

# JuTrack User Manual

MSU–Beam Dynamics Group  
Jinyu Wan

September 25, 2025

## 1 Introduction

JuTrack is a Julia-based accelerator modeling package with nested automatic differentiation (AD) capabilities. By using the LLVM-based Enzyme AD plugin, JuTrack can obtain derivatives of essentially any simulation result with respect to any set of input parameters, all while maintaining high runtime performance. In addition, a fast dual-number Truncated Power Series Algebra (TPSA) method is also implemented to support AD in pure Julia environment. Unlike finite-difference (numerical) derivatives that approximate changes and can be plagued by step-size tradeoffs and numerical noise, AD evaluates exact derivatives through the chain rule, ensuring machine-precision accuracy. This enables new possibilities: fast sensitivity analysis of complex collective effects, efficient beam optics tuning via gradient descent, and integration with machine learning or control applications requiring gradient information.

Julia is a high-performance programming language designed for technical and scientific computing. It combines the speed of compiled languages with the flexibility of dynamic languages, making it an excellent choice for tasks ranging from data analysis to numerical simulations. One thing to note is that you might experience a slower startup due to just-in-time (JIT) compilation. This initial delay happens as Julia compiles your code for optimal performance on your system. However, subsequent executions will be much faster as the compiled code is reused.

## 2 Installation

JuTrack is written in pure Julia and requires Julia 1.10 or later (Julia 1.10.4 is recommended for best compatibility). Ensure Julia is installed on your system. You can download Julia from the official Julia website <https://julialang.org/downloads/oldreleases/>.

JuTrack is open source and can be downloaded from GitHub via:

```
1 git clone https://github.com/MSU-Beam-Dynamics/JuTrack.jl
```

Create Julia environment and install dependencies

```
1 cd JuTrack.jl
2 julia --project=. -e "using Pkg; Pkg.instantiate()"
```

## 3 Import JuTrack

Activate the JuTrack environment in the Julia code (change the path to your JuTrack directory accordingly):

```
1 using Pkg
2 Pkg.activate("/path/to/JuTrack.jl")
3 Pkg.instantiate()
```

Import JuTrack:

```
1 using JuTrack
```

Once installed, you can verify the setup by running a simple test:

```
1 using JuTrack
2 D = DRIFT(name="D1", len=1.0)           # a 1 m drift
3 Q = KQUAD(name="Q1", len=0.5, k1=0.2)   # a 0.5 m quadrupole
4 line = [D, Q, D]                       # simple beamline
5 beam = Beam([0.01 0.0 0.0 0.0 0.0 0.0], energy=1.0e9) # one electron with 1
    GeV energy
6 linepass!(line, beam)                  # track particle through line
7 println("Final coordinates: ", beam.r[1, :])
```

If the package is correctly installed, the above code will execute and print the final phase-space coordinates of the particle after the line (for example, `[0.000766, 0.000... 0.0, ...]`). This confirms that JuTrack is functioning. You are now ready to use JuTrack for accelerator modeling and particle tracking.

## 4 Multi-threading setup (optional)

JuTrack supports multi-threaded execution for tracking simulations. To utilize multiple CPU cores (e.g. for tracking many particles in parallel), set the environment variable `JULIA_NUM_THREADS` before starting Julia:

```
1 export JULIA_NUM_THREADS=<N>
```

(where  $N$  is the number of threads to use, typically the number of CPU cores). You can add this line to your shell startup script (e.g. `.bashrc` on Linux/macOS) to make it permanent. After launching Julia with this environment, JuTrack will automatically distribute particle tracking across threads when using the appropriate functions (see `plinepass!` and `pringpass!` in the documentation below). No other special configuration is required.

## 5 Precompile

Julia compiles functions the first time they are called (Just-In-Time, JIT). For complex code, this first call can take from seconds to minutes, and the cost repeats every time a new Julia process starts.

A custom sysimage precompiles those methods ahead of time, so loading and running functions has little to no latency, making Julia feel much closer to a traditional compiled language.

