Truncated Power Series Algebra

Taylor Expansions and Matrices out of Tracking Codes

Étienne Forest[1,2]

[1]High Energy Research Accelerator Organization Tsukuba, Japan

[2]Department of Accelerator Science
Graduate University for Advanced Studies, Hayama, Japan

December 2017

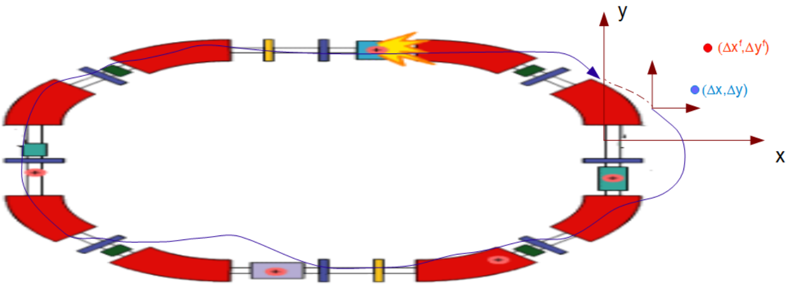
Outline

1. Central Theme beyond TPSA
2. Why TPSA?
3. TPSA primary operations: +,-,*,/
4. TPSA overloading of basic functions: exp(x) and atan(x) as examples
5. Putting things together: getting the map of a pendulum
6. Pendulum revisited: normalising programme
7. Running a Twiss loop in a code
8. A small code: Twiss loop and analytical formula

What is TPSA and why do we care?

The eigenvalues of the matrix M tell us about linear stability!

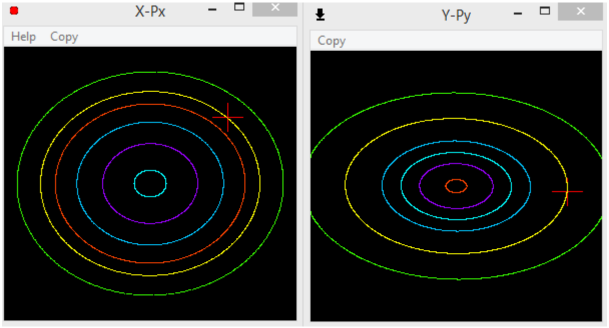
$$\begin{pmatrix}\Delta x^f\\ \Delta p_x^f\\ \Delta y^f\\ \Delta p_y^f\end{pmatrix} = M \begin{pmatrix}\Delta x\\ \Delta p_x\\ \Delta y\\ \Delta p_y\end{pmatrix}$$



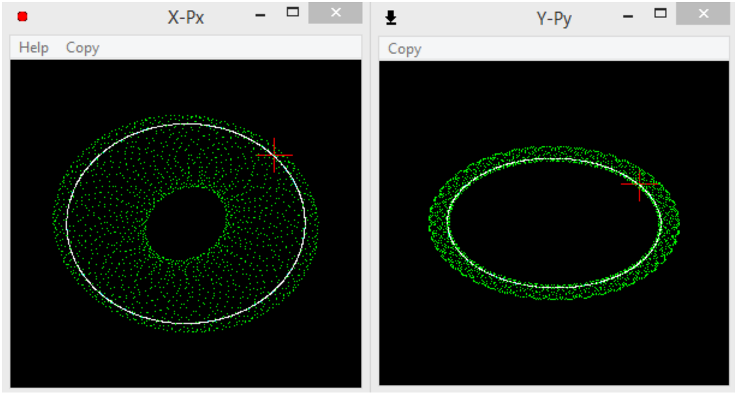
For example, can we get M without *any extra work*?

Uncoupled linear system

z_for_tpsa.f90 (ptc_run)

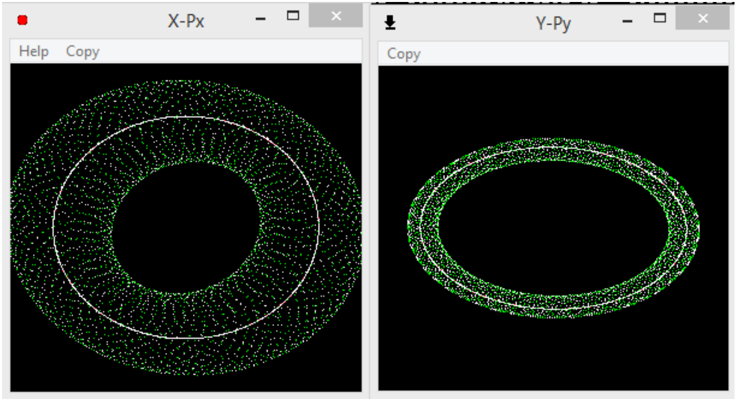


$$\left\langle x^2 \right\rangle = \widetilde{\beta}_x A_x = \beta_x \underbrace{\frac{A_x}{4\pi}}_{J_x} \qquad \left\langle y^2 \right\rangle = \widetilde{\beta}_y A_y = \beta_y \underbrace{\frac{A_y}{4\pi}}_{J_y} \tag{1}$$



1. The white curve corresponds to a linear uncoupled system
2. The green curve corresponds to a coupled system : $a_2 = 0.025$ in the first quadrupole QF1

