# "fiziko" v. 0.1.3 package for MetaPost

Sergey Slyusarev

March 8, 2019

**Abstract**

This document describes a bunch of macros provided by "fiziko" library for MetaPost.

## 1 Introduction

This MetaPost library was initially written to automate some elements of black and white illustrations for a physics textbook. First and foremost it provides functions to draw things like lines of variable width, shaded spheres and tubes of different kinds, which can be used to produce images of a variety of objects. The library also contains functions to draw some objects constructed from these primitives.

## 2 Usage

Simply include this in the beginning of your MetaPost document:

```
1    input fiziko.mp
```

## 3 Global variables

A few global variables control different aspects of the behavior of the provided macros. Not all possible values are meaningful and some will definitely result in ugly pictures or errors.

### 3.1 minStrokeWidth

This variable controls minimal thickness of lines that are used for shading. Below this value lines are not getting thinner, but become dashed instead, maintaining roughly the same amount of ink per unit length as a thinner line would take.

Default value is one fifth of a point. There are several things that depend on this value, so it's convenient to change it using a macro:

```
1    defineMinStrokeWidth(1/2pt);
```

### 3.2   lightDirection

This variable controls direction from which light falls on shaded objects. It's of `pair` type and is set in radians. Default direction is top-left:

```
1    lightDirection := (−1/8pi, 1/8pi);
```

## 4   "Lower level" macros

Currently, algorithms are quite stupid and will produce decent results only in certain simple circumstances.

### 4.1   offsetPath (*path*)(*offset function*)

This macro returns offset path (of type `path`) to a current path with a distance from the original path controlled by some arbitrary function; typically, it is a function of path length, set as either `offsetPathTime` or `offsetPathLength`. Former is simply `time` on current path and changes from 0 to `length(path)`, and latter changes from 0 to 1 over the path `arctime` (as a function of `arclength`).

```
1    path p, q;
2    p := (0,0){dir(30)}..(5cm, 0)..{dir(30)}(10cm, 0);
3    q := offsetPath(p)(1cm*sin(offsetPathLength*pi));
4    draw p;
5    draw q dashed evenly;
```
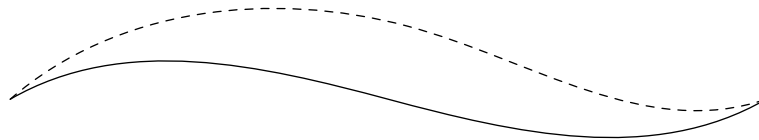


### 4.2   brush (*path*)(*offset function*)

This macro returns a `picture` of a line of variable width along given path, which is controlled by some arbitrary function, analogous to `offsetPath`. If line is getting thinner than `minStrokeWith`, it is drawn dashed.

```
1    path p;
2    p := (0,0){dir(30)}..(5cm, 0)..{dir(30)}(10cm, 0);
3    draw brush (p)(2minStrokeWidth*sin(offsetPathLength*pi));
```

## 4.3   sphere.c (*diameter*)

This macro returns a `picture` of a sphere with specified diameter shaded with concentric strokes. Strokes are arranged to fit those of `tube.l`.

```
1      for i := 1 step 1 until 6:
2          draw sphere.c(i*1/4cm) shifted (1/2cm*(i*(i+1)))/2, 0);
3      endfor;
```

## 4.4   sphere.l (*diameter, angle*)

This macro returns a `picture` of a shaded sphere with specified diameter. Unlike `sphere.c` macro, this one draws latitudinal strokes around axis rotated at specified `angle`.

```
1      for i := 1 step 1 until 6:
2          draw sphere.l(i*1/4cm, -90 + i*30)
3              shifted (1/2cm*(i*(i+1)))/2, 0);
4      endfor;
```

