Lab 2. Maple programming

Note. Use Maple procedures for the following tasks.

1. Implement the conditional operator **if … then … elif … else** : if x is not a natural number, then print a message about this, otherwise check the number of digits. For a single-digit number, print a message stating that "x is a single-digit number", for a two-digit number, print a message stating that "x is a two-digit number", in other cases, print a message stating that "x is a multi-digit number with more than two digits". To check, consider several numbers. In messages, use the substitution of the value of x. To determine whether the number x is a natural number, you can use the function **`type(x,posint).`** To print a message with the substitution of the value, use the command **`printf("%a is a natural number.",x );`**

2. **while** loop : display all prime numbers between 1 and 100 as a sequence . Organize the sequence formation in a loop. Do the same with the **for / from operator .** To determine whether a number x is prime, use the **`isprime (x) function`** .

3. **while** loop : display a list of all perfect squares in the range from 1 to 100. Organize the list generation in a loop. Do the same with the **for / from operator .** Do the same with the **`select`** command**.** To determine whether a number is a perfect square, use the **`issqr`** command.

4. **for / in** loop: calculate the product of those numbers in the range from 1 to 1000 that are perfect cubes, using set structure. Before executing the loop, generate a set of such numbers (perfect cubes from 1 to 1000) using the $ operator.

5. **for / in** loop: convert a list of values in degrees [0, 45, 90, 180, 360] to radians. Also output the resulting values as a list. Organize the list growth in a loop. Do the same using the **`seq (..,.. in .. )`** command**.**

6. **for / from … while** loop: calculate the sum of the squares of the first fifteen prime numbers. Print the resulting sum and the value of the loop counter.

7. **for / in … while** loop : calculate the sum of those numbers in the range from 1 to 200 that are both perfect squares and perfect cubes. Before executing the loop, generate only the set of perfect cubes (using the $ operator). Implement the check for a perfect square in the loop body using a conditional operator and the **`issqr`** command.

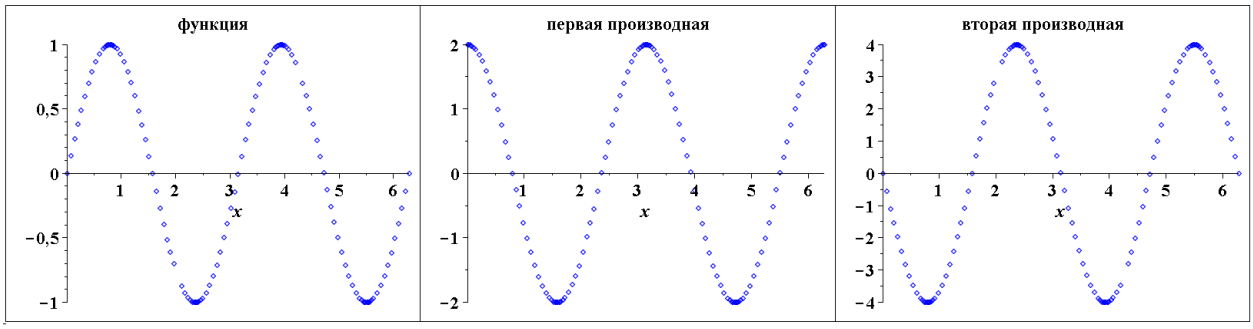
8. Print the stem of each word in the string “Maple is my favorite mathematical software " using the **`map`** command**.** To work with strings, use the **`Split`** and **`Stem`** commands from the **`StringTools`** package.

9. Write a procedure *gm_pos* that returns the geometric mean of two numbers and accepts only positive integers (add a **`posint`** type declaration to the arguments). Additionally**,** specify default values: $a = 1$, $b = 1$. Test the procedure and explain the results (numbers to test: 13 and 17, 13.5 and 17.5, -13 and 17, 13 and -17).

10. Write a procedure, *testSC* , that checks whether a number is both a perfect square and a perfect cube. The procedure should accept one or more arguments (i.e., be able to test several numbers at once as a sequence). If the answer is yes, the procedure should return true , and if the answer is no, it should return false . To check for a perfect square, use the **`issqr ( n )`** function . To check for a perfect cube, write an auxiliary user-defined procedure , *IsCub* (look at the **`issqr`** procedure codein Maple (see the lectures for displaying the procedure code on the screen) and write the same for a perfect cube). To write a procedure with an arbitrary number of arguments, use the reserved names **`_passed`** and **`_npassed.`** Test the procedure on various numbers.

