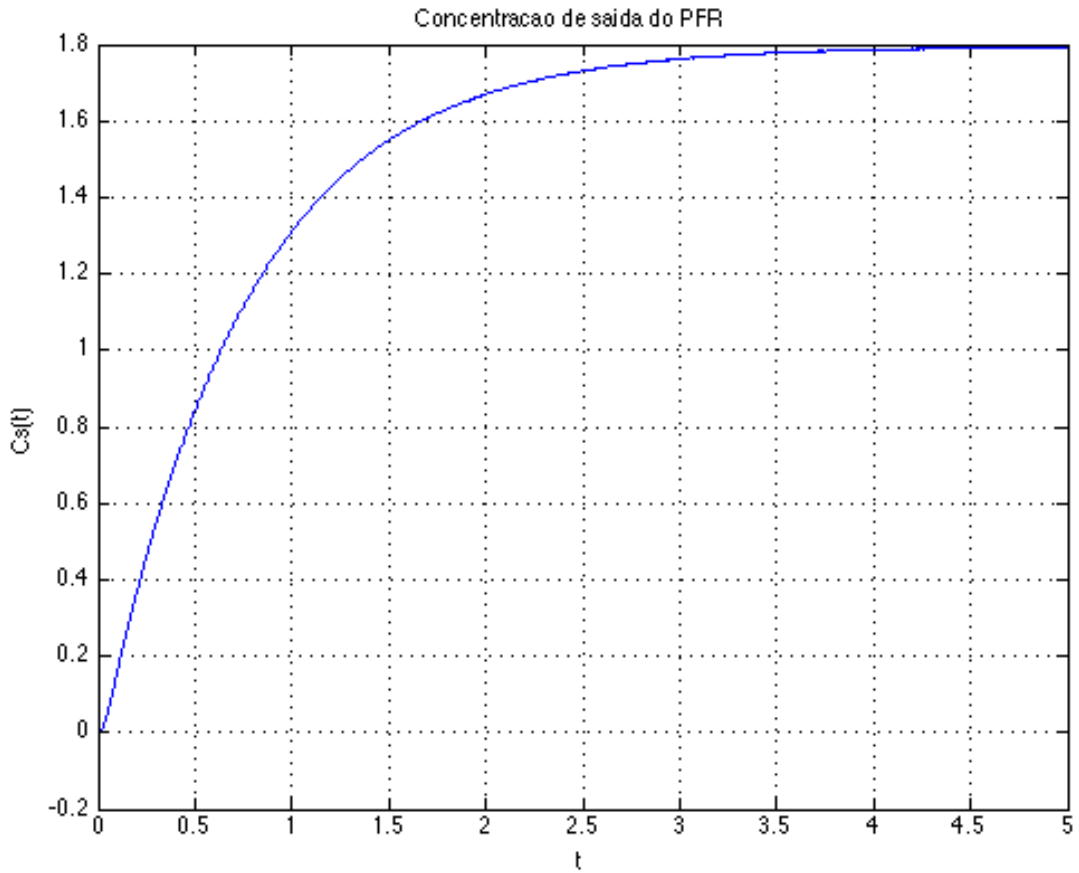


Figura 32: Concentração de saída do reator PFR em função do tempo.

Bibliografia


Stormy Attaway.
MATLAB® - A Practical Introduction to Programming and Problem Solving.
Elsevier, 2 edition, 2012.


Stephen J. Chapman.
MATLAB® Programming for Engineers.
Bookware Companion Series, 4 edition, 2007.


Bruce A. Finlayson.
Introduction to Chemical Engineering Computing.
Wiley-Interscience, 2006.


Brian R. Hunt, Ronald L. Lipsman, and Jonathan M. Rosenberg.
A Guide to MATLAB® for Beginners and Experienced Users.
Cambridge University Press, 3 edition, 2014.


Edward B. Magrab, Shapour Azarm, Balakumar Balachandran, James Duncan, Keith Herold, and Gregory Walsh.
An Engineer's Guide to MATLAB® - with Applications from Mechanical, Aerospace, Electrical, Civil and Biological Systems Engineering.
Prentice Hall, 3 edition, 2011.