## 4.5   tube.l (*path*)(*offset function*)

This macro returns a `picture` of a shaded "tube" of a variable width along a given path, which is controlled by some arbitrary function, analogous to `offsetPath`. "Tube" drawn by this macro is shaded be longitudal strokes. Once tube is generated, you can call `tubeOutline` path global variable, if you need one.

```
1      path p;
2      p := (0,0){dir(30)}..(5cm, 0)..{dir(30)}(10cm, 0);
3      draw tube.l (p)(1/2cm*sin(offsetPathLength*pi));
```
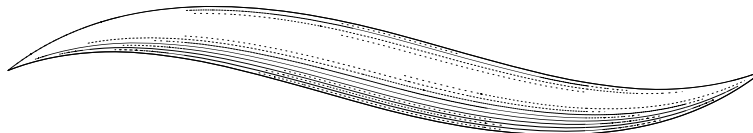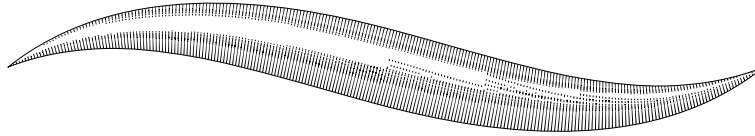
## 4.6   tube.t (*path*)(*offset function*)

This macro returns a `picture` of a shaded "tube" of variable width along given path, which is controlled by some arbitrary function, analogous to `offsetPath`. "Tube" drawn by this macro is shaded be transverse strokes. Once tube is generated, you can call `tubeOutline` path global variable, if you need one.

```
1      path p;
2      p := (0,0){dir(30)}..(5cm, 0)..{dir(30)}(10cm, 0);
3      draw tube.t (p)(1/2cm*sin(offsetPathLength*pi));
```

## 4.7   tube.e (*path*)(*offset function*)

This macro returns the outline of a tube as a path.

# 5   "Higher level" macros

Using macros described in the previous section it is possible to construct more complex images. Macros for drawing some often used images are present in this package.

## 5.1   eye (*angle*)

This macro returns a `picture` of an eye pointed at the direction `angle` (in degrees). Eye size is controlled by a global variable `eyescale`, which has default value of `eyescale := 1/2cm;`.

```
1      save eyescale;
2      for i := 1 step 1 until 6:
3          eyescale := 1/6cm*i;
4          draw eye(i*60) shifted (1/2cm*(i*(i+1))/2, 0);
5      endfor;
```
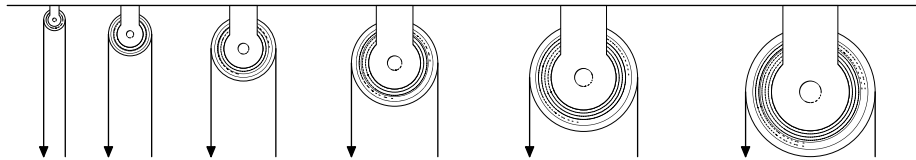
## 5.2   pulley (*diameter, angle*)

This macro returns a `picture` of a pulley with specified `diameter` and its support pointed at the direction `angle` (in degrees). Note that pulley's support protrudes from its center by `pulleySupportSize*diameter` and by default $\text{pulleySupportSize} = 3/2$. Once pulley is generated, you can call `pulleyOutline` path global variable, if you need one.

```
1      draw (−1/8cm, 0)−−(12cm, 0);
2      for i := 1 step 1 until 6:
3          r := 1/7cm*i;
4          draw image(
5              draw pulley(2r, 0) shifted (0, −4/3r);
6              draw (r, −4/3r) −− (r, −2cm);
7              drawarrow (−r, −4/3r) −− (−r, −2cm);
8          ) shifted (1/2cm*(i*(i+1))/2, 0);
9      endfor;
```
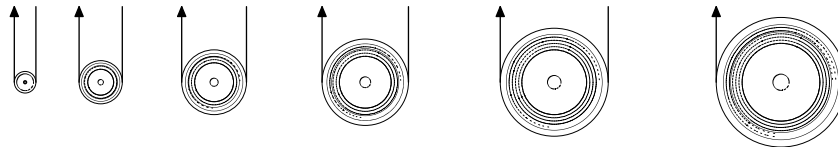
4