    To check , the first few numbers that are both perfect squares and cubes are in the intersection set of perfect squares and perfect cubes.

11. Write a procedure named *PlotDerivatives* that plots a function, its first derivative, and its second derivative on three adjacent plots. The procedure must contain two required arguments: the function (of type **`algebraic`**) and the plot range along the x-axis (of type **`range ( realcons )`**). The remaining arguments can be plotting options that should be passed directly to the **`plot`** commands. Output a suitable title for each plot, for example:



Test the procedure on the commands:

```
> plotDerivatives ( arcsin (x ),-1.. 1);
> plotDerivatives (sin(2*x ),0.. 2* Pi,color = blue,style
  =point, font=[ TIMES,BOLD ,16]);
```

Hint : Use the reserved name **`_rest`** and include the **`plots`** package in the procedure body. To display three plots side by side, define a 1 x 3 array **`of`** three plot commands **`and`** use the **`display`** command from the **`plots`** package to print this array to the screen. To define the array:

```
A:=Array(1..3,[plot( … ),plot (… ),plot (…)] );display (A)
```

12. Write a recursive procedure *myfactorial* that calculates n ! for any nonnegative integer n (type **`nonnegint`**) without using the factorial sign. If the argument type mismatches, the procedure should display a message indicating that the number is not a natural number and return the number itself.

    Hint : Use the **`return`** command in the procedure body. The recurrence formula for calculating n ! is:

$$n! = \begin{cases} 1, & n = 0; \\ (n-1)! \cdot n, & n \geqslant 1. \end{cases}$$

    To check . Use the command **`n !`**

13. Write a procedure, *DigitCount2* , that calculates the sum of the digits of a given two-digit natural number. Use a type declaration for the argument (type **posint**) . If the given number is not two-digit, arrange for the procedure to exit abnormally with an error message (use the **error** command, substituting the value of the entered number). Test the procedure.

    Hint : To get the quotient and remainder when dividing by 10, you can use the commands **iquo(n ,10)** and **irem(n ,10)** .

Extra task for additional points

14. Write a procedure, *backsub* , that solves a system of linear algebraic equations Ax = b with an upper triangular matrix A using the backsubstitution method. The method is as follows. First, the last unknown is calculated $x_n = b_n / a_{nn}$, then the remaining unknowns are calculated using the rule:

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj} x_j}{a_{kk}} \text{ For } k = n-1,\ n-2,\ ...,1..$$

    The procedure should include:

    - declaration of argument types (Matrix , Vector)
    - use of the necessary packages
    - checking argument types and outputting appropriate user error messages
    - output error messages in cases where the matrix dimensions do not match the vector dimensions, the matrix is not upper triangular , or the matrix is singular.

    Test the procedure on the given example and an arbitrary matrix. Compare the result with the **BackwardSubstitute** and **LinearSolve** commands in the **LinearAlgebra** package .

$$\begin{aligned} 3x_1 - 2x_2 + x_3 - x_4 &= 8 \\ 4x_2 - x_3 + 2x_4 &= -3 \\ 2x_3 + 3x_4 &= 11 \\ 5x_4 &= 15 \end{aligned}$$

    Then try entering an incomplete set of arguments, arguments of the wrong type, or an excessive number of arguments, and see what error messages are returned. In what cases are custom error messages and user error messages returned?

Note: The matrix from the example can be defined with the command

```
M:=Matrix(4,[ [3 ],[ -2,4 ],[1,- 1,2 ],[ -1,2,3,5] ],shape =
   triangular,scan = columns);
```

Arbitrary matrix :

```
M:= RandomMatrix( 4,outputoptions =[shape=triangular]);
M:= RandomMatrix( 4,outputoptions =[shape= triangular,datatype
   =float]);
```

To determine whether a matrix is upper triangular , you can convert it to upper triangular form and compare the resulting matrix with the original. The **Equal** command from the **LinearAlgebra** package compares matrices for equality of elements.