Winfried Auzinger
Dirk Praetorius

Sommersemester 2013
15./16.05.2013

Übungsaufgaben zur VU Computermathematik

Serie 7

Eine Sammlung verschiedener Problemstellungen.

Exercise 7.1. Design a procedure `mynormal(r)` which expects a rational expression with integer coefficients as its argument and normalizes it in such a way that the numerator and denominator polynomials are 'monic', i.e., the coefficients at their highest power is 1, with an extra factor such that the result is correct.

Hint: Use `degree` and `coeff`.

Exercise 7.2. Design a procedure `argmax(x::{Vector,Matrix}),absolute_value)` which accepts arguments of type `Vector` or `Matrix` and returns the positions[1] of the maximal elements (in the case `absolute_value=false`), or the position of the elements with the largest absolute value (in the case `absolute_value=true`), in a numerical object `x` of type `Vector` or `Matrix`. For a `Matrix`, the 'position' is the corresponding pair of indices. Return the answer in form of a list [of lists] ordered by position and, in addition, the value of the entries where the maximum is attained.

(In Maple, there is `max` but there seems not to exist something like `argmax`.)

Use sequential search. For the case of a `Matrix`, design two versions of `argmax`: Sequential search through the matrix, or iterating over its rows or columns using recursive calls of `argmax`.

Hint: Using `type` you can determine the type of an object.

Exercise 7.3. Some advanced programming features:

• The `overload` command allows you to split the implementation of a command that operates on different argument types into separate procedures. The basic syntax is `overload(l)` where `l` is a list of procedures. A simple example: Assume you want to represent matrix right division $A/B := A \cdot B^{-1}$ and normal division a/b by a single function `slash`. This works as follows:

```
slash  := overload([

          proc(A::Matrix,B::Matrix)
          option overload:
              return A.B^(-1)
          end proc,

          proc(a::anything,b::anything)
          option overload:
             return a/b
          end proc

          ]):
```

[1] The maximal element may not be unique.

The first branch only accepts arguments of type `Matrix`. The data type `anything` means 'any possible type', i.e., the second procedure acts as a default. Such a construction is often more useful than embedding various `if` - constructs into a single procedure.

• The `try ... catch ... end try` construct allows you to 'protect' parts of your code, with a controlled error handling by the `catch` - branch if the `try` - branch fails. A simple example:

```
try
   M := Matrix(m,m);
   N := Matrix(n,n);
   K := Matrix(k,k);
catch:
   error "One of the matrix dimension has not been correctly specified."
end try;
```

This is often more useful than trying to avoid in advance that such an error occurs by various `if` - constructs.

a) Verify the above simple example for use of `overload`, and extend it by a `try ... catch ... end try` construct which, for the case that dividing by `B` or `b`, respectively, an error occurs, the `catch` - branch forces a return of `infinity`.

b) Refine `a)` in the following way:

Add extra `if` - constructs in order to return `FAIL` when both arguments passed to `slash` are 0 or zero-matrices, respectively, if `A` or `B` is not a square matrix, or if the dimensions of the (square) matrices `A` and `B` are different.

Exercise 7.4. Check the help page `? index` and look for `packages`. Here you see a complete list of packages. Make your choice, study the documentation, and create a worksheet with explaining features and showing some examples.

A package is activated via `with(`*name_of_package*`);`

Example: If you like discrete mathematics and combinatorics, you may look at the package `combinat`.

Exercise 7.5. Let a triangle $\Delta = \overline{ABC} \subseteq \mathbb{R}^2$ be given, with $A = (x_1, y_1)$, $B = (x_2, y_2)$, and $C = (x_3, y_3)$. To compute the integral of a real-valued function $f(x, y)$ defined over Δ, we represent points the $(x, y) \in \Delta$ in the form

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \xi \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix} + \eta \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \end{bmatrix}, \quad 0 \le \xi + \eta \le 1,$$

with coordinates $(\xi, \eta) \in \Delta_{ref}$, where Δ_{ref} is the simple 'reference triangle' with vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$. Note that $(x(0,0), y(0,0)) = (x_1, y_1)$, $(x(1,0), y(1,0)) = (x_2, y_2)$, and $(x(0,1), y(0,1)) = (x_3, y_3)$.

