2 As Leis de Newton

2.2 Vetores

2.2.1 Conceitos básicos

Nesta seção, apresentamos os conceitos básicos sobre vetores.

Uma força é um exemplo típico de grandeza que é representada por um vetor. Outros exemplos são deslocamento e velocidade. O conceito de vetor pode-se desenvolver de uma forma bem ampla, de modo que, por exemplo, soluções de sistemas de equações lineares ou de equações diferenciais também possam ser representadas por vetores.

Já sabemos que um vetor no espaço é uma combinação de um *comprimento* (número real positivo), uma *direção* e um *sentido*. (Linhas paralelas orientadas num mesmo sentido definem uma única *direção orientada*, naquele sentido, enquanto que linhas paralelas, orientadas em sentidos opostos, definem duas direções orientadas em sentidos opostos.)

Dois vetores serão *iguais* quando forem caraterizados pelo mesmo comprimento, direção e sentido; por isso, vetores iguais podem ser representados pelo mesmo segmento de reta orientado $\overline{PQ}$ ou por dois segmentos de reta paralelos $\overline{PQ}$ e $\overline{P'Q'}$ de mesmo comprimento e mesma orientação.

De modo geral, vetores são indicados por negritos **a, b, v, F** etc., ou por um símbolo como $\overline{PQ}$.

Este vetor podemos considerar como vetor deslocamento de uma partícula movendo-se ao longo de uma reta. **P** será a posição inicial e **Q** a posição final.

O deslocamento não depende da trajetória da pertícula, mas apenas dos pontos inicial e final.

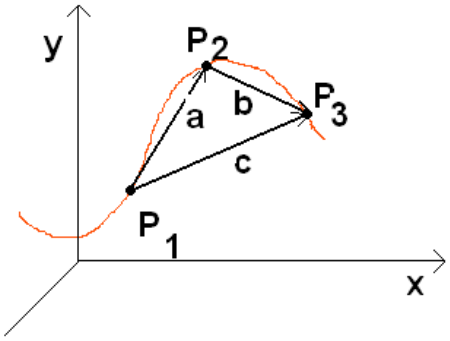

Fig. 2.2-1

O deslocamento **c** é equivalente aos deslocamentos sucessivos **a** e **b**; isto é, **c = a + b**.

Aqui formamos a **soma** de vetores pela "regra do polígono vetorial". Esse método consiste em conectar a origem de um vetor na extremidade do outro. A soma, ou a resultante, é o vetor que vai do ponto de origem à extremidade do último vetor adicionado.

Outro método para a adição vetorial é a "regra do paralelogramo". A soma de dois vetores, que formam entre si um ângulo qualquer, pode ser representada pela diagonal dem um paralelogramo, construído a partir da conexão entre as origens dos vetores.

2.2.2 Representação de vetores

Para representar um vetor em MuPAD, precisamos das coordenadas dos pontos P e Q.

Na figura seguinte desenhamos o vetor $\mathbf{a} = \overline{PQ}$, onde P = (1,2) e Q = (2.5,5).

- ```
  v1:= plot::Arrow2d([1, 2],[2.5,5],Color=RGB::Red):
  plot(v1)
  ```

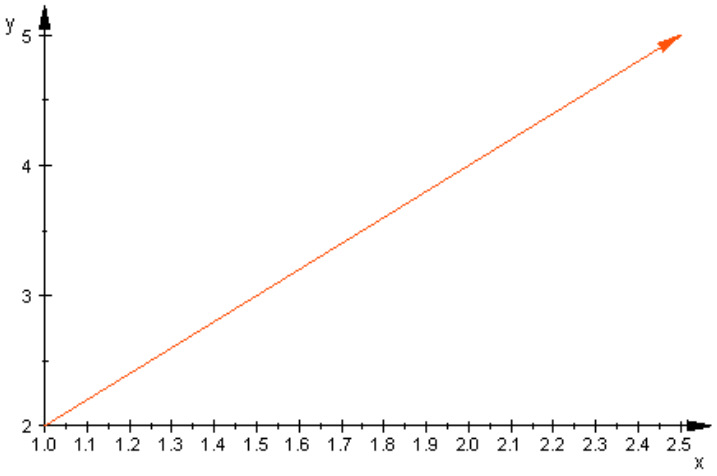

Fig. 2.2-2

Compare também a seguinte figura:

- ```
  v1:= plot::Arrow2d([1, 2],[2.5,5],Color=RGB::Green):
  plot(v1,Axes=None)
  ```

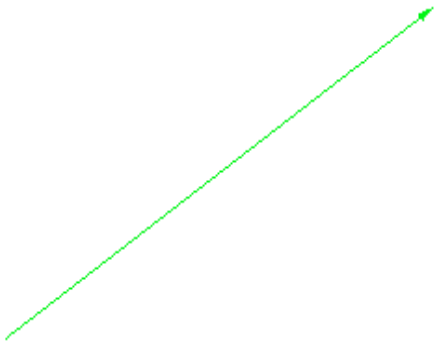

Fig. 2.2-3

Agora desenhamos três vetores posição com o mesmo origem, **O** =(0,0), utilizando diferentes estilos:

- ```
  plot(plot::Arrow2d([1, 1], Color = RGB::Red,
  TipStyle = Open, TipLength = 10*unit::mm),
  plot::Arrow2d([-1, 1], Color = RGB::Green,
  LineWidth = 0.8*unit::mm,
  ```
  ```