## 5.3 pulleyWheel (*diameter*)

This macro returns a `picture` of a pulley wheel with specified `diameter`.

```
1    for i := 1 step 1 until 6:
2        r := 1/7cm*i;
3        draw image(
4            draw pulleyWheel(2r);
5            draw (r, 0) -- (r, 1cm);
6            drawarrow (-r, 0) -- (-r, 1cm);
7        ) shifted (1/2cm*(i*(i+1))/2, 0);
8    endfor;
```
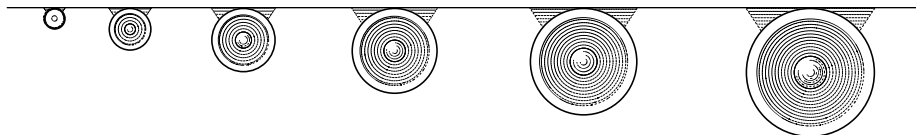


## 5.4 wheel (*diameter, angle*)

This macro returns a `picture` of a wheel with specified `diameter` and its support
pointed at the direction `angle` (in degrees).

```
1    draw (-1/8cm, 0)--(12cm, 0);
2    for i := 1 step 1 until 6:
3        r := 1/7cm*i;
4        draw wheel(2r, 0) shifted (0, -r) shifted (1/2cm*(i*(i+1))/2, 0);
5    endfor;
```
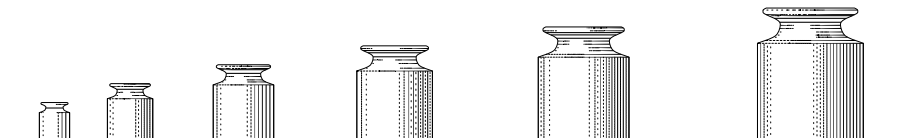


## 5.5 weight.s (*height*)

This macro returns a `picture` of a weight of a specific `height` that is standing
on the point `(0, 0)`.

```
1    for i := 1 step 1 until 6:
2        draw weight.s(1/4cm + i*1/4cm) shifted (1/2cm*(i*(i+1))/2, 0);
3    endfor;
4    draw (-1/8cm, 0)--(12cm, 0);
```
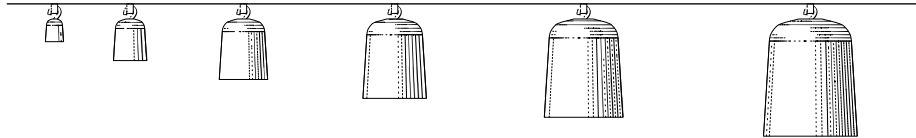
## 5.6   weight.h (*height*)

This macro returns a `picture` of a weight of a specific `height` that is is hanging from the point `(0, 0)`.

```
1       for  i  :=  1  step  1  until  6:
2           draw  weight.h(1/4cm + i*1/4cm)  shifted  (1/2cm*(i*(i+1))/2,  0);
3       endfor;
4       draw  (12cm,  0)−−(−1/8cm,  0);
```



## 5.7   spring (*point a, point b, number of steps*)

This macro returns a `picture` of a spring stretched between points `a` and `b` (of type `pair`), with specified `number of steps`. Spring width is controlled by global variable `springwidth` with the default value `springwidth := 1/8cm;`.
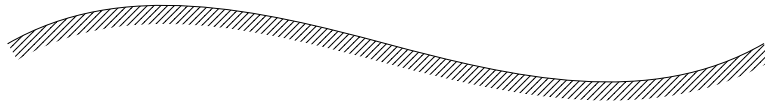
```
1       pair a,  b;
2       a  :=  (0,  0);
3       for  i  :=  1  step  1  until  6:
4           springwidth  :=  1/16cm + i*1/48cm;
5           b  :=  (i*1/3cm,  − i*1/5cm);
6           draw  spring  (a,  b,  10)  shifted  (2/5cm*(i*(i+1))/2,  0);
7       endfor;
```
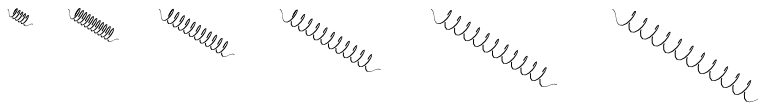


## 5.8   solidSurface (*path*)

This macro returns a `picture` of a solid surface on the right side of a given `path`.

```
1       path p;
2       p  :=  (0,0){dir(30)}..(5cm,  0)..{dir(30)}(10cm,  0);
3       draw  solidSurface(p);
```



## 5.9   solid (*path, angle, type*)

Fills given `path` with strokes of specific type at a given `angle`. `type` can be 0 ("solid" strokes) and 1 ("glass" strokes).