The 2-dimensional substitution formula reads

$$\iint_{\Delta} f(x,y)\, dy\, dx \;=\; \iint_{\Delta_{ref}} \delta(\xi,\eta) f(x(\xi,\eta), y(\xi,\eta))\, d\eta\, d\xi\,.$$

with the Jacobian determinant $\delta(\xi, \eta)$ of the coordinate transformation. In our case, $\delta(\xi, \eta)$ is constant:

$$\delta(\xi,\eta) \;\equiv\; \delta \;=\; (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1),$$

which corresponds to the ratio of the areas of the two triangles. Thus,

$$\iint_{\Delta} f(x,y)\, dy\, dx \;=\; \delta \cdot \int_{\xi=0}^{1} \int_{\eta=0}^{1-\xi} f(x(\xi,\eta), y(\xi,\eta))\, d\eta\, d\xi\,.$$

Design a procedure `triangleint(Delta,f)` which computes the integral in this way Specify the vertices of the triangle by `Delta=[[x[1],y[1]],[x[2],y[2]],[x[3],y[3]]]`.

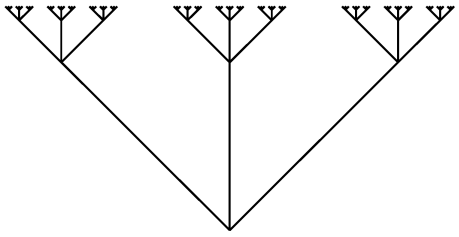
Exercise 7.6. The function `plots[animate]` can be used to produce videos, i.e., a sequence of plots depending on a parameter. After defining the corresponding plot structure, rendering of the animation is done in an interactive way. Consult the help page for `plots[animate]`.

Use this feature to visualize the behavior of the Taylor polynomials

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + \ldots \pm \frac{x^n}{n}$$

of the function $\ln(1+x)$ in the interval $(-1, 1)$. Create a nice animation. In this case, the varying parameter is the polynomial degree n.

Exercise 7.7.



Design a recursive procedure `tree(level,pos,len,scale)` which generates a plot of a 'fractal' tree. The arguments:

- `level` specifies the desired number of recursive levels.
- `pos` is a list `[x,y]` containing the cartesian coordinates of the root of the current tree.
- The current tree consists of 3 branches emanating from the root with inclination $-45°$, vertically, and $+45°$, and `len` specifies the current length of the vertical branch of the current tree, see figure.
- `scale` is a factor with which `len` is multiplied when proceeding to the next recursive level.

The recursion works in the following way.

`tree(level,pos,len,scale)`

does the following job:

(i) Generate the 3 branches of the tree specified by the arguments `pos` and `len`.

(ii) If `level>1`: Perform 3 recursive calls of tree with `level-1`, 3 new values for `pos` which correspond to the endpoints of the branches of the current tree, and with `len` replaced by `scale*len`.

The figure displayed above was generated by calling `tree(4,1,[0,0],1/4)`.

Hint: Use global variables `p` and `counter`. Before calling `tree`, these are initialized: `p:='p'`, and `counter:=0`. When `tree` builds up a branch, the value `counter` of is increased by 1, the plot of the branch is generated using, e.g., `plots[pointplot]`, and the resulting plot structure is saved in `p[counter]`. After activating `tree` you can render these plots using `plots[display]`. Playing with the parameters `level` (initial value on call) and `scale` you can generate different funny trees.

Exercise 7.8. Generalize your procedure `tree` from 7.7. Play with plot options (e.g., line thickness, color), and generalize the procedure by plotting more branches at different angles, or by introducing a certain stochastic behavior (e.g., random number of branches of a random behavior of `scale` at each level, using `rand()`), or whatever you may try.