```
TipStyle = Closed, TipAngle = PI/2),
plot::Arrow2d([0, -sqrt(2)], Color = RGB::Blue,
LineStyle = Dashed))
```

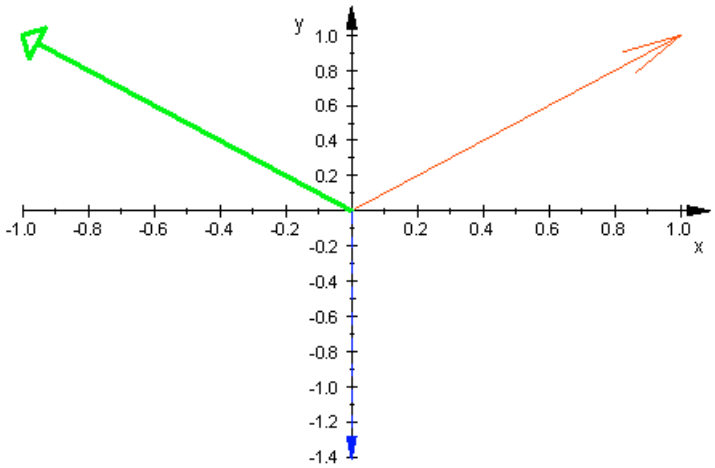

Fig. 2.2-4

A soma dos vetores com os pontos finais x1= (2,2) e x2 = (2.5,5) apresentamos da seguinte maneira:

- ```
  x1:=matrix([[2,2]]):x2:=matrix([[2.5,5]]):
  x3:= x1+x2;
  v1:=plot::Arrow2d([x1[1],x1[2]]):
  v2:=plot::Arrow2d([x2[1],x2[2]]):
  v3:=plot::Arrow2d([x3[1],x3[2]],Color=RGB::Red,TipStyle =
  Open):
  info1:=plot::Text2d("Vetor 1", [x1[1],x1[2]],
  HorizontalAlignment = Left,
  VerticalAlignment = BaseLine):
  info2:=plot::Text2d("Vetor 2", [x2[1],x2[2]],
  HorizontalAlignment = Left,
  VerticalAlignment = BaseLine):
  ```
  ```

```
info3:=plot::Text2d("Soma", [x3[1],x3[2]],
HorizontalAlignment = Left,
VerticalAlignment = BaseLine):
plot(v1,v2,v3,info1,info2,info3)
```

(4.5 7)

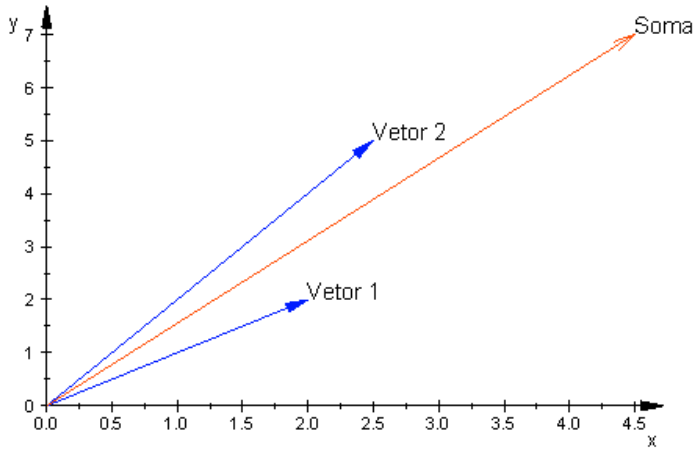

Fig. 2.2-5

Para MuPAD, um vetor é uma matriz-coluna ou uma matriz-linha.

A matriz-coluna é aquela que possui uma única coluna (n = 1), matriz-linha é aquela onde m = 1.
(m = número de linhas, n = número de colunas).

Exemplos:

Matriz-Coluna (m = 3, n = 1):

$$\begin{pmatrix} 1 \\ -5 \\ 7 \end{pmatrix}$$

MuPAd pede para esta matriz a forma

```
A:= matrix([[1]],[-5],[7]])
```

ou, mais simples:

```
A:= matrix([1,-5,7])
```

Matriz-Linha: (m = 1, n = 3):

$$\begin{pmatrix} 3 & -2 & 1 \end{pmatrix}$$

A notação em MuPAD é

```
A:= matrix([[3,-2,1]])
```

2.2.3 Representação analítica de vetores

A representação gráfica de vetores é inconveniente. É muito mais prático representar os vetores de forma analítica a partir de uma base. As operações com vetores ficam especialmente simples se os vetores-base são ortogonais entre si. Assim, temos o que se denomina *representação cartesiana* de vetores.

Em duas dimensões, um vetor **a** terá a forma analítica $\mathbf{a} = a_x\,\mathbf{i} + a_y\,\mathbf{j}$.
Os vetores unitários **i, j** são a base da representação. Os vetores unitários possuem módulos unitários e estão dirigidos nos sentidos positivos dos eixos x, y, respectivamente, em um sistema de coordenadas dextrogiro.

$a_x\,\mathbf{i}$ e $a_y\,\mathbf{j}$ são as componentes vetoriais de **a**, e a_x , a_y são as suas componentes escalares (às vezes denominadas de *coordenadas* de **a**).
As componentes escalares são dadas analiticamente por

$$a_x = a\cos\theta \qquad (2.2\text{-}1a)$$

$$a_y = a\,\mathrm{sen}\theta \qquad (2.2\text{-}1b)$$

O teorema de Pitágoras permite expressar o módulo de **a** na forma

$$\mathbf{a} = \sqrt{\mathbf{a}_x^{\,2} + \mathbf{a}_y^{\,2}} \qquad (2.2\text{-}2)$$

A orientação do vetor **a** achamos com

$$tg\theta = \frac{a_x}{a_y} \qquad (2.2\text{-}3)$$

Exemplo:

O vetor **a** = 4 **i** + 3 **j** tem o módulo $a = \sqrt{4^2 + 3^2} = 5$ e o ângulo $\theta = \text{arctg}(\frac{3}{4}) = 36{,}87^o$