We provide ready-made helper scripts under the `precompile/` directory. The file `_agg.warmup.jl` includes some sample functions. Add the function calls you want Julia to precompile in this file, and remove any you don't need to reduce the size of the sysimage.

1) Test precompile statements. Move to the JuTrack.jl folder, and type

Windows (PowerShell/CMD):

```
1 julia --project=. --trace-compile=precompile\trace_base.jl precompile\
    run_warmup.jl
```

macOS/Linux:

```
1 julia --project=. --trace-compile=precompile/trace_base.jl precompile/
    run_warmup.jl
```

2) Build the sysimage

Windows:

```
1 julia --project=. precompile\build_sysimage.jl
```

macOS/Linux:

```
1 julia --project=. precompile/build_sysimage.jl
```

This creates precompile/JuTrack\_sysimage.(dll—dylib—so) for your platform.

3) Use it

Windows:

```
1 julia --sysimage=precompile\JuTrack_sysimage.dll your_code.jl
```

macOS:

```
1 julia --sysimage=precompile/JuTrack_sysimage.dylib your_code.jl
```

Linux:

```
1 julia --sysimage=precompile/JuTrack_sysimage.so your_code.jl
```

## 6 Coordinate system

JuTrack tracks particles in 6-D space based on the design trajectory of an accelerator,

$$\begin{array}{ll} x & [\text{m}] \\ \frac{p_x}{p_0} & \\ y & [\text{m}] \\ \frac{p_y}{p_0} & \\ z = -\beta c \tau & [\text{m}] \\ \delta = \frac{p - p_0}{p_0} & \end{array}$$

where  $z$  is the path lengthening with respect to the reference particle and  $\tau$  is the arrival time compared to the reference particle. Note the sign of  $z$  may not align with other packages, e.g., it is opposite to MAD-X.

## 7 Exact path lengthening and linear approximation

Based on Ref. [Forest(2018)],  $p_z/p_0 = \sqrt{(1 + \delta)^2 - (p_x/p_0)^2 - (p_y/p_0)^2}$ . In some accelerator modeling codes such as [DriftPass](#) in [AT](#), this is approximated as  $p_z/p_0 = (1 + \delta)$ . In JuTrack, we use exact  $p_z$  by default. User can manually switch it to linear approximation before tracking by using

```
1 ues_exact_dirft(0)
```

To use the exact  $p_z$  again, run

```
1 ues_exact_dirft(1)
```

## 8 Function Documentation

JuTrack provides a range of types and functions to define accelerator elements, create beam objects, perform particle tracking (with and without space-charge effects), compute optical functions, and obtain derivatives via automatic differentiation. This section documents each major function in the package. Most functions in JuTrack use keyword arguments, e.g., `k1` and `len` for a quadrupole, and every keyword argument has a default value. All keyword arguments are optional and users can choose any of them to override defaults without worrying about position, providing great flexibility in coding.

## 8.1 Lattice Element Definitions

Accelerator elements (drifts, magnets, cavities, etc.) in JuTrack are represented as Julia structs with physical properties as fields. Each element has a constructor function to create an instance. Elements can be placed in a sequence (Julia Vector) to form a beamline or ring.

### 8.1.1 DRIFT

Create a drift space.

```
1 DRIFT(; name::String="DRIFT", len::Float64=0.0, [transform arguments])
```

- name:	Name of the element	
- len:	Length of the drift space	[m]
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts at entrance/exit	[m]
R1, R2 (6×6 matrices)	Rotation transformations at entrance/exit	

Example:

```
1 D1 = DRIFT(name="D1", len=2.0)
```

### 8.1.2 QUAD

Creates a quadrupole magnet using first-order matrix map. For canonical element, please use KQUAD.

```
1 QUAD(; name::String = "Quad", len::Float64 = 0.0, k1::Float64 = 0.0, [transform args])
```

- name:	Name of the element	
- len:	Length of the drift space	[m]
- k1:	quadrupole strength	[m <sup>-2</sup> ]
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts at entrance/exit	[m]
R1, R2 (6×6 matrices)	Rotation transformations at entrance/exit	

