MuPAD Programming

Question 1:
Given y=sin(3x)cos(2x), evaluate dy/dx at x=nπ for n=0,1,...6.

Answer 1 Not automated, no comments

```
reset():
```

```
diff(sin(3*x)*cos(2*x),x)|x=0;
```

$$3$$

```
diff(sin(3*x)*cos(2*x),x)|x=PI;
```

$$-3$$

```
diff(sin(3*x)*cos(2*x),x)|x=2*PI;
```

$$3$$

```
diff(sin(3*x)*cos(2*x),x)|x=3*PI;
```

$$-3$$

```
diff(sin(3*x)*cos(2*x),x)|x=4*PI;
```

$$3$$

```
diff(sin(3*x)*cos(2*x),x)|x=5*PI;
```

$$-3$$

```
diff(sin(3*x)*cos(2*x),x)|x=6*PI;
```

$$3$$

Answer 2

Assign the given expression to y

```
reset():
```

```
y:=sin(3*x)*cos(2*x):
```

Now use a loop to evaluate all the required derivatives

```
for n from 0 to 6 do
   print(diff(y,x)|x=n*PI);
end_for;
```

$$3$$
$$-3$$
$$3$$
$$-3$$
$$3$$
$$-3$$
$$3$$

So $dy/(dx)=3(-1)^n$ for n=0,1..6.

Question 2:
Find Taylor series for f(x) = tan(x)

1. The most basic way: no automation, clumsy, no easy checks:

```
tan(0)+(diff(tan(x),x)|x=0)*x+(diff(tan(x),x,x)|x=0)*x^2/2!+(diff(tan(x),x,x,x)|x=0)*x^3/3!;
```

$$\frac{x^3}{3}+x$$

Check:

```
exp(0)+(diff(exp(x),x)|x=0)*x+(diff(exp(x),x,x)|x=0)*x^2/2!+(diff(exp(x),x,x,x)|x=0)*x^3/3!;
```

$$\frac{x^3}{6}+\frac{x^2}{2}+x+1$$

2. Start to automate, with a loop to evaluate the coefficients:

```
a[0]:=tan(0);
```

$$0$$

```
for j from 1 to 5 do
   d[j]:=diff(tan(x),x$j):
   a[j]:=d[j]|x=0:
end_for:
```

```
sum(a[r]*x^r/r!,r=0..5);
```

$$\frac{2x^5}{15}+\frac{x^3}{3}+x$$

3. Now make the loop more general (so one can change n)

```
n:=19:
```

```
for j from 1 to n do
   d[j]:=diff(tan(x),x$j);
   a[j]:=d[j]|x=0;
```

```
end_for:
```

```
sum(a[r]*x^r/r!,r=0..n);
```

$$\frac{443861162\,x^{19}}{1856156927625}+\frac{6404582\,x^{17}}{10854718875}+\frac{929569\,x^{15}}{638512875}+\frac{21844\,x^{13}}{6081075}+\frac{1382\,x^{11}}{155925}+\frac{62\,x^{9}}{2835}+\frac{17\,x^{7}}{315}+\frac{2\,x^{5}}{15}+\frac{x^{3}}{3}+x$$

4. Now say we want to generalise more....to any suitably differentiable function:

```
reset():
```

```
MyPoly:=proc(f,n,x) local j,a,d,p;
  begin
  a[0]:=f(0):
  for j from 1 to n do
     d[j]:=diff(f(x),x$j):
     a[j]:=d[j]|x=0:
  end_for:
  sum(a[r]*(x^r)/r!,r=0..n):
end_proc:
```

5. Try with some inbuilt functions:

```
MyPoly(tan,5,x);
```

$$\frac{2\,x^{5}}{15}+\frac{x^{3}}{3}+x$$

```
MyPoly(exp,6,x);
```

$$\frac{x^{6}}{720}+\frac{x^{5}}{120}+\frac{x^{4}}{24}+\frac{x^{3}}{6}+\frac{x^{2}}{2}+x+1$$

Now try with a function we define ourselves:

```
f:=x->cos(2*x)+3*exp(x);
MyPoly(f,4,x)
```

$$x\to \cos(2\,x)+3\,e^{x}$$

$$\frac{19\,x^{4}}{24}+\frac{x^{3}}{2}-\frac{x^{2}}{2}+3\,x+4$$

And do a check:

```
g:=x->x^3;
MyPoly(g,3,x);
```

$$x\to x^{3}$$

$$x^{3}$$

6. Now add some comments:

```
reset():
```

```
// Choose an informative name that doesn't clash in MuPAD
// f is the input function whose series is required
// n (an integer) is the highest power required
MyPoly:=proc(f,n,x) local j,a,d,p;
  begin
  a[0]:=f(0):
  for j from 1 to n do
     d[j]:=diff(f(x),x$j): // This is the jth derivative
     a[j]:=d[j]|x=0: // evaluated at x=0
  end_for:
  sum(a[r]*(x^r)/r!,r=0..n): //output
end_proc:
MyPoly(tan,5,x);
```

