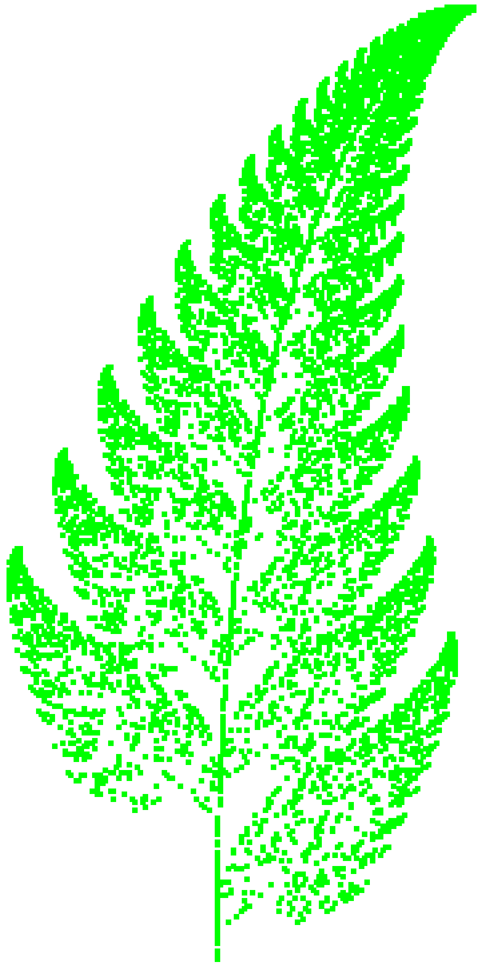
```
fractal := proc(n)
  local Mat1, Mat2, Mat3, Mat4,
  Vector1, Vector2, Vector3, Vector4,
  Prob1, Prob2, Prob3, Prob4,
  P, prob, counter, fractalplot,
  starttime, endtime;
  Mat1 := Matrix([[0.0, 0.0], [0.0, 0.16]]);
  Mat2 := Matrix([[0.85, 0.04], [-0.04, 0.85]]);
  Mat3 := Matrix([[0.2,-0.26], [0.23, 0.22]]);
  Mat4 := Matrix([[-0.15, 0.28], [0.26, 0.24]]);
  Vector1 := Vector([0, 0]);
  Vector2 := Vector([0, 1.6]);
  Vector3 := Vector([0, 1.6]);
  Vector4 := Vector([0, 0.44]);
  Prob1 := 0.01;
  Prob2 := 0.85;
  Prob3 := 0.07;
  Prob4 := 0.07;
  P := Vector([0, 0]);
  writedata("/Users/ulflorenz/tmp/fractaldata", [[P[1], P[2]]], [float,float]);

  starttime := time() :
  for counter from 1 to n do
    prob := rand()/10^12;
    if prob < Prob1 then P := Mat1.P + Vector1
      elif prob < Prob1 + Prob2 then P := Mat2.P + Vector2
      elif prob < Prob1 + Prob2 + Prob3 then P := Mat3.P + Vector3
      else P := Mat4.P + Vector4;
    fi;
   writedata[APPEND]("/Users/ulflorenz/tmp/fractaldata", [[P[1], P[2]]], [float,float]);
   od;
  fractalplot := readdata("/Users/ulflorenz/tmp/fractaldata", 2);
  print(plot(fractalplot, scaling = constrained,
        axes = none, color = green, title = cat(n, " iterations"), style = point, symbol = point));
  fremove("/Users/ulflorenz/tmp/fractaldata");
  endtime := time() :
  printf("Execution time was %a seconds.", endtime - starttime);
end:

fractal(10000);
```

10000 iterations



```
Execution time was 12.270 seconds.
```

The mathematics underlying this code is the following iteration scheme. Pick a vector in the plane and apply an affine transformation (multiply by a matrix and add some vector to the result). Plot the resulting point. Apply to the new point a possibly different affine transformation. Repeat. In the given example, there are four different affine transformations involved, and the one that is picked at a given step is randomized; each transformation has a specified probability of being chosen at any particular step.