Example:

```
1 Q1 = QUAD(name="Q1", len=1.0, k1=1.0)
```

### 8.1.3 KQUAD

Creates a quadrupole magnet using fourth-order symplectic integrator.

```
1 KQUAD(; name::String="Quad", len::Float64=0.0, k1::Float64=0.0,
2         NumIntSteps::Int64=10, rad::Int64=0, [transform args])
```

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k1:	Quadrupole strength	[m <sup>-2</sup> ]
- NumIntSteps:	Number of integration slices	
- rad:	Synchrotron radiation flag. 0=off and 1=on (default 0)	
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	
PolynomA/B	Higher-order multipole coefficients	

Example:

```
1 Q1 = KQUAD(name="Q1", len=0.5, k1=1.5) # 0.5 m quadrupole with k1=1.5 m^-2
```

### 8.1.4 KSEXT

Creates a sextupole magnet using fourth-order symplectic integrator.

```
1 KSEXT(; name::String="Sext", len::Float64=0.0, k2::Float64=0.0,  
2       NumIntSteps::Int64=10, rad::Int64=0, [transform args])
```

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k2:	Sextupole strength	[m <sup>-3</sup> ]
- NumIntSteps:	Number of integration slices	
- rad:	Synchrotron radiation flag (0=off)	
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	
PolynomA/B	Higher-order multipole coefficients	

Example:

```
1 S1 = KSEXT(name="S1", len=0.5, k2=5.0) # 0.5 m sextupole with k2=5.0 m-3
```

### 8.1.5 KOCT

Creates an octupole magnet using fourth-order symplectic integrator.

```
1 KOCT(; name::String="Oct", len::Float64=0.0, k3::Float64=0.0,  
2       NumIntSteps::Int64=10, rad::Int64=0, [transform args])
```

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k3:	Octupole strength	[m <sup>-4</sup> ]
- NumIntSteps:	Number of integration slices	
- rad:	Synchrotron radiation flag (0=off)	
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	
PolynomA/B	Higher-order multipole coefficients	

Example:

```
1 O1 = KOCT(name="O1", len=0.5, k3=100.0) # 0.5 m octupole with k3=100 m-4
```

### 8.1.6 SBEND

Creates a sector dipole bending magnet. PolynomB[1] represent the dipole field error and the field related to the bending angle should not be included in this term.

```
1 SBEND(; name::String="BEND", len::Float64=0.0, angle::Float64=0.0,  
2       e1::Float64=0.0, e2::Float64=0.0, PolynomB::Vector{Float64}=[0.0, 0.0,  
       0.0, 0.0], [transform args])
```

- name:	Name of the element	
- len:	Arc length of the dipole	[m]
- angle:	Total bending angle	[rad]
- e1:	Entrance angle	[rad]
- e2:	Exit angle	[rad]
- Optional arrays:		
PolynomB	Field expansion coefficients	[m <sup>-1</sup> , m <sup>-2</sup> , m <sup>-3</sup> , m <sup>-4</sup> ]
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	

Example:

```
1 B1 = SBEND(name="B1", len=1.2, angle=0.15, e1=0.075, e2=0.075)
```

### 8.1.7 RBEND

Creates a rectangular dipole bending magnet. The length of the RBEND is the arc length in JuTrack. PolynomB[1] represent the dipole field error and the field related to the bending angle should not be included in this term.

```
1 RBEND(; name::String="BEND", len::Float64=0.0, angle::Float64=0.0,  
2       PolynomB::Vector{Float64}=[0.0, 0.0, 0.0, 0.0], [transform args])
```

- name:	Name of the element	
- len:	Length of the dipole	[m]
- angle:	Total bending angle	[rad]
- Optional arrays:		
PolynomB	Field expansion coefficients	$[m^{-1}, m^{-2}, m^{-3}, m^{-4}]$
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	