$$\frac{2\,x^{5}}{15}+\frac{x^{3}}{3}+x$$

7. This time make the procedure output a function:

```
MyPolyFn:=proc(f,n,x) local j,a,d,p;
  begin
  a[0]:=f(0):
  for j from 1 to n do
     d[j]:=diff(f(x),x$j):
     a[j]:=d[j]|x=0:
  end_for:
  p:=sum(a[r]*(x^r)/r!,r=0..n):
  fp::unapply(p,x):
end_proc:
```

```
MyPolyFn(tan,5,x);
```

$$x\to \frac{2\,x^{5}}{15}+\frac{x^{3}}{3}+x$$

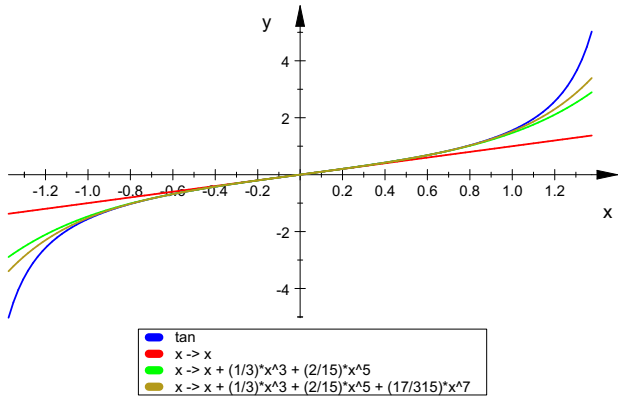
```
MyPolyFn(tan,5,x)(0.5);
```

$$0.5458333333$$

```
tan(0.5);
```

$$0.5463024898$$

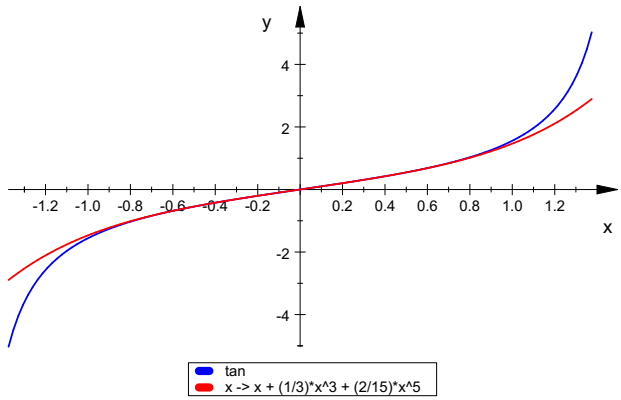
```
plot(tan,MyPolyFn(tan,1,x),MyPolyFn(tan,5,x),MyPolyFn(tan,7,x),x=-7*PI/16..7*PI/16,LegendVisible);
```



8. Now alter the procedure so that when called it automatically produces a graph of a particular Maclaurin's series and on the same diagram plots a graph of the underlying function that the series is trying to approximate

```
MyPolyFnGraph:=proc(f,n,x) local j,a,d,p;
  begin
  a[0]:=f(0):
  for j from 1 to n do
     d[j]:=diff(f(x),x$j):
     a[j]:=d[j]|x=0:
  end_for:
  p:=sum(a[r]*(x^r)/r!,r=0..n):
  fp::unapply(p,x):
  plot(f,MyPolyFn(f,n,x),x=-7*PI/16..7*PI/16,LegendVisible);
end_proc:
```

```
MyPolyFnGraph(tan,5,x);
```



======= End of question 2 =======

Question 3:

Write procedure which counts negative, positive and zero elements of the given list

```
reset():
```

```
f:=proc(a : Type::ListOf(Type::Real)) : Type::ListOf(Type::Integer)
  local n, z, p, i;
begin
  p:=0:
  z:=0:
  n:=0:
  for i from 1 to nops(a) do
    if op(a,i)>0 then p:=p+1:
    elif op(a,i)=0 then z:=z+1:
    else n:=n+1:
    end_if
  end_for:
  return ([n,p,z]):
end_proc:
```

```
a:=[1,2,0,3,0.00001,3,4,-1,2,1,-3,-0.1,-3,-4]:
f(a)
```

[5, 8, 1]

9. Write a proceudure that outputs the median of a set of numbers.
What do we need to do?
Sort the list in ascending order
Find out if we have an odd or an even number of entries
If odd, output the (n+1)/2 th value
If even, output the mean of the n/2 th and n/2 + 1 th values

```
reset():
```

```
Mediancalc:=proc(L) local L1,n;
  begin
  L1:=sort(L):
  n:=nops(L1):
  if testtype(n, Type::Odd)=TRUE then L1[(n+1)/2];
    else (L1[n/2]+L1[n/2+1])/2;
  end_if:
end_proc:
```

```
Mediancalc([1,4,2,3]);
```

$$\frac{5}{2}$$

10. The declaration of default values is demonstrated. The following procedure uses the default values if the procedure call does not provide all parameter values:

```
f := proc(x, y = 1, z = 2) begin [x, y, z] end_proc:
f(x, y, z), f(x, y), f(x)
```

$$[x, y, z], [x, y, 2], [x, 1, 2]$$

```
delete f:
```

11. The automatic type checking of procedure arguments and return values is demonstrated. The following procedure accepts only positive integers as argument:

```
f := proc(n : Type::PosInt) begin n! end_proc:
```