## 5.10 woodBlock (*width, height*)

Returns a `picture` of a rectangular block of wood with its bottom-left corner in the origin.
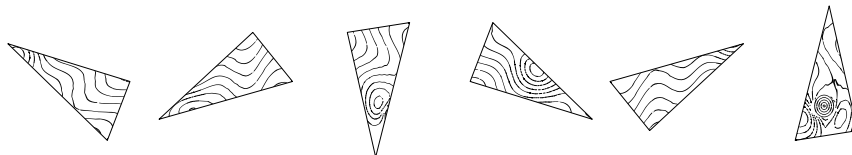
```
1      draw woodBlock(10cm, 1/2cm);
```



## 5.11 woodenThing (*path, angle*)

Returns a `picture` of a wood texture at a given `angle` fitted into a given `path`.

```
1      path p, q;
2      p := dir(−60) scaled 1/2 −− dir(90) scaled 2/3 −− dir (−120) scaled 3/5 −− cycle;
3      for i := 1 step 1 until 6:
4          q := (p scaled 3/2cm) rotated (i*60);
5          draw woodenThing (q, i*60) shifted (2cm*i, 0);
6      endfor;
```



## 5.12 globe (*radius, longitude, latitude*)

This macro returns a `picture` of the globe of specified `radius` centered at specific `longitude` and `latitude`;

```
1      for i := 1 step 1 until 6:
2          draw globe(i*1/4cm, i*60, −90 + i*30)
3              shifted (1/2cm*(i*(i+1))/2, 0);
4      endfor;
```
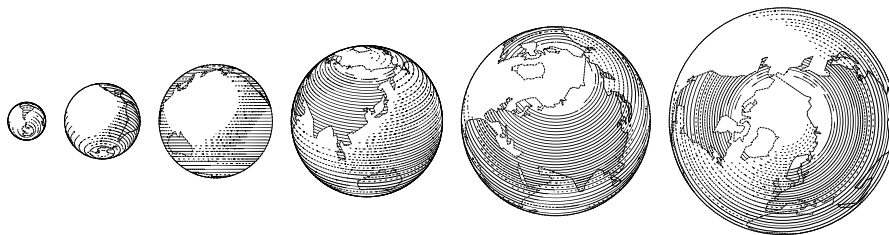


## 5.13 Knots

There are two macros to handle knot drawing: `addStrandToKnot` and `knotFromStrands`. Currently the algorithm is not especially stable.

### 5.13.1  addStrandToKnot (*knotName*) (*path, ropeWidth, ropeType, intersectionOrder*)

This macro adds a strand to knot named `knotName` and returns nothing. Strand follows the given `path` and has a given `ropeWidth`. `ropeType` can be `"l"`, `"t"` (as in `tube.l` and `tube.t`) or `"e"` (for an unshaded strand). `intersectionOrder` is a string of comma separated numbers which represent a "layer" to which intersections along the strand go.

### 5.13.2  knotFromStrands (*knotName*)

This macro returns a picture of a knot with a given `knotName`.

```
1      path p[];
2      p1 := (dir(90)*4/3cm) {dir(0)} .. tension 3/2
3          .. (dir(90 + 120)*4/3cm){dir(90 + 30)} .. tension 3/2
4          .. (dir(90 − 120)*4/3cm){dir(−90 − 30)} .. tension 3/2
5          .. cycle;
6      p2 := (fullcircle scaled 3cm) shifted (0, −3/2cm);
7      p3 := (fullcircle scaled 4cm);
8      addStrandToKnot (theknot) (p1 shifted (4cm, −4cm), 1/5cm, "l",
9          "−1,1,−1,1,−1,1,−1,1,−1");
10     addStrandToKnot (theknot) (p2 shifted (4cm, −4cm), 1/6cm, "l",
11         "");
12     addStrandToKnot (theknot) (p3 shifted (4cm, −4cm), 1/7cm, "l",
13         "−1,1");
14     draw knotFromStrands (theknot);
```
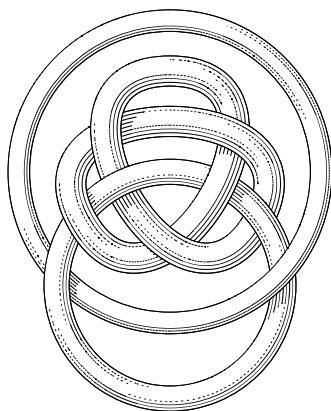