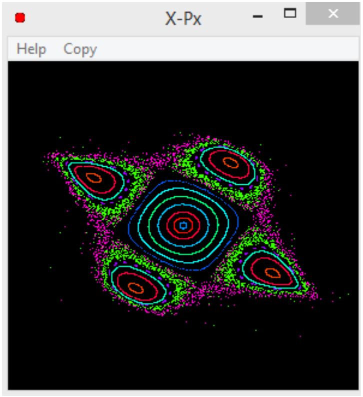
Coupled system



$$\left\langle x^2 \right\rangle = \widetilde{\beta}_{11} A_1 + \widetilde{\beta}_{12} A_2 = \beta_{11} J_1 + \beta_{12} J_2 \qquad \left\langle y^2 \right\rangle = \widetilde{\beta}_{21} A_1 + \widetilde{\beta}_{22} A_2 = \beta_{21} J_1 + \beta_{22} J_2 \tag{2}$$

Nonlinearity : Octupole kick in QF1

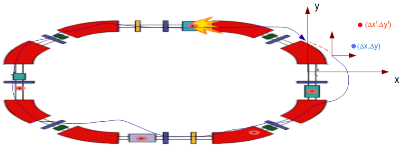
A figure like the one below can be explained with TPSA based tools except for the chaos between the central island and the four external ones.



$$\Delta p_x \approx -Lb_4 \left(x^3 - 3xy^2\right) \quad \text{and} \quad \Delta p_y \approx -Lb_4 \left(y^3 - 3x^2 y\right) \tag{3}$$

In the above example, $b_4 = -6000$.

Instead of using TPSA, some people write and use "matrix codes"



$$M = E_N \circ E_{N-1} \circ \cdots \circ E_2 \circ E_1 \quad (4)$$

1. Matrix codes have a catalogue of matrices for the *N* elements of the ring.
2. These matrices, in general Taylor maps, are computed around a special orbit, "the ideal orbit".
3. Therefore the total matrix of Eq. (4) is **not** the correct one for the blue orbit on the graph.
4. **TPSA can provide the matrix around the actual orbit, the blue orbit, provided you can compute the blue closed orbit.**
5. It is the purpose of an integrator to compute **any orbit.**

Suppose we can compute $\vec{z}^f = f\left(\vec{z}\right)$

Suppose we have a programme, *any programme*, *even a programme with a million physics mistakes*, which supposedly computes the trajectory for one turn:

$$\vec{z}^f = f\left(\vec{z}\right) \tag{5}$$

where

$$\vec{z} = (x, p_x) \quad \text{or} \quad \vec{z} = (x, p_x, y, p_y, \cdots) \tag{6}$$

and suppose that the programme computes the closed orbit of the ring

$$\vec{z}_c \quad \Rightarrow \quad \vec{z}_c = f\left(\vec{z}_c\right) \tag{7}$$

Motion around $\vec{z}_c$

Then we can re-express $\vec{z}$, around $\vec{z}_c$:

$$\vec{z} = \vec{z}_c + \Delta\vec{z} \Rightarrow \vec{z}^f = \vec{z}_c + \Delta\vec{z}^f = f\left(\vec{z}_c + \Delta\vec{z}\right) \tag{8}$$

Then, *in theory*, we can expand in power of $\Delta\vec{z}$:

$$\Delta\vec{z}^f = M\Delta\vec{z} + T^2\Delta\vec{z}\Delta\vec{z} + T^3\Delta\vec{z}\Delta\vec{z}\Delta\vec{z}\cdots \tag{9}$$
$$\text{where} \quad M_{ij} = \left.\frac{\partial f_i}{\partial z_j}\right|_{\vec{z}_c}$$

Can we get M, T^2, etc...accurately (to machine precision) and effortlessly?

Yes!

In our field, thanks to the work of Martin Berz.

Example: pendulum integrator
The usual programme for real numbers

cern_cas/mini_tpsa/mini_package

```
program z_pendulum0
use my_own_da
implicit none
! dp  means double precision
! pi is 3.1415.... etc
real(dp) :: z0(2),zf(2),omega,dt,freq,k=0

freq=0.12_dp;   dt=0.1_dp; omega=2*pi*freq;

z0=0
zf=f(z0)

call print(zf(1))
call print(zf(2))

contains

function f(z)
real(dp) f(2),z(2)
   f(1)=z(1)+dt*z(2)
   f(2)=z(2)-dt*omega**2*sin(f(1))
end function f
```

Notice that $z = (0, 0)$ is the fixed point (closed orbit) for the pendulum. How to get the matrix expansion around that point? By having only that code?
Answer: with TPSA you only have to replace real numbers by some special numbers called my_taylor in my own programme!!!

TPSA is the answer

1. The creator of TPSA takes a language (say Fortran90, C++, etc. . .) which allows *operator overloading* which means redefinition of existing operations
2. The creator of TPSA creates a new object, say my_taylor, which represents a Taylor series in that computer language
3. The creator of TPSA extends all the operations (+,-,x,/,=) and the intrinsic functions of the language to deal with Taylor series through operator overloading
4. Then the user of TPSA makes sure that his favourite code, say $f\left(\vec{z}\right)$, is converted to use this new Taylor type
5. Then this user of TPSA and any user of that favourite code can extract Taylor series around any orbit including the closed orbit
6. **Important: no knowledge of the internal physics is necessary. One could analyse a code with proprietary physics.**