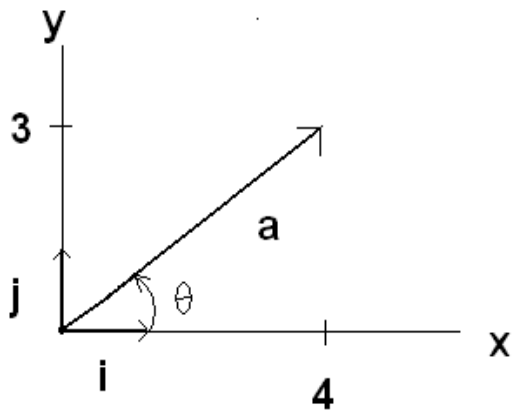

Fig. 2.2-6

Agora mostraremos como podemos fazer essas operações com MuPAD:

- `A:=matrix([[4,3]]):`
- `norm(A,2)// ?norm informação sobre a "2-norm"`

com o resultado 5 para o módulo (que MuPAD chama de `"norm(A,2)"`)

Agora, para o ângulo fazemos o cálculo seguinte:
(Note que `float(arctan(3/4))` calcula o ângulo em rad. A relação entre radiano e graus é estabelecida por 1rad = 180°/π)

- `arctan(3/4)`

`arctan(3/4)`

- `float(%)`

```
0.6435011088
```

- `float(%*180/PI)`

```
36.86989765
```

Para **multiplicar** um vetor **por um número c** basta multiplicar suas componentes por este número:

$$c\mathbf{a} = (ca_x)\mathbf{i} + (ca_y)\mathbf{j}.$$

Para se **somar** dois vetores basta somar suas componentes.

Se $\mathbf{a} = a_x\mathbf{i} + a_y\mathbf{j}$ e $\mathbf{b} = b_x\mathbf{i} + b_y\mathbf{j}$ tem-se $\mathbf{a} + \mathbf{b} = (a_x + b_x)\mathbf{i} + (a_y +)\mathbf{j}$.

O **produto escalar** dos vetores **a** e **b** define-se por

$$\mathbf{a} \cdot \mathbf{b} = a\ b \cos\theta \qquad (2.2\text{-}4)$$

onde θ (theta) ı o βngulo formado pelos dois vetores. Freqóentemente usa-se a letra Φ (phi), -para a matemαtica, não faz diferença.
A partir dessa definição geométrica do produto escalar, não é difícil demonstrar a seguinte relação:

$$\mathbf{a} \cdot \mathbf{b} = a_x\ b_x + a_y\ b_y \qquad (2.2\text{-}5)$$

A equação (2.5) é freqüentemente adotada como definição alternativa de produto escalar.

A **generalização** do que vimos nesta seção para o caso de três dimensões se faz sem dificuldade.
Os seguintes exemplos ilustram a aplicação da função `linalg::scalarProduct(u, v)` de MuPAD. Esta função pertence à biblioteca "linalg" do MuPAD, que contém um grande número de funções da Álgebra linear. Acho que vale a pena dar uma olhada nesta "biblioteca".

- `u := matrix([-2,3]): v := matrix([1.2, -2.5]):`

- `linalg::scalarProduct(u, v)`

```
-9.9
```

Este resultado podemos também obter utilizando as regras do **produto de duas matrizes**.
Mas só podemos efetuar o produto de duas matrizes se o número de colunas da primeira for igual ao número de linhas da segunda. As matrizes u, v possuem uma coluna e três linhas. Então será preciso escrever a matriz u como matriz-linha: (-2, 3), o seja: `matrix([[-2,3]])`
Com esta pequena operação podemos calcular o produto escalar como
(-2)(1.2)+(3)(-2.5) = -9.9

Também MuPAD sabe fazer este cálculo:

- `u := matrix([[-2,3]]): v := matrix([1.2, -2.5]):`
- `p:=u*v;//matriz com um elemento só`
- `p[1,1]// o único elemento da matriz-produto p`

```
[-9.9]
```

```
-9.9
```

Os elementos c_{ik} da **matriz-produto** calculam-se por meio da fórmula $c_{ik} = a_{ir} \cdot b_{rk}$.
c_{ik} é o elemento da i-ésima linha e k-ésima coluna da matriz-produto.
c_{ik} é obtido, multiplicando os elementos da i-ésima linha da primeira matriz pelos elementos correspondentes da k-ésima coluna da segunda matriz, e somando estes produtos.

Observe que o número de colunas da primeira matriz deve ser igual ao número de linhas na segunda.
Se não for assim, não será possível efetuar a multiplicação.

Sejam

$$A_{2x3} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

$$B_{3x2} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix}$$

Então, a matriz-produto **A·B** é a matriz 2 x 2 definida como:

$$A \cdot B = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b2 + a_{23}b_{32} \end{pmatrix} \quad (2.2\text{-}6)$$

Exemplo:

Sejam

$$A = \begin{pmatrix} 3 & 1 \\ 5 & 2 \\ 4 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} -1 & -1 \\ 1 & 4 \end{pmatrix}$$

duas matrizes.
Encontre os produtos **A·B** e **B·A**

- `A:=matrix([[3,1],[5,2],[4,3]]):`
- `B:=matrix([[-1,-1],[1,4]]):`
- `A*B`

$$\begin{pmatrix} -2 & 1 \\ -3 & 3 \\ -1 & 8 \end{pmatrix}$$

- `B*A`

FAIL

Não é possível calcular o produto **B·A**, porque o número de colunas da **B**, da primeira matriz, é diferente do número de linhas da matriz **A**, da segunda.

O **ângulo** entre dois vetores podemos achar usando Eq. (2.2-4), mas o cálculo direto, usando o MuPAD, é bastante complicado. Note que o produto das matrizes **A** e **B** é uma matriz com um elemento só, C[1,1] = 9.

```
A:=matrix([[2,5]]):
B:=matrix([-3,3]):
a:=norm(A,2): //módulo de A
b:=norm(B,2):
C:=A*B:
c:=a*b:
aphi:=arccos(C[1,1]/c):
phi:=float(aphi*180/PI)// ou théta
```

```
66.80140949
```

Utilizemos, agora, a função "`linalg::angle`" da biblioteca "linalg":

```
phi := linalg::angle(
matrix([2, 5]), matrix([-3, 3])):
float(phi*180/PI)
```

```
66.80140949
```

Obtivemos nosso resultado, esta vez, bem mais rápido? Conclusão:

Não há nada pior na vida do que um caminho inadequado.