Simpler example:

Similar-to-Fibonacci-Numbers are

$sff[1] := 0;\ sff[2] := 1; sff[3] := 1; sff[i] := sff[i-1] + sff[i-2] + 1$

> $restart; sff := [seq(0,\ i = 1..100)]; sff2 := Array(1..1000);$

$sff := [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,$
$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,$
$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

$$sff2 := \begin{bmatrix} \textit{1 .. 1000 Array} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix} \quad \textbf{(1)}$$

```
>
> sff[2] := 1; sff[3] := 1; sff2[2] := 1; sff2[3] := 1;
```

$$sff_2 := 1$$
$$sff_3 := 1$$
$$sff2_2 := 1$$
$$sff2_3 := 1 \quad \textbf{(2)}$$

```
> sff[5] := sff[3] + sff[4] + 1;
```

$$sff_5 := 2 \quad \textbf{(3)}$$

```
> for i from 4 to 100 do
    sff[i] := sff[i − 1] + sff[i − 2] + 1;
    if i = 97 then print(sff[i]) fi;
  end do:
```

103361417709716646143 **(4)**

```
> sff1 := sff[1]; sff2 := sff[2]; sff3 := sff[3];
```

$$sff1 := 0$$
$$sff2 := 1$$
$$sff3 := 1 \quad \textbf{(5)}$$

```
> for i from 4 to 10000 do
    sff4 := sff3 + sff2 + 1;
    sff2 := sff3 : sff3 := sff4;
    if i = 997 then print(sff3) fi;
  end do:
```

12683370600837669425873747487304959404558986155565407569187860372438331470308\
91742558530143341688129203272323535263040122297871663969739134207554410971\
9681422127844715800860260531567430904209964480196551867743 **(6)**

Simple commands

e.g. all direct commands we saw so far.

Comparison Operators (<, >, >, <=, >=)

```
> a := 0; b := 1;
```

$$a := 0$$

(7)

$$b := 1 \quad (7)$$

```
> evalb(a = 0); #evalb prints boolean results to screen
```

$$true \quad (8)$$

```
> evalb(b > 2);
```

$$false \quad (9)$$

```
> evalb(b + a <= 0);
```

$$false \quad (10)$$

```
> a = 0;
```

$$0 = 0 \quad (11)$$

Flow Control (if, for, while, ...)

if <conditional expression> **then** <statement sequence>
| **elif** <conditional expression> **then** <statement sequence> |
| **else** <statement sequence> |
end if

(Note: Phrases located between | | are optional.)

```
> if (a > 0) then f := x^2 fi;
> if (a = 0) then f := x^2 fi;
```

$$f := x^2 \quad (12)$$

```
> if (a < 9) then
    f := x^2 + 1; #  ";" is necessary, because: several statements without structure
    g := x^2          # ";" not necessary
  else
    g := x^2 + 1;
    f := x^2;
  end if;
```

$$f := x^2 + 1$$
$$g := x^2 \quad (13)$$

The **for ...while ... do** loop

```
>
>
```

1) Print even numbers from 6 to 10.

```
> for i from 6 by 2 to 10 do print(i) end do;
```

6

8

10 (14)

2) Find the sum of all two-digit odd numbers from 11 to 99.

```
> mysum := 0;
  for i from 11 by 2 while i < 100 do
    mysum := mysum + i
  end do:
  mysum;
```

$$mysum := 0$$

$$2475 \quad (15)$$

3) Multiply the entries of an expression sequence.

```
> restart;
  total := 1 :
  for z in 1, x, y, q^2, 3 do
    total := total·z
  end do:
  total;
  x := 2 :
  q := 3 :
  total;
```

$$3\,x\,y\,q^2$$

$$54\,y \quad (16)$$

3) Add together the contents of a list.

```
> ?cat
```

```
> restart;
  y := 3;
  myconstruction := "";
  for z in [1, "+", y, "·", "q^2", "·", 3] do
    myconstruction := cat(myconstruction, z)
  end do;
  myconstruction;
```