Appetizer to be revisited

```
program z_pendulum1
use my_own_da
type(my_taylor) z(2),zf(2),omega,freq,dt
! dp  means double precision and  pi is 3.1415.... etc
real(dp) z0(2)

order_of_taylor=4 ; ! order of the Taylor series
z0=0
freq=0.12_dp  ; dt=0.1_dp  ; omega=2*pi*freq

z(1)=z0(1)+dx_1  ! dx_1 is a predefined  ``infinitesimal''
z(2)=z0(2)+dx_2  ! dx_2 is a predefined  ``infinitesimal''

zf=f(z)

call print(zf(1),title="zf(1) as an array")
call print(zf(2),title="zf(2) as an array")
call print_for_human(zf(1),title="zf(1) as a polynomial")
call print_for_human(zf(2),title="zf(2) as a polynomial")

contains

function f(z)
type(my_taylor) f(2),z(2)
f(1)=z(1)+dt*z(2)                  ⇐   represents a drift for example
f(2)=z(2)-dt*omega**2*sin(f(1))    ⇐   represents a thin RF cavity for example
end function f

end program z_pendulum1
```

Appetizer to be revisited
The pendulum code is converted to use TPSA

```
zf(1) as an array
    (1,0,0) 0.1000000000000E+01
    (0,1,0) 0.1000000000000E+00

zf(2) as an array
    (1,0,0)-0.5684892135027E-01
    (0,1,0) 0.9943151078650E+00
    (3,0,0) 0.9474820225046E-02
    (2,1,0) 0.2842446067514E-02
    (1,2,0) 0.2842446067514E-03
    (0,3,0) 0.9474820225046E-05
```

$$M = \begin{pmatrix} 0.1 & 0.1 \\ -5.684 & 0.9943 \end{pmatrix}$$

$$z_1^f = M_{11} z_1 + M_{12} z_2 + T_{130} z_1^3 + T_{121} z_1^2 z_2 + T_{112} z_1 z_2^2 + T_{103} z_2^3$$

$$z_2^f = M_{21} z_1 + M_{22} z_2 + T_{230} z_1^3 + T_{221} z_1^2 z_2 + T_{212} z_1 z_2^2 + T_{203} z_2^3$$

$$T_{230} = 0.00947 \qquad T_{221} = 0.00284 \qquad T_{212} = \frac{1}{10} T_{221} \qquad T_{203} = \frac{1}{1000} T_{230}$$

```
zf(1) as a polynomial
     0.1000000000000E+01   *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +   0.1000000000000E+00   *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0

zf(2) as a polynomial
    -0.5684892135027E-01   *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +   0.9943151078650E+00   *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0
 +   0.9474820225046E-02   *  dx_1 ^3  * dx_2 ^0  * dx_3 ^0
 +   0.2842446067514E-02   *  dx_1 ^2  * dx_2 ^1  * dx_3 ^0
 +   0.2842446067514E-03   *  dx_1 ^1  * dx_2 ^2  * dx_3 ^0
 +   0.9474820225046E-05   *  dx_1 ^0  * dx_2 ^3  * dx_3 ^0
```

How does this TPSA work?

Computing an exponential of two variables

```
program z_example0
use my_own_da
real(dp)x0,y0,z


! input
x0=1; y0=-1;

z=(x0+y0)**2
write(6,*) "z=(x0+y0)**2 = ", z
z=exp(z)
write(6,*) "exp(z) = ",z

end program z_example0
```

This trivial programme computes a quadratic polynomial $z = (x0 + y0)^2$ and its exponential. The results are clearly 0 and 1. Let us see how TPSA can allow us, effortlessly, to expand these functions in terms of *dx* and *dy*. Let us work it out first by “hand”.

“Analytical calculation”

$$\begin{aligned} z &= \Big(\underbrace{x_0 + dx}_{x} + \underbrace{y_0 + dy}_{y}\Big)^2 \\ &= (x_0+y_0)^2 + 2\,(x_0+y_0)\,(dx+dy) + (dx+dy)^2 \end{aligned} \quad (10)$$

$$\begin{aligned} \exp(z) &= E_0 \left\{ 1 + 2\,(x_0+y_0)\,(dx+dy) + \left(2(x_0+y_0)^2 + 1\right)(dx+dy)^2 \right\} \\ \text{where} \quad E_0 &= \exp\left((x_0+y_0)^2\right) \end{aligned} \quad (11)$$

Example code of TPSA: Module my_own_da

We run this code

```
program z_example
use my_own_da
real(dp)x0,y0,E_0
type(my_taylor) x,y,z

order_of_taylor=2  !  Order of Taylor Series
x0=1; y0=-1;

x=x0+dx_1 ; call print_for_human(x,6,"Variable x" )
y=y0+dx_2 ;call print_for_human(y,6,"Variable y")

z=(x+y)**2
call print_for_human(z,6,"Variable (x+y)**2 ")
z=exp(z)
call print_for_human(z,6,"Variable exp((x+y)**2) ")

!!! looking inside z
write(6,*) " Internal storage "," index,  exponents(1..3)  , complex value "
do i=0,size(z%a)-1
if(abs(z%a(i))/=0) write(6,'(20x,4(i2,4x),2(E20.13,1x))')i,jexp1(i),jexp1(2),jexp1(3),z%a(i)
enddo
!  Checking analytic calculations
E_0=exp((x0+y0)**2)
write(6,*) "E_0= exp((x0+y0)**2)",E_0
write(6,*) "dx, dy  coefficient ", E_0*(2*(x0+y0))
write(6,*) "dx**2, dy**2  coefficient ", E_0*(2*(x0+y0)**2+1)
write(6,*) "dx*dy  coefficient ", E_0*2*(2*(x0+y0)**2+1)
end program z_example
```