O produto escalar definido pela Eq. (2.2-4) resulta em uma grandeza escalar, o que justifica o termo *produto escalar*.

É natural a se fazer a pergunta se existe algum produto de vetores que gere uma grandeza de natureza **vetorial**. Já sabemos que na mecânica existe tal produto, o **produto vetorial**. Este produto aparece especialmente com relação a *momentos* e *velocidades angulares*. Nós tratávamos já disso na seção 2.1.

O produto vetorial de **a** por **b**, nessa ordem, é um vetor **c** = **a** x **b** tal que **c** = **0** se, e somente se, **a** e **b** são colineares. (É útil definir um **vetor-zero**, que tem comprimento 0. O vetor-zero (ou vetor-nulo) é paralelo e perpendicular a qualquer vetor **a** .)

Se **a** e **b** não forem colineares, então **c** será tal que o seu comprimento é

$$c = a \cdot b \cdot \text{sen}\ \theta \qquad (2.2\text{-}7)$$

onde theta é o ângulo entre **a** e **b**.
Eq. (2.2-7) é igual à área do paralelogramo formado pelos dois vetores.

O sentido do vetor **c** é definido pela *regra da mão direita*, ou *regra do parafuso*: O vetor **c** tem o sentido do dedo polegar quando a mão direita se posiciona de tal modo que os outros dedos giram de **a** para **b** quando a mão se fecha.

A definição do produto vetorial se baseia em um conceito antrópico. Portanto, pode-se dizer que o produto vetorial é com efeito muito útil, mas que não pertence inteiramente ao domínio da matemática, como sim é o caso com o produto escalar, que não precisa de uma "regra do parafuso" ou de algo equivalente.

Diz-se que o produto vetorial é um *pseudovetor*. De fato, há várias grandezas físicas que são pseudovetores.

Exemplos:

Se **a** = 3 **i** - 4 **j** e **b** = -2 **i** + 3 **k** , qual é o vetor **c** = **a** x **b** ?

- `a := matrix([[3,-4,0]]): b := matrix([[-2, 0, 3]]):`
- `linalg::crossProduct(a, b)`

Resultado: -12 **i** - 9 **j** -8 **k**

Generalização:

Calcule para os vetores **a** = a_x **i** + a_y **j** + a_k **k** e **b** = b_x **i** + b_y **j** + b_z **k** o vetor **c** = **a** x **b**.

a x **b** = (a_x **i** + a_y **j** + a_k **k**) x (b_x **i** + b_y **j** + b_z **k**)

= a_x b_x (**i** x **i**) + a_x b_y (**i** x **j**) + a_x b_z (**i** x **k**)

+ a_y b_x (**j** x **i**) + a_y b_y (**j** x **j**) + a_y b_z (**j** x **k**)

+ a_z b_x (**k** x **i**) + a_z b_y (**k** x **j**) + a_z b_z (**k** x **k**) = (a_y b_z - a_z b_y) **i** + (a_z b_x - a_x b_z) **j** + (a_x b_y - a_y b_x) **k**

Esse resultado pode ser escrito sob a forma de determinantes:

$$\bar{a}\times\bar{b}=\begin{vmatrix}a_y & a_z\\ b_y & b_z\end{vmatrix}\cdot\hat{i}+\begin{vmatrix}a_z & a_x\\ b_z & b_x\end{vmatrix}\cdot\hat{j}+\begin{vmatrix}a_x & a_y\\ b_x & b_y\end{vmatrix}\cdot\hat{k} \qquad (2.2\text{-}8)$$

Esta equação podemos interpretar como sendo a expansão de um determinante:

$$\bar{a}\times\bar{b}=\begin{vmatrix}\hat{i} & \hat{j} & \hat{k}\\ a_x & a_y & a_z\\ b_x & b_y & b_z\end{vmatrix} \qquad (2.2\text{-}9)$$

A verificação destes resultados podemos efetuar utilizando MuPAD:

```
a := matrix([[ax, ay, az]]): b := matrix([[bx, by, bz]]):
linalg::crossProduct(a, b)
```

Resultado:

```
(ay bz - az by, - ax bz + bx az, ax by - ay bx)
```

2.2.4 Vetores e Curvas paramétricas

Se uma curva for dada sob a forma paramétrica

$$x = x(t),\quad y = y(t),\quad a<= t <=b, \qquad (2.2\text{-}10)$$

então o parâmetro t pode ser interpretado como sendo tempo, e o comprimento s da parte de curva entre a e b como sendo a distância percorrida pelo ponto **x**(t) = (x,y) nesse intervalo de tempo.

As quantidades dx/dt e dy/dt podem ser interpretadas como sendo as componentes x e y do vetor-velocidade do ponto móvel (x,y). O vetor-velocidade é tangente à curva definida pelas equações paramétricas (2.10). O comprimento (módulo) do vetor-velocidade é ds/dt:

$$\frac{ds}{dt} = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} \qquad (2.2\text{-}11)$$

Na seguinte figura, o vetor **x**(t+h)-**x**(t) representa uma secante que aproxima-se, quando h tende a 0, do vetor-velocidade **x'**(t).
A **tangente** a uma curva definimos como sendo a posição-limite de uma secante.
O vetor d**x**/dt representa a tangente à curva no ponto **x**(t) .