$y := 3$

myconstruction := ""

myconstruction := "1"

myconstruction := "1+"

myconstruction := "1+3"

myconstruction := "1+3*"

myconstruction := "1+3*q^2"

myconstruction := "1+3*q^2*"

myconstruction := "1+3*q^2*3"

"1+3*q^2*3" **(17)**

```
> ?parse
```

```
> q := 4;
```

$$q := 4 \quad (18)$$

```
> qq := parse(myconstruction);
```

$$qq := 1 + 9\,q^2 \quad (19)$$

```
> qq;
```

$$145 \quad (20)$$

Procedures

Flow control constructions, simple commands and comparison operators can be bound together; in a so called
procedure. The simplest possible procedure looks as follow.

proc(parameter sequence)
statements;
end proc:

```
> restart;
  myfactorial := proc(n)
     local r, i;
     r := 1;
     for i from 1 by 1 to n do
       r := r · i;
       #print(r);
     od;
     return r;
   end proc;
```

myfactorial := **proc**(n) **local** r, i; r := 1; **for** i **to** n **do** r := r * i **end do**; **return** r **end proc** (21)

```
> myfactorial(4);
```

$$24 \quad (22)$$

Maple allows recursive procedure calls:

```
> restart;
   myfactorial2 := proc(n)
     if (n < 2) then return 1
     else return n·myfactorial2(n − 1);
     fi;
   end proc;
```

myfactorial2 := **proc**(n) (23)

if n < 2 **then return** 1 **else return** n * myfactorial2(n − 1) **end if**

end proc

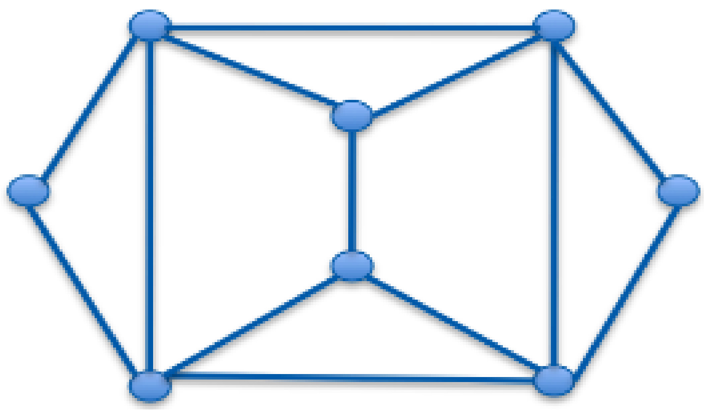
```
> myfactorial2(4);
```

myfactorial2(4) (24)