$\longleftarrow$ dx_1 is an infinitesimal of the first variable

$\longleftarrow$ dx_2 is an infinitesimal of the second variable

Running z_example

```
Variable x
    0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^0  * dx_3 ^0
 +  0.1000000000000E+01  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
```

$\longleftarrow$ 1 + dx_1

```
Variable y
   -0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^0  * dx_3 ^0
 +  0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0
```

$\longleftarrow$ -1 + dx_2

```
Variable (x+y)**2
    0.1000000000000E+01  *  dx_1 ^2  * dx_2 ^0  * dx_3 ^0
 +  0.2000000000000E+01  *  dx_1 ^1  * dx_2 ^1  * dx_3 ^0
 +  0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^2  * dx_3 ^0
```

$\longleftarrow$ ((1 + dx_1) + (-1 + dx_2))^2

$= \text{dx\_1}^2 + \text{dx\_2}^2 + 2\text{dx\_1dx\_2}$

```
Variable exp((x+y)**2)
    0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^0  * dx_3 ^0
 +  0.1000000000000E+01  *  dx_1 ^2  * dx_2 ^0  * dx_3 ^0
 +  0.2000000000000E+01  *  dx_1 ^1  * dx_2 ^1  * dx_3 ^0
 +  0.1000000000000E+01  *  dx_1 ^0  * dx_2 ^2  * dx_3 ^0
```

$\longleftarrow = 1 + (\text{dx\_1} + \text{dx\_2})^2 + \cdots$

```
  Internal storage  index,  exponents(1..3)  , complex value
                     0     0     0     0     0.1000000000000E+01  0.0000000000000E+00
                     4     2     0     0     0.1000000000000E+01  0.0000000000000E+00
                     5     1     0     0     0.2000000000000E+01  0.0000000000000E+00
                     6     0     0     0     0.1000000000000E+01  0.0000000000000E+00
 E_0= exp((x0+y0)**2)    1.00000000000000
 dx, dy  coefficient   0.000000000000000E+000
 dx**2, dy**2  coefficient    1.00000000000000
 dx*dy  coefficient    2.00000000000000
```

How does it work?

overloading =

Definition of my_taylor:

```
 TYPE my_taylor
    complex(dp) a(0:n_mono)
END TYPE my_taylor
```

$\Longleftarrow$ my_taylor is just a collection of polynomial coefficients

The array a(0:n_mono) contains the monomials of the Taylor series

```
subroutine input_my_taylor_in_my_taylor( s2, s1 )
    implicit none
    type (my_taylor), intent (in) :: s1
    type (my_taylor), intent (inout) :: s2
     s2%a=s1%a
 end subroutine input_my_taylor_in_my_taylor
```

$\Longleftarrow$ Taylor = Taylor

```
subroutine input_my_taylor_in_real( s2, s1 )
    implicit none
    type (my_taylor), intent (in) :: s1
    real(dp), intent (inout) :: s2
     s2=s1%a(0)
end subroutine input_my_taylor_in_real
```

$\Longleftarrow$ Taylor = real(8)

How does it work?

Definition of Taylor Series: 4th degree one

Definition of my_taylor:

```
TYPE my_taylor
    complex(dp) a(0:n_mono)
END TYPE my_taylor
```

$\Longleftarrow$ my_taylor is just a collection of polynomial coefficients

```
    i      j1(i) j2(i) j3(i)        i      j1(i) j2(i) j3(i)        i      j1(i) j2(i) j3(i)
 #  0       0     0     0        #  1       1     0     0        #  2       0     1     0
 #  3       0     0     1        #  4       2     0     0        #  5       1     1     0
 #  6       0     2     0        #  7       1     0     1        #  8       0     1     1
 #  9       0     0     2        # 10       3     0     0        # 11       2     1     0
 # 12       1     2     0        # 13       0     3     0        # 14       2     0     1
 # 15       1     1     1        # 16       0     2     1        # 17       1     0     2
 # 18       0     1     2        # 19       0     0     3        # 20       4     0     0
 # 21       3     1     0        # 22       2     2     0        # 23       1     3     0
 # 24       0     4     0        # 25       3     0     1        # 26       2     1     1
 # 27       1     2     1        # 28       0     3     1        # 29       2     0     2
 # 30       1     1     2        # 31       0     2     2        # 32       1     0     3
```

The array a(0:n_mono) contains the monomials of a Taylor series *t*

$$t \quad = \sum_{i=0,n\_mono} t\%a(i)\; x_1^{j1(i)}\, x_2^{j2(i)}\, x_3^{j3(i)} \qquad (12)$$