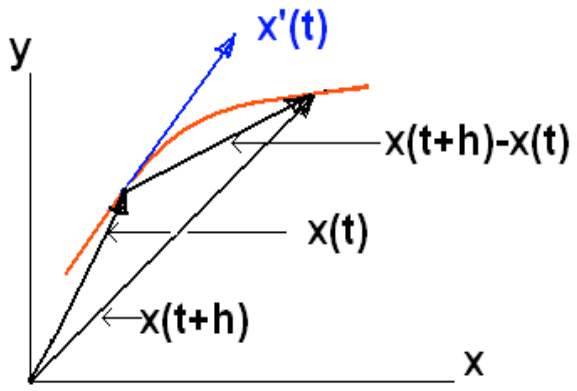

Fig. 2.2-7

Se **x** (t) for o vetor-posição no instante t com as coordenadas (2.2-10), então o vetor-velocidade é definido por o seguinte limite

$$\mathbf{x}'(t) = \lim_{h \to 0} \frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} \qquad (2.2\text{-}12)$$

A quantidade (**x**(t+h)-**x**(t))/h é o vetor **x**(t+h)-**x**(t) vezes o escalar 1/h.
Portanto (**x**(t+h)-**x**(t))/h é um vetor e seu limite é um vetor d**x**/dt; compare com a definição (2.2-12). **x**(t) é uma função vetorial.

Para **derivar** uma função vetorial, derivamos cada componente separadamente.

Um *círculo* com radio r é descrito em forma paramétrica por

$$x(t) = r \operatorname{sen}(t),\ y(t) = r \cos(t) \text{ e } 0 <= t < 2PI \qquad (2.2\text{-}13)$$

Uma *elipse* com semi-eixos a e b é descrita por

$$x(t) = a \cos(t),\ y(t) = b \operatorname{sen}(t) \text{ e } 0 <= t < 2PI \qquad (2.2\text{-}14)$$

Usando MuPAD 3, vamos desenhar um círculo com raio = 2 unidades e um ponto que corresponde ao valor t:= t1= PI/4.
As coordenadas da posição desse ponto para r =2 serão x (PI/4) = y (PI/4) = 1,4142.

Nesse ponto traçamos o vetor-velocidade (= vetor- tangente) e o vetor-aceleração. Com um *fator de escala* "scale" reduzimos o tamanho do vetor com relação à curva. Na seguinte figura vemos os vetores velocidade e aceleração para uma partícula em movimento circular uniforme no sentido horário. Ambos possuem módulos constantes mas variam continuamente em direção.

O vetor aceleração (verde) está dirigido ao longo do raio r, no sentido do centro do círculo.
(Com x(t) = r cos (t), y(t) = r sen (t) resulta um movimento anti-horário).

(É interessante "plotar" outras curvas, por exemplo com x (t) = t^2 e y (t) = t + cos(3 t); 0 <= t <= PI.)

Agora desenvolvemos uns programas para representar os vetores **velocidade** e **aceleração** para uma partícula em movimento circular uniforme, utilizando MuPAD 3, que também permite criar representações **animadas**.

Primeiramente traçamos os vetores **v** e **a** para um ponto fixo no "instante" t1=PI/4.

Programa 1

```
x:=t->2*sin(t):

y:=t->2*cos(t):

curve:=plot::Curve2d([x(t),y(t)],t= 0..2*PI):

pos:=t->(x(t),y(t)):vetor posição

vel:=t->(x'(t),y'(t)):vetor velocidade

acel:=t->(x''(t),y''(t)):aceleração

t1:=PI/4:instante elegido

scale:=0.6:determina o tamanho do vetor

x1:=pos(t1)[1]:início do vetor (x'(t),y'(t))

y1:=pos(t1)[2]:para t=t1

v1:=vel(t1)[1]:velocidade do ponto (x(t),y(t))
```

```
v2:=vel(t1)[2]://para t=t1

a1:=acel(t1)[1]://aceleração

a2:=acel(t1)[2]:

x2:=x1+v1*scale://extremo do vetor (x'(t),y'(t))

y2:=y1+v2*scale:

x3:=x1+a1*scale://extremo do vetor (x''(t),y''(t))

y3:=y1+a2*scale:

p:=plot::Point2d(x1,y1,Color=RGB::Green,PointSize=3*unit::m
m):

ve:=plot::Arrow2d([x1,y1],[x2,y2],Color=RGB::Red):

ac:=plot::Arrow2d([x1,y1],[x3,y3],Color=RGB::Green):

plot(curve,p,ac,ve,Scaling = Constrained)
```

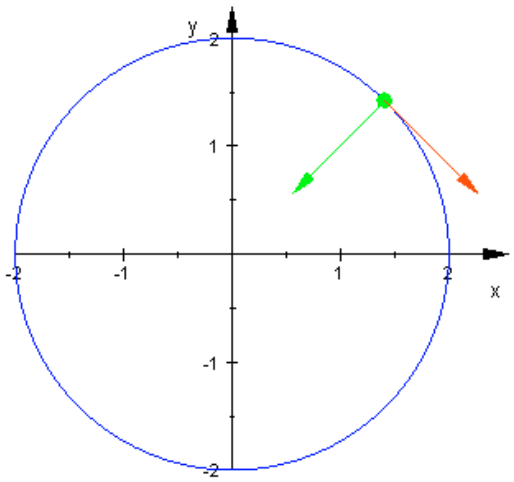

Fig. 2.2-8

Com a instrução "`Scaling = Constrained`" vetores ortogonais também aparecerão sendo ortogonais na tela e um círculo será um círculo.

(MuPAD 3 criará um plot de tal forma que ele caiba no "plot window", ajustando as dimensões horizontais em forma independente das dimensões verticais. Muitas vezes, não queremos isso, mas com `Scaling = Constrained` é fácil de evitar as distorções resultantes. No entanto, compare o Programa 4, onde não utilizamos este comando.)

O **valor absoluto** de velocidade e aceleração traçadas com "plot::Function2d". O gráfico mostra que o módulo de velocidade e aceleração, a = v^2/r, são constantes. Calculamos o módulo por meio dos produtos escalar **v**·**v** e **a**·**a**.