Example:

```
1 RB = RBEND(name="RB", len=1.0, angle=0.1)
```

### 8.1.8 ESBEND

Creates a sector dipole bending magnet using exact integrator. PolynomB[1] represent the dipole field error and the field related to the bending angle should not be included in this term.

```
1 ESBEND(; name::String="EBEND", len::Float64=0.0, angle::Float64=0.0,  
2       e1::Float64=0.0, e2::Float64=0.0, PolynomB::Vector{Float64}=[0.0, 0.0,  
       0.0, 0.0], [transform args])
```

- name:	Name of the element	
- len:	Arc length of the dipole	[m]
- angle:	Total bending angle	[rad]
- e1:	Entrance angle	[rad]
- e2:	Exit angle	[rad]
- Optional arrays:		
PolynomB	Field expansion coefficients	$[m^{-1}, m^{-2}, m^{-3}, m^{-4}]$
T1, T2 (6-element vectors)	Misalignment shifts	[m]
R1, R2 (6×6 matrices)	Rotation transformations	

Example:

```
1 B1 = ESBEND(name="B1", len=1.2, angle=0.15, e1=0.075, e2=0.075)
```

### 8.1.9 RFCA

Creates an simple RF accelerating cavity. The energy kick is implemented at the middle of the cavity.

```
1 RFCA(; name::String="RF", len::Float64=0.0, volt::Float64=0.0,  
2       freq::Float64=0.0, lag::Float64=0.0, philag::Float64=0.0,  
3       energy::Float64=E0)
```

- name: Name of the cavity
- len: Effective length [m]
- volt: Peak voltage [V]
- freq: RF frequency [Hz]
- lag: Position lag [m]
- philag: Phase lag [rad]
- energy: Reference energy [eV]

Example:

```
1 RFC = RFCA(name="RFC", volt=1e6, freq=400e6, energy=3e9)
```

### 8.1.10 CRABCAVITY

Creates a simple crab cavity. The energy kick is implemented at the middle of the cavity.

```
1 CRABCAVITY(; name::String="CRAB", len::Float64=0.0, volt::Float64=0.0,
2           freq::Float64=0.0, phi::Float64=0.0, energy::Float64=E0)
```

- name: Name of the cavity
- len: Effective length [m]
- volt: Peak voltage [V]
- freq: RF frequency [Hz]
- phi: Phase offset [rad]
- energy: Reference energy [eV]

Example:

```
1 CC = CRABCAVITY(name="CC", volt=3e6, freq=400e6, phi=pi/2, energy=3e9)
```

### 8.1.11 SOLENOID

Creates a solenoid implemented as a matrix.

```
1 SOLENOID(; name::String = "Solenoid", len::Float64 = 0.0, ks::Float64 = 0.0, [
           transform args])
```

- name: Name of the solenoid
- len: Effective length [m]
- ks: Solenoid strength [rad/m]
- Optional arrays:
  - T1, T2 (6-element vectors) Misalignment shifts [m]
  - R1, R2 (6×6 matrices) Rotation transformations

Example:

```
1 sol = SOLENOID(name="sol", len=4.0, ks=1.0)
```

### 8.1.12 CORRECTOR

Creates a corrector.

```
1 CORRECTOR(; name::String="COR", len::Float64=0.0,
2           xkick::Float64=0.0, ykick::Float64=0.0)
```

- name: Name of the corrector
- len: Length [m]
- xkick: Horizontal kick angle [rad]
- ykick: Vertical kick angle [rad]

Example:

```
1 HCOR = CORRECTOR(name="HCOR", xkick=1e-4) # 0.1 mrad horizontal kick
```

### 8.1.13 MARKER

Creates a marker element.

```
1 MARKER(; name::String="MARKER")
```

- name: Name of the marker

Example:

```
1 BPM1 = MARKER(name="BPM1")
```

### 8.1.14 StrongGaussianBeam

Create a strong Gaussian beam through beam-beam force induced by opposing beam at iteration point. The opposing beam is considered rigid.

```
1 StrongGaussianBeam(charge::Float64, mass::Float64, atomnum::Float64,
2                     np::Int, energy::Float64, op::AbstractOptics4D, bs::Vector{Float64},
                     nz::Int)
```

- charge:	Particle charge	[e]
- mass:	Particle mass	[eV]
- atomnum:	Particle atomic number	
- np:	Number of particles	
- energy:	Total beam energy	[eV]
- op:	4-D optics at IP. Implemented as a optics4DUC	
- bs:	Beam size at IP	[m]
- nz:	Number of slices in z direction.	