How does it work?
Actual function ADD(S1,S2) overloading +

So addition and subtraction are easy, for example, the function "add" overloads (+):

```
FUNCTION ADD( S1, S2 )
IMPLICIT NONE
TYPE (MY_TAYLOR) ADD
TYPE (MY_TAYLOR), INTENT (IN) :: S1, S2

ADD%A=S1%A + S2%A

END FUNCTION ADD
```

$$\begin{aligned} s_1 + s_2 &= \sum_{i=0,n\_mono} s_1\%a(i)\ x_1^{j1(i)} x_2^{j2(i)} x_3^{j3(i)} + \sum_{i=0,n\_mono} s_2\%a(i)\ x_1^{j1(i)} x_2^{j2(i)} x_3^{j3(i)} \\ &= \sum_{i=0,n\_mono} \{s_1\%a(i) + s_2\%a(i)\}\ x_1^{j1(i)} x_2^{j2(i)} x_3^{j3(i)} \end{aligned} \tag{13}$$

Multiplication
Actual function MUL(S1,S2) overloading *

Definition of my_taylor:

```
TYPE MY_TAYLOR
 COMPLEX(DP) A(0:N_MONO)
END TYPE MY_TAYLOR


FUNCTION MUL( S1, S2 )
IMPLICIT NONE
TYPE (MY_TAYLOR) MUL
TYPE (MY_TAYLOR), INTENT (IN) :: S1, S2
INTEGER I,J,K
MUL%A=0.0_DP
DO I=0,N_MONO
DO J=0,N_MONO
K=MUL_TABLE(I,J)
IF(K==-1.OR.JORDER(K)>MY_ORDER) CYCLE
MUL%A(K)=S1%A(I)*S2%A(J)+MUL%A(K)
ENDDO
ENDDO
CALL CLEAN(MUL)

END FUNCTION MUL
```

Multiplication: the only “hard” part’
Programme z_mul.90

Powers of the first and second variable

```
Multiplication table for my_order =           2

        x%a(i) * y%a(j) =  z%a(k)

         i        j        k

         0        0        0     ->     0    0    *    0    0    =    0    0
         0        1        1     ->     0    0    *    1    0    =    1    0
         0        2        2     ->     0    0    *    0    1    =    0    1
         0        4        4     ->     0    0    *    2    0    =    2    0
         0        5        5     ->     0    0    *    1    1    =    1    1
         0        6        6     ->     0    0    *    0    2    =    0    2
         1        0        1     ->     1    0    *    0    0    =    1    0
         1        1        4     ->     1    0    *    1    0    =    2    0
         1        2        5     ->     1    0    *    0    1    =    1    1
         2        0        2     ->     0    1    *    0    0    =    0    1
         2        1        5     ->     0    1    *    1    0    =    1    1
         2        2        6     ->     0    1    *    0    1    =    0    2
         4        0        4     ->     2    0    *    0    0    =    2    0
         5        0        5     ->     1    1    *    0    0    =    1    1
         6        0        6     ->     0    2    *    0    0    =    0    2
variable x
         0 (1.00000000000000,0.000000000000000E+000)          0           0
         1 (1.00000000000000,0.000000000000000E+000)          1           0
variable y
         0 (-1.00000000000000,0.000000000000000E+000)          0           0
         2 (1.00000000000000,0.000000000000000E+000)          0           1
variable z=x*y
         0 (-1.00000000000000,0.000000000000000E+000)          0           0
         1 (-1.00000000000000,0.000000000000000E+000)          1           0
         2 (1.00000000000000,0.000000000000000E+000)          0           1
         5 (1.00000000000000,0.000000000000000E+000)          1           1
```

$$x * y = (1 + dx_1) * (-1 + dx_2)$$

$$= (1 * -1) + (-1 * dx_1) + 1 * dx_2 + dx_1 * dx_2$$

Division:
first the inv(x) function

To perform a division, we invoke an inversion function:

$$inv\left(x\right) = \frac{1}{x} = \frac{1}{x_0 + \Delta x} = \frac{1}{x_0}\frac{1}{1+\frac{\Delta x}{x_0}}$$

$$= \frac{1}{x_0}\left(1 - \frac{\Delta x}{x_0} + \left(\frac{\Delta x}{x_0}\right)^2 - \left(\frac{\Delta x}{x_0}\right)^3 + \cdots\right) \tag{14}$$

The power series is carried to the order of truncation.

The general trick is to isolate the “constant” part, say $1/x_0$

```
FUNCTION INV( S1 )
IMPLICIT NONE
TYPE (MY_TAYLOR) INV,T,TT
TYPE (MY_TAYLOR), INTENT (IN) :: S1
INTEGER I

T=S1/S1%A(0)
T%A(0)=0.D0
INV=1.0_DP
TT=1.0_DP
DO I=1,MY_ORDER
TT=-T*TT
INV=INV+TT
ENDDO

INV=INV/S1%A(0)
CALL CLEAN(INV)

END FUNCTION INV
```

Notice that all the operations involved are the basic (+,-,*,/)

Division:

Actual function DIV(S1,S2) overloading /

The function DIV which overloads / uses INV and plain multiplication overloaded by MUL

```
FUNCTION DIV( S1, S2 )
IMPLICIT NONE
TYPE (MY_TAYLOR) DIV
TYPE (MY_TAYLOR), INTENT (IN) :: S1, S2
DIV=INV(S2)
DIV=S1*DIV
CALL CLEAN(DIV)
END FUNCTION DIV
```