Programa 2

```
x:=t->2*sin(2*t):

y:=t->2*cos(2*t):

v:=matrix([[x'(t),y'(t)]])://vector-velocidade

a:=matrix([[x''(t),y''(t)]])://vector-aceleração

v0:=sqrt(linalg::scalarProduct(v, v))://val.abs da
veloidade

a0:=sqrt(linalg::scalarProduct(a, a))://val.abs da
aceleração

v1:=plot::Function2d(v0(t),t=0..2*PI, Color=RGB::Red):

a1:=plot::Function2d(a0(t),t=0..2*PI,Color=RGB::Green):

plot(v1,a1)
```

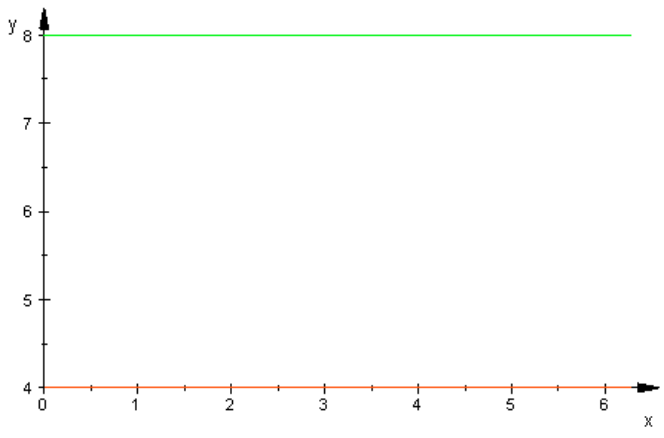

Fig. 2.2-9

Animação

Agora vamos *animar* o gráfico dos vetores de velocidade e aceleração do Programa 1. (Somente a partir da Versão 3 pode-se fazer gráficos animados. Note que só é necessário introduzir o domínio do tempo "`t = 0..2*PI`" nas três instruções "`plot`". Clique duas vezes, com o botão esquerdo do mouse, sobre a figura criada. Feito isto, entre no menu *Animation*. Na janela que surgir, clique em *Start*.)

Programa 3

```
x:=t->2*sin(t):

y:=t->2*cos(t)://Elipse: y:=t->3*cos(t):

curve:=plot::Curve2d([x(t),y(t)],t= 0..2*PI):

pos:=t->(x(t),y(t)):vetor posição

vel:=t->(x'(t),y'(t)):vetor velocidade

acel:=t->(x''(t),y''(t)):aceleração

scale:=0.6:determina o tamanho do vetor

x1:=pos(t1)[1]:início do vetor (x'(t),y'(t))

y1:=pos(t1)[2]:

v1:=vel(t1)[1]:velocidade do ponto (x(t),y(t))

v2:=vel(t1)[2]:

a1:=acel(t1)[1]:aceleração

a2:=acel(t1)[2]:

x2:=x1+v1*scale:extremo do vetor (x'(t),y'(t))

y2:=y1+v2*scale:

x3:=x1+a1*scale:extremo do vetor (x''(t),y''(t))

y3:=y1+a2*scale:

p:=plot::Point2d([x1,y1],t1=0..2*PI,Color=RGB::Black,
PointSize=3*unit::mm):

ve:=plot::Arrow2d([x1,y1],[x2,y2],t1=0..2*PI,Color=RGB::Red
):
```

```
ac:=plot::Arrow2d([x1,y1],[x3,y3],t1=0..2*PI,Color=RGB::Gre
en):

plot(curve,p,ac,ve,Scaling = Constrained)
```

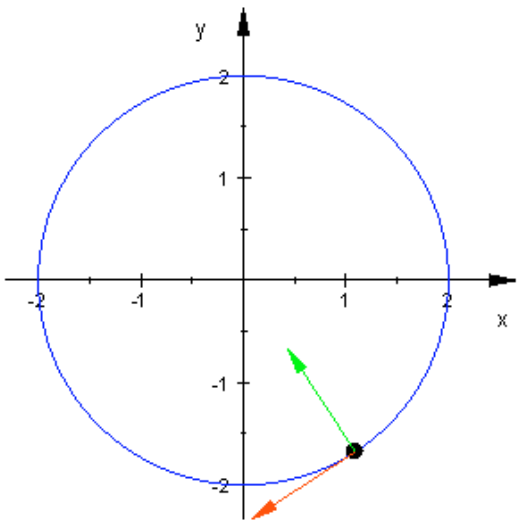

Fig. 2.2-10

Agora, você deveria fazer um estudo do movimento de uma partícula ao longo de uma **trajetória elíptica** em um plano xy horizontal. Você verá que o vetor-velocidade não fica perpendicular ao vetor-aceleraçaõ. Os módulos dos vetores também mudam com a posição do ponto. Mas o vetor-aceleraçaõ permanecerá mostrando sempre para o centro.
O "plot" do módulo dos vetores já não mostra linhas horizontais, e sim variações entre um máximo e um mínimo. Os módulos de **v** e **a** são, agora, funções oscilatórias. Tudo isso pode ser verificado muito facilmente modificando um pouco os programas anteriores.

Finalmente, consideraremos a questão de como representar **o movimento de um projétil** (no vácuo). Neste caso, a aceleração é a aceleração constante da gravidade: **a = g**. O projétil não possui aceleração horizontal e a nossa animação do movimento vai mostrar que o vetor aceleração é sempre dirigido verticalmente para baixo. Quando a aceleração é constante, a trajetória é uma parábola e, como veremos no proximo capítulo (2.3.1), as coordenadas são dadas por as seguintes equações paramétricas:

$$x(t) = v_0\, t \cos(\alpha) \qquad (2.2\text{-}15a)$$

$$y(t) = -g\, t^2/2 + v_0\, t \operatorname{sen}(\alpha) \qquad (2.2\text{-}15b)$$

O projétil é lançado com uma velocidade inicial de módulo v_0 = 70 m/s a um ângulo de 30^0 (= ângulo de tiro) com a horizontal. É só modificar um pouco o nosso Programa 1, para mostrar a trajetória que o projétil percorre, junto com os vetores **v** e **a**. Pode-se calcular que o móvil retorna ao solo (altura de lançamento) depois de 7,136 s. Olhe no vetor-velocidade, e observe que ele varia continuamente!