Graphs, Paths and Matrices

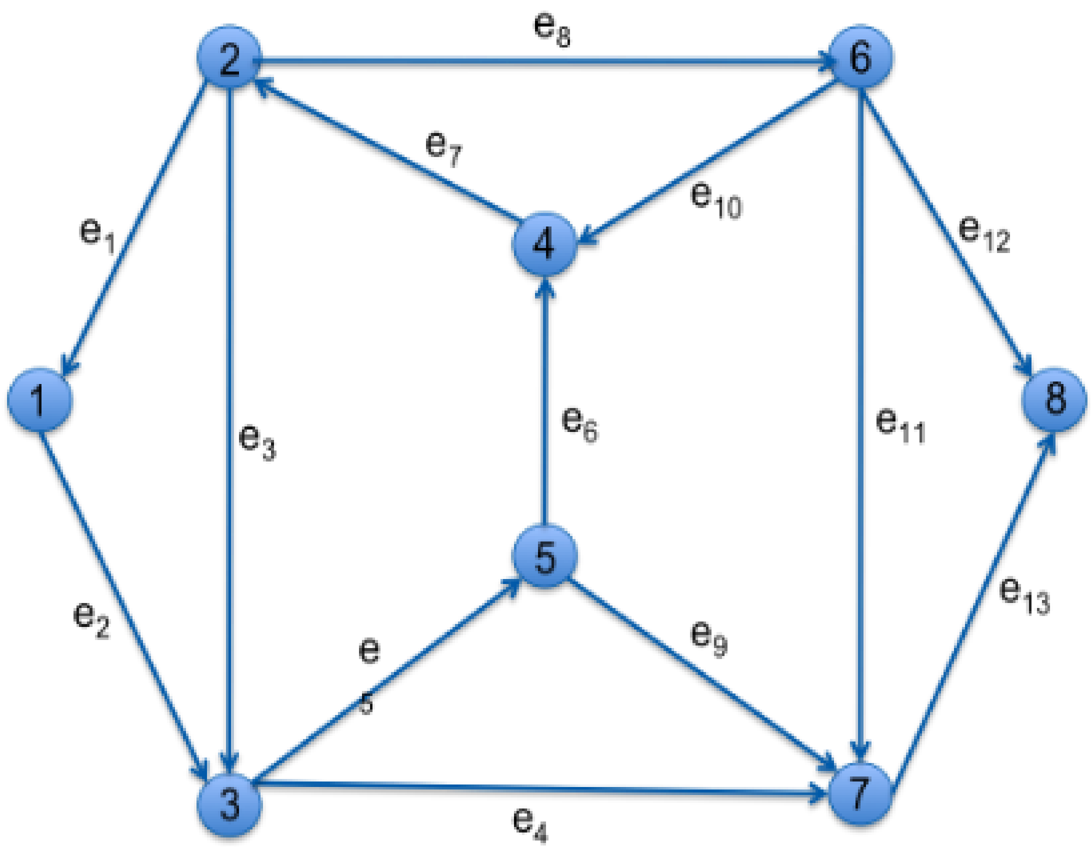
What is a graph?

An undirected graph consists of a pair G=(V,E), where $E \subseteq \{\{u,v\} \mid u,v \in V\}$.
The elements of E are not ordered..



Elements from V are called nodes (or vertices; Knoten in dt.), elements from E are called edges (Kanten in dt.)

A directed graph (gerichteter Graph) is as well a pair G=(V,E). However, the elements of E are ordered pairs of elements from V. Thus, $E \subseteq \{(u,v) \mid u,v \in V\}$.
Elemente of V are called nodes, elements from E are called edges (im dt.: gerichtete Kanten oder Bögen).



The node-edge-incidence matrix (Knoten-Kanten-Inzidenzmatrix) is one of may possibilitzies how to encode graphs. The lines represent nodes, the columns represent edges:

$$A := Matrix([\ [\ 1,-1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0],$$
$$[-1,\ 0,-1,\ 0,\ 0,\ 0,\ 1,-1,\ 0,\ 0,\ 0,\ 0,\ 0],$$
$$[\ 0,\ 1,\ 1,-1,-1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0],$$
$$[\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,-1,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0],$$
$$[\ 0,\ 0,\ 0,\ 0,\ 1,-1,\ 0,\ 0,-1,\ 0,\ 0,\ 0,\ 0],$$
$$[\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,-1,-1,-1,\ 0],$$
$$[\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,-1],$$
$$[\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1]\ ]);$$

$$\left[\begin{array}{c} \textit{8 x 13 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{array}\right] \qquad \textbf{(25)}$$

$x := \langle 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1\rangle : A.x;$

$$\begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \textbf{(26)}$$

$$x := \left\langle 0, 1, 0, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, \frac{1}{2}, \frac{1}{2} \right\rangle : A.x$$

$$\begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \textbf{(27)}$$

x describes a (selected) subset of the edge-set. We can interpret this as moving a (part of a) unit over edges. If we demand $A_i = 0$ for all $2 \le i \le 13$, we encode so called flow conservation.