Example of function: Exp(x)
Easiest one to overload: programme z_exp.f90

Again the general trick is to isolate the “constant” part, say $\exp(x_0)$

$$\begin{aligned}\exp(x) &= \exp(x_0+\Delta x) = \exp(x_0)\exp(\Delta x)\\ &= \exp(x_0)\left(1+\Delta x+\frac{1}{2!}\Delta x^2+\frac{1}{3!}\Delta x^3+\cdots\right)\end{aligned} \tag{15}$$

```
FUNCTION DEXPT( S1 )
IMPLICIT NONE
TYPE (MY_TAYLOR) DEXPT,T,TT
TYPE (MY_TAYLOR), INTENT (IN) :: S1
INTEGER I,NO
T=S1
T%A(0)=0.0_DP

DEXPT=1.0_DP
TT=1.0_DP

DO I=1,MY_ORDER
 TT=TT*T/I
 DEXPT=DEXPT + TT
ENDDO

DEXPT=EXP(S1%A(0))*DEXPT
CALL CLEAN(DEXPT)

END FUNCTION DEXPT
```

A bit harder: atan(x)

Thank heaven for complex numbers : programme z_atan_tan.90

$$\tan^{-1}(x) = -i \log\left(\sqrt{\frac{1}{1+x^2}}\,(1+i\,x)\right) \tag{16}$$

```
FUNCTION ATANT(S1)
  IMPLICIT NONE
  TYPE (MY_TAYLOR) ATANT,C
  TYPE (MY_TAYLOR), INTENT (IN) :: S1
   C=SQRT(1.0_DP/(1.0_DP+S1**2))
   C=C+I_*S1*C  ! COS(ATANT)+I * SIN(ATANT)
   ATANT=LOG(C)/I_
END FUNCTION  ATANT
```

Notice that we need the square root! But to get the square root we can use the logarithm and the exponential. So indeed we have the logarithm:

```
FUNCTION DLOGT( S1 )
  implicit none
  TYPE (my_taylor) DLOGT,T,TT
  TYPE (my_taylor), INTENT (IN) :: S1
  INTEGER I
    T=S1/S1%A(0); T%A(0)=0.0_DP;
    DLOGT=0.0_dp
    TT=-1.0_DP
   DO I=1,MY_ORDER
    TT=-T*TT
    DLOGT=DLOGT+TT/i
   ENDDO
  DLOGT=DLOGT+ LOG(S1%A(0))
END FUNCTION  DLOGT
```

$$\begin{aligned}\log(x_0+\Delta x) &= \log(x_0)+\log\left(1+\frac{\Delta x}{x_0}\right)\\ &= \log(x_0)+\frac{\Delta x}{x_0}-\frac{1}{2}\left(\frac{\Delta x}{x_0}\right)^2+\frac{1}{3}\left(\frac{\Delta x}{x_0}\right)^3+\cdots\end{aligned}$$

The Pendulum Programme

z_pendulum1.f90

It is remarkable that the map for a step of integration of a pendulum is an approximate representation of a circular ring with a single RF cavity (ignoring the transverse planes.)

For some calculations, it is a sufficiently accurate model. No joke!

The Pendulum Programme

```
program z_pendulum1
use my_own_da
type(my_taylor) z(2),zf(2),omega,freq,dt
! dp  means double precision and  pi is 3.1415.... etc
real(dp) z0(2)

order_of_taylor=4 ; ! order of the Taylor series
z0=0
freq=0.12_dp  ; dt=0.1_dp  ; omega=2*pi*freq

z(1)=z0(1)+dx_1  ! dx_1 is a predefined  ``infinitesimal''
z(2)=z0(2)+dx_2  ! dx_2 is a predefined  ``infinitesimal''

zf=f(z)

call print(zf(1),title="zf(1) as an array")
call print(zf(2),title="zf(2) as an array")
call print_for_human(zf(1),title="zf(1) as a polynomial")
call print_for_human(zf(2),title="zf(2) as a polynomial")

contains

function f(z)
type(my_taylor) f(2),z(2)
f(1)=z(1)+dt*z(2)                ⟸   represents a drift for example
f(2)=z(2)-dt*omega**2*sin(f(1))    ⟸   represents a thin RF cavity for example
end function f

end program z_pendulum1
```

The Pendulum Programme
The results

```
zf(1) as an array
   (1,0,0) 0.1000000000000E+01
   (0,1,0) 0.1000000000000E+00

zf(2) as an array
   (1,0,0)-0.5684892135027E-01
   (0,1,0) 0.9943151078650E+00
   (3,0,0) 0.9474820225046E-02
   (2,1,0) 0.2842446067514E-02
   (1,2,0) 0.2842446067514E-03
   (0,3,0) 0.9474820225046E-05

zf(1) as a polynomial
    0.1000000000000E+01  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +  0.1000000000000E+00  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0

zf(2) as a polynomial
   -0.5684892135027E-01  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +  0.9943151078650E+00  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0
 +  0.9474820225046E-02  *  dx_1 ^3  * dx_2 ^0  * dx_3 ^0
 +  0.2842446067514E-02  *  dx_1 ^2  * dx_2 ^1  * dx_3 ^0
 +  0.2842446067514E-03  *  dx_1 ^1  * dx_2 ^2  * dx_3 ^0
 +  0.9474820225046E-05  *  dx_1 ^0  * dx_2 ^3  * dx_3 ^0
```