(Sua componente horizontal permanece constante mas a componente vertical varia ao longo da trajetória e sendo nula no ponto mais alto.)
Insira as seguintes linhas para ver também as **componentes** de **v**:

```
vx:=plot::Arrow2d([x1,y1],[x2,y1],t1=0..7.13,
Color=RGB::Black):
vy:=plot::Arrow2d([x1,y1],[x1,y2],t1=0..7.13,
Color=RGB::Blue):
plot(curve,p,ac,ve,vx,vy)
```

Note que no programa tiramos o comando "`Scaling = Constrained`". Porquê?

Programa 4 (projétil):

```
v0:=70:
g:=9.81:
alpha:=PI/6:
x:=t->v0*t*cos(alpha):
y:=t->-g/2*t^2+v0*t*sin(alpha):
curve:=plot::Curve2d([x(t),y(t)],t= 0..7.13):
pos:=t->(x(t),y(t)):vetor posição
vel:=t->(x'(t),y'(t)):vetor velocidade
acel:=t->(x''(t),y''(t)):aceleração
scale:=0.6:determina o tamanho do vetor
x1:=pos(t1)[1]:início do vetor (x'(t),y'(t))
```

```
y1:=pos(t1)[2]://para t=t1

v1:=vel(t1)[1]://velocidade do ponto (x(t),y(t))

v2:=vel(t1)[2]://para t=t1

a1:=acel(t1)[1]://aceleração

a2:=acel(t1)[2]:

x2:=x1+v1*scale://extremo do vetor (x'(t),y'(t))

y2:=y1+v2*scale:

x3:=x1+a1*3*scale:// o fator escala foi aumentado para
alongar

y3:=y1+a2*3*scale:// o vetor-aceleração

p:=plot::Point2d([x1,y1],t1=0..7.13,Color=RGB::Black,PointS
ize=3*unit::mm):

ve:=plot::Arrow2d([x1,y1],[x2,y2],t1=0..7.13,Color=RGB::Red
):

ac:=plot::Arrow2d([x1,y1],[x3,y3],t1=0..7.13,Color=RGB::Gre
en):

plot(curve,p,ac,ve)
```

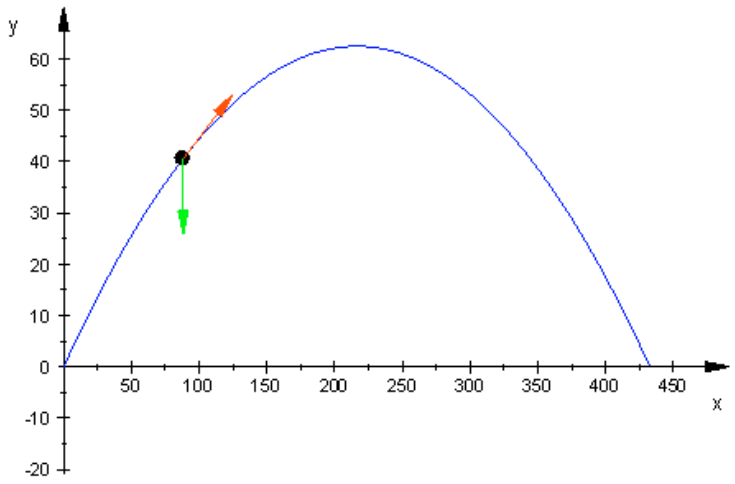

Fig. 2.2-11

Para acharmos o **tempo do vôo** (tempo decorrido até o projétil atingir o solo), pomos y :=0.

- ```
  v0:=70:g:=9.81:
  alpha:=PI/6:
  x:=t->v0*t*cos(alpha):
  y:=t->-g/2*t^2+v0*t*sin(alpha):
  solve(y(t)=0,t)[2]
  ```

```
7.135575943
```

t_v = 7.1356 s é o *segundo* elemento da lista das soluções.

Também pode-se calcular o tempo total, como veremos um pouco mais adiante, por meio da fórmula

```
2*v0*sin(alpha)/g
```

```
7.135575943
```

O **tempo de subida** equivale ao intervalo de tempo decorrido desde o instante do lançamento até o instante em que o móvel atinge o vértice da parábola. Neste instante, a componente vertical da velocidade é nula. Então podemos escrever:

```
vy:=y'(t):
solve(vy=0,t)
```

```
{3.567787971}
```

O tempo de descida é igual ao de subida.

A **altura máxima** de 62,44 m é obtida por meio de

```
solve(vy=0,t):
y(solve(vy=0,t)[1])
```

```
62.4362895
```

Para o **alcance máximo** obtemos 432,57 m.
```

Estes valores podem-se controlar inspecionando diretamente a curva Fig. 2.2-11.

(Para isso, clique com o botão esquerdo do mouse sobre a curva e oprima, ao mesmo tempo, a tecla "Ctrl." Vai surgir uma caixinha com as coordenadas do ponto.)

Agora faremos uma **tabela** das coordenadas da trajetória do projétil.
Primeiramente usamos o comando "`output::tableForm`" :

Tabela de valores de t, x, y usando "`output::tableForm`":

- ```
  v0:=70:g:=9.81:alpha:=PI/6:
  DIGITS :=6:
  x:=t->v0*t*cos(alpha):
  y:=t->-g*t^2/2+v0*t*sin(alpha):
  output::tableForm([[t $ t = 1..5,"t"]]):
  output::tableForm([[float(x(t)) $ t = 1..5,"x"]]):
  output::tableForm([[float(y(t)) $ t = 1..5,"y"]]):
  ```