# 6  Other macros

Some macros that are not directly related to drawing are listed below

## 6.1  refractionPath (*initial ray, shape, refraction coefficient*)

This macro returns a `path` that represent refraction of some `ray` (any variable of type `path`, point next to last in a given path is considered a source of a "ray", and

last point determines its direction) through some `shape` with given `refraction coefficient`. When appropriate, the "ray" is fully internally reflected.

Setting `refraction coefficient` to 0 results in reflection instead of refraction in all cases.

```
1    path r, p;
2    p := fullcircle scaled 2.1cm;
3    draw p;
4    for i:= 1cm step -1/4cm until -1cm:
5        r := (-4cm, i) -- (-1cm, i);
6        draw refractionPath(r, p, 1.5);
7    endfor;
```
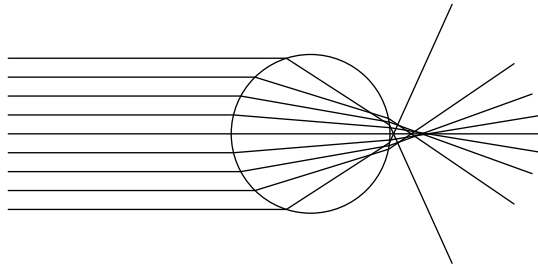
## 6.2   lens ((*left radius, right radius*), *thickness, diameter, units*)

This macro returns a `path` that represent a section of a lens with given radii of curvature (positive value for convex, negative — for concave), thickness (i. e. distance benween sides' centers) and diameter (i. e. height) in given arbitrary units.

```
1    draw lens((5, 10), 1/2, 2, cm);
2    draw lens((-10, -5), 1/4, 2, cm) shifted (2cm, 0);
3    draw lens((infinity, 7), 1/4, 2, cm) shifted (4cm, 0);
```
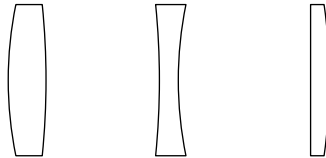
# 7   Auxilary macros

Some macros that not related to physical problems at all are listed below.

## 7.1   *picture* maskedWith *path*

This macro masks a part of a `picture` with closed `path`. In fact this is inversion of metapost's built-in `clip` but, in contrast to the latter, it does not modify original image. Note that it requires that counter-clockwise `path` to work properly.

## 7.2 *path* firstIntersectionTimes *path*

This macro is similar to metapost's `intersectiontimes` but it returns intersection times with smallest time on first path.

## 7.3 pathSubdivide *path, n*

This macro returns original `path` with **n**-times more points.

## 7.4 drawmidarrow (*path*)

Draws `path` with arrows in the middles of segments with length no less than `midArrowLimit` (another global variable, 1cm by default).

## 7.5 markAngle (*point a, point o, point b*)(*text*)

This macro marks an angle `aob` (counter-clockwise) with some `text`

```
1       pair a, o, b;
2       for i:= 30 step 60 until 390:
3           o := (10cm*(i/360), 0);
4           a := dir(i/2)*4/3cm shifted o;
5           b := dir(i)*4/3cm shifted o;
6           draw (a—o—b);
7           markAngle(a, o, b)(btex $\alpha$ etex);
8       endfor;
```
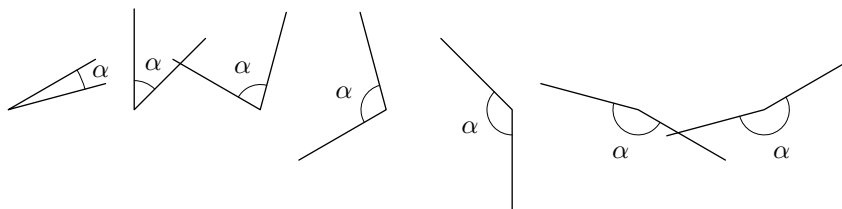