Example:

```
1 opIPp=optics4DUC(0.8, 0.0, 0.072, 0.0) # (betax, alphax, betay, alphay) at IP
2 sgb = StrongGaussianBeam(1.0, m_p, 1.0, Int(0.688e11), 275e9, opIPp, [95e-6,
    8.5e-6, 0.06], 9)
```

### 8.1.15 LongitudinalWake

Create a thin-length longitudinal wake field.

```
1 LongitudinalWake(times::AbstractVector, wakefields::AbstractVector, fliphalf::
    Float64=-1.0)
```

- times:	Time of the wakefields
- wakefields:	Wakefields values
- fliphalf	Flip flag

### 8.1.16 TRANSLATION

Implementation of the [MAD-X](#) element TRANSLATION. The element TRANSLATION changes the reference system by applying a translation of the reference system.

```
1 TRANSLATION(;name::String = "t1", dx::Float64 = 0.0, dy::Float64 = 0.0, ds::
    Float64 = 0.0)
```



- name: Name of the element
- dx: Translation in x [m]
- dy: Translation in y [m]
- ds: Translation in s [m]

Example:

```
1 t1 = TRANSLATION(name="t1", dx=1.0)
```

### 8.1.17 YROTATION

Implementation of the [MAD-X](#) element YROTATION. The element YROTATION rotates the straight reference system about the vertical (y) axis.

```
1 YROTATION(;name::String = "yrot1", angle::Float64 = 0.0)
```

- name: Name of the element
- angle: Rotation angle [rad]

Example:

```
1 yrot1 = YROTATION(name="yrot1", angle=0.05)
```

## 8.2 Space-charge enabled elements

JuTrack can include space-charge effects by using special element types with suffix `_SC`. So far, JuTrack only supports transverse space-charge effects based on [J. Qiang, PHYSICAL REVIEW ACCELERATORS AND BEAMS 20, 014203 (2017)], which follows a spectral Galerkin method for coasting beam in a rectangular conducting pipe. Essentially, the electric field of the beam is expanded in sine/cosine modes up to  $N_l$  and  $N_m$  in each plane, and a symplectic space-charge kick is applied. To use space-charge in a simulation, replace standard elements with their `_SC` counterparts and set beam parameters, e.g., current, energy and charge, appropriately. These correspond to the elements above but include a space-charge kick incorporated into the element's transfer map:

DRIFT\_SC, KQUAD\_SC, KSEXT\_SC, KOCT\_SC, SBEND\_SC, QUAD\_SC

These types take the same arguments as their base versions (DRIFT, KQUAD, etc.) plus additional parameters for the space-charge calculation: `a` and `b` (the horizontal and vertical half-size of the beam pipe, in [m]), `Nl` and `Nm` (the number of modes in the Fourier series expansion for space-charge field in x and y directions), and `Nsteps` (number of integration steps for space-charge kicks within the element).

Example:

```
1 Q_sc = KQUAD_SC(name="Qsc", len=0.1, k1=0.5, a=0.013, b=0.013, Nl=15, Nm=15,
    Nsteps=1)
```

defines a 0.1 m quadrupole with space-charge calculation assuming a 13 mm × 13 mm rectangular vacuum pipe, using 15×15 modes. Using `_SC` elements in the lattice will invoke space-charge kicks during tracking (the computational overhead scales with  $N_l \cdot N_m$  and the number of particles).