```
[1, 2, 3, 4, 5, "t"]
[60.6218, 121.244, 181.865, 242.487, 303.109, "x"]
[30.095, 50.38, 60.855, 61.52, 52.375, 33.42, "y"]
```

É possível simplificar o programa:

- ```
  v0:=70:g:=9.81:alpha:=PI/6:
  DIGITS :=6:
  x:=t->v0*t*cos(alpha):
  y:=t->-g*t^2/2+v0*t*sin(alpha):
  output::tableForm([[T,X,Y] $ t=0..0],Column=1,Unique):
  output::tableForm([[t,float(x(t)),float(y(t))] $ t=0..5]
  ,Column=1, Unique)
  ```

```
[T, X, Y]

[0, 0.0, 0.0]

[1, 60.6218, 30.095]

[2, 121.244, 50.38]

[3, 181.865, 60.855]

[4, 242.487, 61.52]

[5, 303.109, 52.375]
```

O método mais simples para criar uma tabela de valores computados é usar uma matriz (agradeço a Wolfgang Lindner por indicar esse caminho!):

```
v0:=70:g:=9.81:alpha:=PI/6:

DIGITS :=6:

x:=t->v0*t*cos(alpha):

y:=t->-g*t^2/2+v0*t*sin(alpha):

matrix([[T,X,Y],[t,float(x(t)),float(y(t))] $ t=0..8])
```

T	X	Y
0	0	0
1	60.6218	30.095
2	121.244	50.38
3	181.865	60.855
4	242.487	61.52
5	303.109	52.375
6	363.731	33.42
7	424.352	4.655
8	484.974	−33.92

Não podemos deixar de ficar impressionados com um método tão simples!

Claro, poderíamos ser mais exigentes e escrever um programa na linguagem de programação do MuPAD. O procedimento ("procedure") consiste em encher um "array" dos valores de *t, x* e *y*.
O seguinte "procedure" faz isso *passos*-vezes. Para deixar o número de operações aberto, utilizamos a variável *passos*. Tudo funciona como em *Basic* ou *Pascal*.

(Usamos, agora, as ferramentas da computação sem grandes explicações.
As noções *array* e *procedure* serão tema de outras seções. Se você sabe programar em *Pascal*, então não vai ter problemas com o seguinte programa escrito na linguagem de programação do MuPAD.)

O nome do procedimento é "Projetil". Para arrancar este "procedure" temos de dar-lhe o valor inicial do tempo, t0, e os números de passos a computar: `proc(t0,passos)`. Na última linha do programa você vê a instrução: `Projetil(0,8)` onde entregamos t0 = 0 e passos = 8.

Programa 5

Tabela como procedimento (procedure):

```
Projetil:=proc(t0,passos)
local t,x,y,tabela;
begin
v0:=70:g:=9.81:alpha:=PI/6:
DIGITS:=6:
t:=t0:
tabela:=array(0..passos, 1..3):
for t from t0 to passos do
x:=float(v0*t*cos(alpha)):
y:=float(-g*t^2/2+v0*t*sin(alpha)):
tabela[t,1]:=t:
tabela[t,2]:=x:
tabela[t,3]:=y:
end_for:
return(tabela)
end_proc:
Projetil(0,8)
```

0	0.0	0.0
1	60.6218	30.095
2	121.244	50.38
3	181.865	60.855
4	242.487	61.52
5	303.109	52.375
6	363.731	33.42
7	424.352	4.655
8	484.974	−33.92

A primeira coluna contém o tempo, a segunda as coordenadas x, e a terceira as y. Depois de 8 segundos o corpo já penetrou no solo a uma profundidade de -33.92 m! Na próxima versão do programa incluímos também um título para as colunas (note o uso das instruções "`if ... then... else`" !).

Programa 6

```
Projetil:=proc(t0,passos)
local t,x,y,tabela;
begin
v0:=70:g:=9.81:alpha:=PI/6:
DIGITS:=6:
t:=t0:
tabela:=array(0..passos, 1..3):
for t from t0 to passos do
x:=float(v0*t*cos(alpha)):
y:=float(-g*t^2/2+v0*t*sin(alpha)):
if t = t0 then
tabela[t,1]:=T/s:
tabela[t,2]:=X/m:
tabela[t,3]:=Y/m:
else
```

```
    tabela[t,1]:=t:
    tabela[t,2]:=x:
    tabela[t,3]:=y:
    end_if: end_for:
    return(tabela)
    end_proc:
    Projetil(0,8)
```

t / s	x / m	y / m
1	60.6218	30.095
2	121.244	50.38
3	181.865	60.855
4	242.487	61.52
5	303.109	52.375
6	363.731	33.42
7	424.352	4.655
8	484.974	−33.92

Comprimento da trajetória (comprimento de arco)

Agora gostaríamos de computar o comprimento da trajetória por meio da Eq. (2.11). Não existindo solução analítica para esse problema, temos de usar o comando "`numeric::int` '" para obter a solução numérica de 455,525 m:

Programa 7

```
v0:=70:
g:=9.81:
alpha:=PI/6:
x:=t->v0*t*cos(alpha):
y:=t->-g/2*t^2+v0*t*sin(alpha):
numeric::int(sqrt(x'(t)^2+y'(t)^2), t = 0..7.13558)
```

455.5252667

Explicação:

Da propriedade (2.2-11) obtemos

$$ds = \sqrt{x'(t)^2 + y'(t)^2} \cdot dt$$

e daí segue para a distância percorrida desde o ponto inicial (t = a) até o ponto final (t = b)

$$s = \int_a^b \sqrt{x'(t)^2 + y'(t)^2} \cdot dt \qquad (2.2\text{-}16)$$

Esta integral definida é aplicável a uma curva dada sob forma paramétrica x = x(t), y = y(t).
Trata-se de um caso especial de uma integral curvilínea.