# 8 Some examples

## 8.1 Gregory-Maksutov type telescope

Lines 3–11 define parameters of lenses and mirrors[1]. Lines 12–16 generate lenses. On line 20 shape of prism is defined. Line 22 cuts part of the rear mirror. Line 26 describes mirror part of the front lens. On lines 31–33 all the glass parts are drawn. On lines 34–36 all the mirror ones. On lines 39–44 rays are traced through all system in order specified in loop on line 41. Lines 49–56 are about the telescope frame.

```
1       path p[], q[], axis[], f[]; pair o;
2       u := 6mm;
3       r1 := −36.979; r2 := −r1; t1 := 0.7; n1 := 1.517; d1 := 5;
4       l1 := 8.182;
5       r3 := −11.657; r4 := r3; t2 := 0.2; n2 := n1; d2 := 3/2;
```

---

[1]Taken from here http://www.google.ru/patents/US2701983

```
6        l2 := 0.4;
7        r5 := −30.433; r6 := 9.598; t3 := 0.39; n3 := 1.621; d3 := d2;
8        l3 := 0.828;
9        r7 := −35.512; r8 := infinity; t4 := 0.7; n4 := 0; d4 := d1;
10       l4 := 5.272;
11       ll0 := 0;
12       for i := 1 upto 4:
13           ll[i] := ll[i−1] + t[i] + l[i];
14           p[i] := lens ((r[i*2 − 1], r[i*2]), t[i], d[i], u)
15                   shifted (ll[i−1]*u, 0);
16       endfor;
17       axis1 := (0, 0) −− (ll4*u, 0);
18       axis2 := reverse(axis1);
19       ll5 := ll4 − 1/2l4;
20       p5 :=  ((−1,−1) −− (1,1) −− (−1,1) −− cycle)
21           scaled (1/2d2*u) shifted (ll5*u, 0);
22       p6 := (subpath
23           (ypart((axis1 shifted (0, 1/2d2*u)) firstIntersectionTimes p4),
24           ypart((axis2 shifted (0, 1/2d2*u)) firstIntersectionTimes p4))
25           of p4) −− cycle;
26       p7 := (subpath
27           (ypart((axis2 shifted (0, 3/4d2*u)) firstIntersectionTimes p1),
28           ypart((axis2 shifted (0, −3/4d2*u)) firstIntersectionTimes p1))
29           of p1);
30       p7 := p7 −− (reverse(p7) shifted ((−1/3t1)*u, 0)) −− cycle;
31       for i := 1, 2, 3, 5:
32           draw p[i] withpen thickpen; draw solid (p[i], 45, 1);
33       endfor;
34       draw solid (p7, −45, 0);
35       draw p6 withpen thickpen; draw solid (p6, −45, 0);
36       draw p6 yscaled −1 withpen thickpen;
37       draw solid (p6 yscaled −1, −45, 0);
38       n7 := 0; n5 := n1;
39       for i := 0, 1:
40           q[i] := (−3/2u, −2u + i*4u) −− (16u, −2u + i*4u);
41           for j = 1, 4, 7, 2, 3, 5:
42               q[i] := refractionPath(q[i], p[j], n[j]);
43           endfor;
44       endfor;
45           o := whatever[point length(q0) of q0, point length(q0)−1 of q0]
46               = whatever[point length(q1) of q1, point length(q1)−1 of q1];
47       for i := 0, 1: drawmidarrow (q[i] −− o) withpen thinpen; endfor;
48       draw eye(−91) shifted o shifted (0, u);
49       f1 := (−1/4u, 1/2d1*u) −− ((ll3 + t4)*u, 1/2d1*u)
50           −− ((ll3 + t4)*u, 1/2d2*u);
51       f2 := (ll1*u − 1/8u, 1/2d2*u) −− (ll5*u − 1/2d2*u, 1/2d2*u)
52           −− (ll5*u − 1/2d2*u, ypart(o));
53       f3 := (ll1*u − 1/8u, −1/2d2*u) −− (ll5*u + 1/2d2*u, −1/2d2*u)
54           −− (ll5*u + 1/2d2*u, ypart(o));
55       draw f1 withpen fatpen; draw f1 yscaled −1 withpen fatpen;
56       draw f2 withpen fatpen; draw f3 withpen fatpen;
```
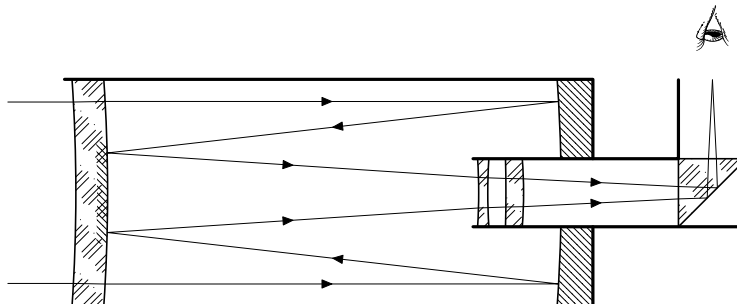
## 8.2 L'Hôpital's Pulley Problem