## 8.3 Beam Construction and Analysis

### 8.3.1 Beam Constructor

To simulate beam dynamics, one must define a Beam object that represents a collection of one or more particles with given initial conditions and beam parameters:

```
1 Beam(r::Matrix{Float64}, energy::Float64;
2     np::Int=size(r,1), charge::Float64=-1.0,
3     mass::Float64=9.11e5, current::Float64=0.0, ...)
```

- r: N×6 matrix of initial phase-space coordinates
- energy: Total energy [eV]
- charge: Particle charge state. -1.0 for electrons [e]
- mass: Particle rest mass [eV]
- current: Beam current (for space-charge calculation only) [A]

Without specific charge and mass, it will create an electron beam by default

```
1 particles = randn(1000, 6) .* [1e-3 1e-4 1e-3 1e-4 0.01 1e-4]
2 beam = Beam(particles, energy=1.0e9) # electron beam with energy of 1 GeV
```

Example of creating a proton beam:

```
1 particles = randn(1000, 6) .* [1e-3 1e-4 1e-3 1e-4 0.01 1e-4]
2 beam = Beam(particles, energy=1.0e9, charge=1.0, mass=m_p, current=20.0)
```

### 8.3.2 get\_emittance!

The beam emittance is not automatically updated when the beam distribution is changed. One has to use the following function to calculate current emittance:

```
1 get_emittance!(beam::Beam)
2 println(beam.emittance)
```

Computes: RMS emittances ( $\varepsilon_x, \varepsilon_y, \varepsilon_z$ )

**Updates** beam.emittance, beam.centroid, beam.beamsize

Formula:  $\varepsilon = \sqrt{\langle x^2 \rangle \langle p_x^2 \rangle - \langle xp_x \rangle^2}$

### 8.3.3 histogram1DinZ!

histogram1DinZ! calculate longitudinal histogram for beam-beam interaction effect calculation:

```
1 histogram1DinZ!(beam::Beam; nbins::Int=50)
```

**Computes** Longitudinal charge distribution

**Updates** beam.zhist (counts), beam.zhist.edges (bin edges)

## 8.4 Tracking Functions

**Note on Particle Loss:** Lost particles have their `lost_flag` set to 1 but remain in the beam array for post-analysis.

### 8.4.1 linepass!

Tracks beam through a sequence of elements.

```
1 linepass!(line, beam) # fast, no returns
2 rout = linepass!(line, beam, refpts) # slow, returns the trajectories.
```

**Inputs** line Vector of lattice elements

beam Initial beam

refpts (optional) A list of integers. Element indices to record state

**Updates** beam.r final coordinates

beam.lost\_flag particle loss flags

Example:

```
1 line = [D1, QF, D2, QD, D3]
2 beam = Beam(rand(1000,6)*1e-3, energy=3e9)
3 linepass!(line, beam)
```

### 8.4.2 ringpass!

Tracks beam through multiple turns in a circular lattice.

```

1 ringpass!(ring, beam, nturn; save::Bool=false)

```

<b>Inputs</b>	ring:	Ring lattice
	beam	Initial beam
	nturn	Number of turns
	save	Save turn-by-turn trajectories?
<b>Updates</b>	If true, returns rout = ringpass!(...), else, no returns	
	beam.r	Final coordinates
	beam.lost_flag	Particle loss flags

Example:

```

1 ringpass!(ring, beam, 1000) # Track for 1000 turns

```

### 8.4.3 Parallel Tracking

Multi-threaded version of linepass!/ringpass!. The implementation of them requires Julia starting with multiple threads. So far, the parallel code may conflict with auto-differentiation.

```

1 plinepass!(line, beam)
2 pringpass!(line, beam, nturn::Int)

```

## 8.5 High-order Multivariate Truncated Power Series Algebra (TPSA)

### 8.5.1 Multivariate TPSA

Creates a TPS variable for multivariate TPSA.

```

1 CTPS(a::T, n::Int, TPS_Dim::Int, Max_TPS_Degree::Int)

```

- a: Constant term value
- n: Variable index (E.g., 1-6 for 6-D phase space)
- TPS\_Dim: Total number of variables
- Max\_TPS\_Degree: Maximum order