The Pendulum Programme: normalising

z_pendulum_map.f90

Normalizing the pendulum

```
                  .
                  .
                  .
zf=f(z)

call print(zf(1),title="zf(1) as an array")
call print(zf(2),title="zf(2) as an array")

call print_for_human(zf(1),title="zf(1) as a polynomial")
call print_for_human(zf(2),title="zf(2) as a polynomial")

one_period_map=zf

call print(one_period_map, title="Map of pendulum")

call normalise( one_period_map, normal  )
A=normal%a_t
rotation = A**(-1)*one_period_map*A
!!!!!!!!!!!!!    Normal form really worked  !!!!!!!!!!
call print_for_human(rotation,title="This should be a rotation")
diagonal=to_phasor*rotation*from_phasor
call print_for_human(diagonal,title="This should be diagonal")

call print_for_human(normal%total_tune,6,title="total tune")
```

$\Longleftarrow$ Make map into a rotation $R = A^{-1}MA$

$\Longleftarrow$ Make rotation into a diagonal map $\Lambda = C^{-1}RC$

$$R = \begin{pmatrix} \cos(\mu) & \sin(\mu) \\ -\sin(\mu) & \cos(\mu) \end{pmatrix}$$

$$\Lambda = \begin{pmatrix} \exp(-i\mu) & 0 \\ 0 & \exp(i\mu) \end{pmatrix}$$

$$C = \begin{pmatrix} 1/2 & 1/2 \\ -i/2 & i/2 \end{pmatrix}$$

$$C^{-1} = \begin{pmatrix} 1 & i \\ 1 & -i \end{pmatrix}$$

.

Normalizing the pendulum: module my_analysis

```
                .
                .
This should be a rotation
    0.9971575539325E+00  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +  0.7534462578964E-01  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0

  variable 2 of map

This should be a rotation
   -0.7534462578964E-01  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0
 +  0.9971575539325E+00  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0

  variable 1 of map

This should be diagonal
   ( 0.9971575539325E+00 + i -0.7534462578964E-01)  *  dx_1 ^1  * dx_2 ^0  * dx_3 ^0

  variable 2 of map

This should be diagonal
   ( 0.9971575539325E+00 + i  0.7534462578964E-01)  *  dx_1 ^0  * dx_2 ^1  * dx_3 ^0

total tune
    0.1200284426549E-01  *  dx_1 ^0  * dx_2 ^0  * dx_3 ^0

  Matrix reconstructed
   1.00000000000000       0.100000000000000
 -5.684892135027548E-002  0.994315107864972
```

A program with a model for magnets
Twiss Loop

z_my_nonlinear_twiss_phase_average_x.f90

A small tracking code: the magnet subroutine

```
subroutine track_magnet( r,mag )
implicit none
type (my_taylor) z(3),rad
type (ray) , intent (inout) :: r        ←  type ray
                                               type(my_taylor) z(3)
                                             end type ray
type (magnet) , intent (in) :: mag
integer n,i,j
real(dp) dl,fac
z=r%z
if(mag%l/=0) then
 dl=mag%l/mag%n
 do i=1,mag%n
  z(1)=z(1)+dl/2.d0*z(2)/(1.d0+z(3))  ←1/2 a drift in second order integrator
  fac=1.0_dp
  do j=0,nmul
   z(2)=z(2)-mag%bn(j)*dl*z(1)**(j)/fac ←Multipople kick
   fac=fac*(j+1)
  enddo
   z(2)=z(2)-mag%bn(0)*mag%h*dl*z(1)+(1.d0+z(3))*mag%h*dl←Terms of the sector bend Hamiltonian
                                                          in the small angle approximation
   z(1)=z(1)+dl/2.d0*z(2)/(1.d0+z(3))
  enddo
else                    1/2 a drift in second order integrator
 fac=1.0_dp
 do j=0,nmul
  z(2)=z(2)-mag%bn(j)*z(1)**(j)/fac   ←Multipople thin lens kick
  fac=fac*(j+1)
 enddo
endif
r%z=z
end subroutine track_magnet
```

The main programme
The type declarations

```
program small_code_twiss
use my_own_little_code_utilities
implicit none
type(my_taylor) z(2),x,phase_advance
type(my_taylor)  x2_average,x_average,x_average_xp
type(normalform) normal
type(my_map) m,id,a_cs,disp,a_l,a_nl,a_tracked  ←− ←− ←− ←−
real(dp) l,k1,fix(3),ma(3,3),h,dx_average_dj,betax,alphax
integer i,n,mf,j,i1,i2,k,nst
type(magnet) :: lattice(6*10)
type(ray) r      ←− ←− ←− ←−

! mad-x lattice
! l : drift, l= 1.0;
! qf : quadrupole, l=1.0, k1=1.0;
! qd : quadrupole, l=1.0, k1=-1.0;
! sf : sextupole, k2= 2.0;
!s1 : line= 10*(qf,sf,l,qd,sf,l);
```

```
type my_map
   type(my_taylor) v(2)
end type my_map
```

```
type ray
   type(my_taylor) z(3)
end type ray
```

```
type normalform
      type(my_map) a_t
      type(my_map) r
      type(my_map) disp
      type(my_map) a_l
      type(my_map) a_nl
      type(my_taylor) total_tune
      real(dp) tune,damping
      real(dp) dtune_dA
      real(dp) dtune_dk
end type normalform
```

The main programme: page 3
The normal form type

```
 type normalform
       type(my_map) a_t
       type(my_map) r
       type(my_map) disp
       type(my_map) a_l
       type(my_map) a_nl
       type(my_taylor) total_tune
       real(dp) tune,damping
       real(dp) dtune_dA
       real(dp) dtune_dk
 end type normalform
```