Line 3 describe initial setup. Line 4 is problem's solution. Lines 8–11 set all the points where they belong. Lines 12–16 are needed to decide where to put the pulley.

```
1       pair p[] , o [];
2       numeric a, d[] , l [] , x [] , y [];
3       l0 := 6; l1 := 4; l2 := 4;
4       x1 := ( l1∗∗2 + abs( l1 )∗(( sqrt (8)∗l0)++l1))/4 l0 ;
5       y1 := l1+−+x1 ;
6       y2 := l2 − (( l0−x1)++y1);
7       d1 := 2/3cm; d2 := 4/3cm; d3 := 5/6 d1 ;
8       p1 := (0, 0);
9       p2 := (l0∗cm, 0);
10      p3 := (x1∗cm, −y1∗cm);
11      p4 := p3 shifted (0, −y2∗cm);
12      o1 := (unitvector (p4−p3) rotated 90 scaled 1/2 d3 );
13      o2 := (unitvector (p3−p2) rotated 90 scaled 1/2 d3 );
14      p5 := whatever [ p3 shifted o1, p4 shifted o1]
15         = whatever [ p3 shifted o2, p2 shifted o2 ];
16      a := angle (p1−p3);
17      draw solidSurface (11/10[p1,p2] −− 11/10[p2, p1]);
18      draw pulley (d1, a − 90) shifted p5;
19      draw image (
20         draw p1 −− p3 −− p2 withpen thickpen ;
21         draw p3 −− p4 withpen thickpen ;
22      ) maskedWith ( pulleyOutline shifted p5 );
23      draw sphere.c(d2) shifted p4 shifted (0, −1/2 d2 );
24      dotlabel. llft (btex $A$ etex , p1);
25      dotlabel. lrt (btex $B$ etex , p2);
26      dotlabel. ulft (btex $C$ etex , p4);
27      label. llft (btex $l$ etex , 1/2[p1, p3]);
28      markAngle(p3, p1, p2)(btex $\alpha$ etex );
```
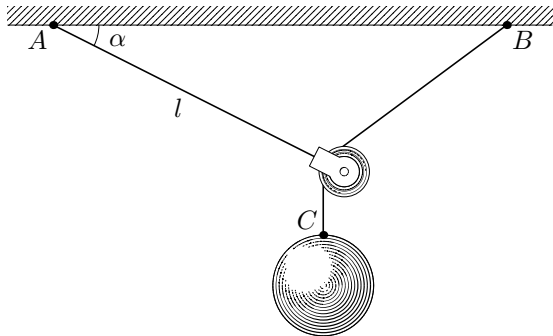