An error is raised if an unsuitable parameter is passed:

```
f(-1)
Error: The type of argument number 1 must be 'Type::PosInt'. The object '-1' is incorrect.
  Evaluating: f
```

In the following procedure, automatic type checking of the return value is invoked:

```
f := proc(n : Type::PosInt) : Type::Integer
begin
  n/2
end_proc:
```

An error is raised if the return value is not an integer:

```
f(3)
Error: The type 'Type::Integer' is expected for the return value. The type '3/2' is incorrect.
  Evaluating: f
```

```
delete f:
```

12. The option *hold* is demonstrated. With hold, the procedure sees the actual parameter in the form that was used in the procedure call. Without hold, the function only sees the value of the parameter:

```
f := proc(x) option hold; begin x end_proc:
g := proc(x) begin x end_proc:
x := PI/2:
f(sin(x) + 2) = g(sin(x) + 2), f(1/2 + 1/3) = g(1/2 + 1/3)
```

$$\sin(x) + 2 = 3,\ \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$$

Procedures using `option hold` can evaluate the arguments with the function context:

```
f := proc(x) option hold; begin x = context(x) end_proc:
f(sin(x) + 2), f(1/2 + 1/3)
```

$$\sin(x) + 2 = 3,\ \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$$

```
delete f, g, x:
```

13. The option *remember* is demonstrated. The *print* command inside the following procedure indicates if the procedure body is executed:

```
f:= proc(n : Type::PosInt)
option remember;
begin
   print("computing ".expr2text(n)."!");
   n!
end_proc:
f(5), f(10)
```

"computing 5!"

"computing 10!"

120, 3628800

When calling the procedure again, all values that were computed before are taken from the internal "remember table" without executing the procedure body again:

```
f(5)*f(10) + f(15)
```

"computing 15!"

```
1308109824000
```

`option remember` is used in the following procedure which computes the Fibbonacci numbers $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ recursively:

```
f := proc(n : Type::NonNegInt)
option remember;
begin
   if n = 0 or n = 1 then return(n) end_if;
   f(n - 1) + f(n - 2)
end_proc:
```

```
f(123)
```
$$22698374052006863956975682$$

Without `option remember`, the recursion is rather slow:

```
f := proc(n : Type::NonNegInt)
begin
   if n = 0 or n = 1 then return(n) end_if;
   f(n - 1) + f(n - 2)
end_proc:
```

```
f(28)
```
$$317811$$

```
delete f:
```

14. The `save` declaration is demonstrated. The following procedure changes the environment variable DIGITS internally. Because of `save DIGITS`, the original value of DIGITS is restored after return from the procedure:

```
myfloat := proc(x, digits)
save DIGITS;
begin
  DIGITS := digits;
  float(x);
end_proc:
```

The current value of DIGITS is:

```
DIGITS
```
$$10$$

With the default setting DIGITS = 10, the following float conversion suffers from numerical cancellation. Due to the higher internal precision, `myfloat` produces a more accurate result:

```
x := 10^20*(PI - 21053343141/6701487259):
float(x), myfloat(x, 20)
```
$$0.0,\ 0.02616405487$$

The value of DIGITS was not changed by the call to `myfloat`:

```
DIGITS
```
$$10$$

The following procedure needs a global identifier, because local variables cannot be used as integration variables in the int function. Internally, the global identifier `x` is deleted to make sure that `x` does not have a value:

```
f := proc(n)
save x;
begin
  delete x;
  int(x^n*exp(-x), x = 0..1)
end_proc:
```

```
x := 3: f(1), f(2), f(3)
```
$$1 - 2\,e^{-1},\ 2 - 5\,e^{-1},\ 6 - 16\,e^{-1}$$

Because of `save x`, the previously assigned value of `x` is restored after the integration:

```
x
```
$$3$$

```
delete myfloat, x, f:
```

15. The following procedure accepts an arbitrary number of arguments. It accesses the actual parameters via args, puts them into a list, reverses the list via revert, and returns its arguments in reverse order:

```
f := proc()
local arguments;
begin
   arguments := [args()];
   op(revert(arguments))
end_proc:
```

```
f(a, b, c)
```

$$c, b, a$$

```
f(1, 2, 3, 4, 5, 6, 7)
```

$$7, 6, 5, 4, 3, 2, 1$$

```
delete f:
```