By calling the the subroutine

```
call normalise(m,normal)
```

where m is of type my_map and normal is type my_normal, we turn m into a rotation r. So the following code is true:

```
 normal%r = normal%a_t**(-1) *  normal%m * normal%a_t
```

The main programme: page 4
The lattice is an array here

```
write(6,*) " creating the simple lattice: 10*(qf,sf,l,qd,sf,l) "
nst=100
l=1.0_dp
h= twopi/20/l
do i=1,10
k=(i-1)*6
 lattice(1+k)%name="qf";lattice(1+k)%l=l;lattice(1+k)%bn=0.0_dp
 lattice(1+k)%bn(0)=h;lattice(1+k)%h=h;lattice(1+k)%bn(1)=1.0_dp;lattice(1+k)%n=nst
 lattice(2+k)%name="sf";lattice(2+k)%l=0.0_dp;lattice(2+k)%bn=0.0_dp
 lattice(2+k)%bn(2)=2.0_dp;lattice(2+k)%h=0;lattice(2+k)%n=0
 lattice(3+k)%name="l";lattice(3+k)%l=1.0_dp;lattice(3+k)%bn=0.0_dp
 lattice(3+k)%h=0;lattice(3+k)%n=1
 lattice(4+k)%name="qd";lattice(4+k)%l=l;lattice(4+k)%bn=0.0_dp
 lattice(4+k)%bn(1)=-1.0_dp;lattice(4+k)%bn(0)=h;lattice(4+k)%h=h
 lattice(4+k)%n=nst
 lattice(5+k)%name="sf";lattice(5+k)%l=0.0_dp;lattice(5+k)%bn=0.0_dp
 lattice(5+k)%bn(2)=2.0_dp;lattice(5+k)%h=0;lattice(5+k)%n=0
 lattice(6+k)%name="l";lattice(6+k)%l=1.0_dp;lattice(6+k)%bn=0.0_dp
 lattice(6+k)%h=0;lattice(6+k)%n=1
enddo
```

The main programme : page 5

Finding the one turn map and initialising a Twiss loop

```
mf=16
open(unit=mf,file="twiss.txt")

delta_is_3rd_parameter=.true.; epsclean=1.e-9

order_of_taylor=4

fix=0.0_dp;call find_closed_orbit( fix,lattice,1);

id=1
r=fix+id
```

$\longleftarrow$ adding the closed orbit to the identity map

```
call track_lattice( r,lattice,1,1)
m=r
```

$\longleftarrow$ Finding the one turn map "m"

```
call normalise(m,normal)
a_cs=normal%a_t
```

$\Big\} \longleftarrow m = a\_cs\; r\; a\_cs^{-1}$

```
x=dx_1+ fix(1);
call average(x,a_cs,x_average,x_average_xp,use_j=.true.)
```

$\Big\} \longleftarrow \langle x \rangle_{turns}$ is a special calculation

```
call print(x_average,title=" <x> from map normal form")

call canonise( normal%a_t ,a_cs )
r=fix+a_cs
```

$\longleftarrow$ Put a_cs in preferred form: Courant-Snyder for example

The main programme : page 6

Finding the one turn map and initialising a Twiss loop

Green text is a computation two lattice functions

Red text is a universal Twiss loop

```
dx_average_dj=0.0_dp;
phase_advance=0.0_dp;
betax=(a_cs%v(1).index.1)**2+(a_cs%v(1).index.2)**2
alphax=-((a_cs%v(1).index.1)*(a_cs%v(2).index.1)+(a_cs%v(1).index.2)*(a_cs%v(2).index.2))
write(mf,'(a3,5(1x,e20.13))') "ini",real(phase_advance%a(sub_index(0,0,0))), &
real(phase_advance%a(sub_index(0,0,1))),real(phase_advance%a(sub_index(2,0,0))),betax,alphax
do j=1,size(lattice)
 call track_magnet( r,lattice(j) )
 a_tracked=r
 fix=r
  call canonise( a_tracked ,a_cs,phase_advance0=phase_advance )
 r=fix+a_cs
betax=(a_cs%v(1).index.1)**2+(a_cs%v(1).index.2)**2
alphax=-((a_cs%v(1).index.1)*(a_cs%v(2).index.1)+(a_cs%v(1).index.2)*(a_cs%v(2).index.2))
write(mf,'(a3,5(1x,e20.13))') lattice(j)%name(1:3),real(phase_advance%a(sub_index(0,0,0))), &
real(phase_advance%a(sub_index(0,0,1))),real(phase_advance%a(sub_index(2,0,0))),betax,alphax
dx_average_dj=(betax)**1.5_dp*lattice(j)%bn(2)/4.0_dp &
*(-sin(phase_advance*twopi)+sin((phase_advance-normal%tune)*twopi)) &
/(1.0_dp-cos(normal%tune*twopi)) + dx_average_dj
enddo
```

$$\frac{\partial\langle\tilde{x}\rangle}{\partial J} = \frac{\beta_s^{1/2}}{2(1-\cos(\mu))}\oint_0^C\left(-\sin(\mu_{s\sigma})+\sin(\mu_{s\sigma}-\mu)\right)\beta_\sigma^{3/2}k_S(\sigma)d\sigma$$

```
dx_average_dj=dx_average_dj*sqrt(betax)
```

⋮