## 8.3 Hooke's law

```
1       numeric l [] ,d, h;
2       pair p[] , q [];
3       l0 := 3/2cm; l1 := 1cm;
4       d := 1cm; h := 7/8cm;
5       p1 := (0, 0); p2 := (0, −l0);
6       p3 := (d, 0); p4 := (d, −l0−l1);
7       p5 := (2d, 0); p6 := (2d, −l0−2l1); p7 := (2d, −l0−2l1−h);
8       draw solidSurface ((2d + 1/2cm, 0)−−(−1/2cm, 0));
9       draw spring (p1, p2, 20);
10      draw spring (p3, p4, 20);
11      draw weight.h(h) shifted p4;
12      draw spring (p5, p6, 20);
```
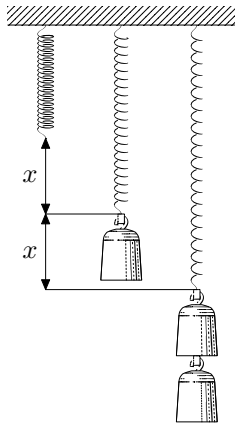
```
13        draw weight.h(h) shifted p6;
14        draw weight.h(h) shifted p7;
15        q1 := (0, ypart(p2));
16        q2 := (0, ypart(p4));
17        q3 := (0, ypart(p6));
18        draw p2 -- q1 withpen thinpen;
19        draw p4 -- q2 withpen thinpen;
20        draw p6 -- q3 withpen thinpen;
21        drawdblarrow q1--q2 withpen thinpen;
22        drawdblarrow q2--q3 withpen thinpen;
23        label.lft (btex $x$ etex, 1/2[q1, q2]);
24        label.lft (btex $x$ etex, 1/2[q2, q3]);
```



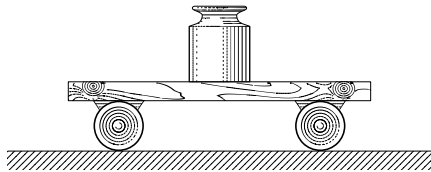## 8.4   Weight on a cart

```
1         numeric l, w, r,  h;
2         l := 4cm;
3         w := 1/4cm;
4         r := 2/3cm;
5         h := 1cm;
6         draw solidSurface((-1/5l, 0) -- (6/5l, 0));
7         draw woodBlock (l, w) shifted (0, r);
8         draw wheel (r, 0) shifted (r, 1/2r);
9         draw wheel (r, 0) shifted (l-r, 1/2r);
10        draw weight.s(h) shifted (1/2l, r + w);
```



## 8.5   Some knots

```
1         path p[];
2         p1 := (dir(90)*4/3cm) {dir(0)} .. tension 3/2
3              .. (dir(90 + 120)*4/3cm){dir(90 + 30)} .. tension 3/2
4              .. (dir(90 - 120)*4/3cm){dir(-90 - 30)} .. tension 3/2
5              .. cycle;
6         p1 := p1 scaled 6/5;
7         addStrandToKnot (primeOne) (p1, 1/4cm, "l", "1,␣-1,␣1");
8         draw knotFromStrands (primeOne);
9         p2 := (0, 2cm) .. (1/2cm, 3/2cm) .. (-1/2cm, 0)
10             .. (1/2cm, -2/3cm) .. (4/3cm, 0) .. (0, 17/12cm)
```

```
11              ..  (−4/3cm,  0)  ..  (−1/2cm,  −2/3cm)  ..  (1/2cm,  0)
12              ..  (−1/2cm,  3/2cm)  ..  cycle;
13         p2 := p2 scaled 6/5;
14         addStrandToKnot (primeTwo) (p2, 1/4cm, "l", "1,_−1,_1,_−1,_1");
15         draw knotFromStrands (primeTwo) shifted (4cm, 0);
16         p3 := (dir(0)∗3/2cm)  ..  (dir(1∗72)∗2/3cm)
17              ..  (dir(2∗72)∗3/2cm)  ..  (dir(3∗72)∗2/3cm)
18              ..  (dir(4∗72)∗3/2cm)  ..  (dir(0)∗2/3cm)
19              ..   (dir(1∗72)∗3/2cm)  ..  (dir(2∗72)∗2/3cm)
20              ..  (dir(3∗72)∗3/2cm)  ..  (dir(4∗72)∗2/3cm)
21              ..  cycle;
22         p3 := (p3 rotated (72/4)) scaled 6/5;
23         addStrandToKnot (primeThree) (p3, 1/4cm, "l", "−1,_1,_−1,_1,_−1");
24         draw knotFromStrands (primeThree) shifted (8cm, 0);
```