Create a constant TPS variable for multivariate TPSA.

```

1 CTPS(a::T, TPS_Dim::Int, Max_TPS_Degree::Int)

```

- a: Constant term value
- TPS\_Dim: Total number of variables
- Max\_TPS\_Degree: Maximum order

Copy a TPS variable.

```

1 CTPS(ctps::CTPS)

```

Example (note that all TPS variables should have the same maximum order and total number of variables):

```

1 x = CTPS(0.0, 1, 6, 2) # x coordinate in 6-D space up to 2nd order
2 px = CTPS(0.0, 2, 6, 2) # Corresponding px variable
3 xpx = x*px # multiplication of x and px

```

### 8.5.2 Tracking using TPSA

A particle's coordinates can be represented as a TPS variable vector. For example, create a particle centered at [0.0 0.0 0.0 0.0 0.0 0.0],

```
1 x = CTPS(0.0, 1, 6, 3)
2 px = CTPS(0.0, 2, 6, 3)
3 y = CTPS(0.0, 3, 6, 3)
4 py = CTPS(0.0, 4, 6, 3)
5 z = CTPS(0.0, 5, 6, 3)
6 pz = CTPS(0.0, 6, 6, 3)
7 rin = [x, px, y, py, z, pz]
```

The 6-D TPS variable vector is trackable in JuTrack,

```
1 linepass_TPSA!(line::Vector, rin::Vector{CTPS})
2 ringpass_TPSA!(ring::Vector, rin::Vector{CTPS}, nturn::Int)
```

<b>Input</b>	line/ring	Lattice elements
	rin	Vector of 6 CTPS variables (x,px,y,py,z, $\delta$ )
	nturn	number of turns
<b>Updates</b>	rin with final TPS expressions	

## 8.6 Optics and Analysis Functions

### 8.6.1 EdwardsTengTwiss

Create an EdwardsTengTwiss variable. Twiss parameters include:

- $\beta_x, \alpha_x, \beta_y, \alpha_y$  (betatron functions)
- $\mu_x, \mu_y$  (phase advances)
- $D_x, D_{px}, D_y, D_{py}$  (dispersion)

```
1 twi = EdwardsTengTwiss(betax::Float64, betay::Float64;
2     alphax::Float64=0.0, alphay::Float64=0.0,
3     dx::Float64=0.0, dy::Float64=0.0,
4     dpx::Float64=0.0, dpy::Float64=0.0,
5     mux::Float64=0.0, muy::Float64=0.0)
```

<b>Inputs</b>	- betax [required]:	Horizontal beta function	[m]
	- betay [required]:	Vertical beta function	[m]
	- alphax:	Horizontal alpha function	
	- alphay:	Vertical alpha function	
	- dx:	Horizontal dispersion function	[m]
	- dy:	Vertical dispersion function	[m]
	- dpx:	Gradient of horizontal dispersion function	
	- dpy:	Gradient of vertical dispersion function	
	- mux:	Horizontal phase advance	[rad]
	- muy:	Vertical phase advance	[rad]
<b>Returns</b>	- twi:	Twiss parameter struct	

### 8.6.2 twissline

Propagate the Twiss parameters through a sequence of elements.

```
1 tout = (tin::EdwardsTengTwiss, seq::Vector, dp::Float64, order::Int, refpts::
2     Vector{Int};
3     E0::Float64=3e9, m0::Float64=m_e, orb::Vector{Float64}=zeros(6))
```

<b>Inputs</b>	- tin:	initial Twiss parameters	
	- seq:	lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order	
	- refpts:	List of element indices	
	- E0 [optional]:	Reference energy	[eV]
	- m0 [optional]:	Mass of the particle	[eV]
	- orb [optional]:	6-D Closed orbit	
<b>Returns</b>	- tout:	List of Twiss parameters at specific element positions	

### 8.6.3 periodicEdwardsTengTwiss

Calculate Twiss parameters for a periodic structure.

```

1 twi = periodicEdwardsTengTwiss(seq::Vector, dp::Float64, order::Int;
2                               E0::Float64=3e9, m0::Float64=m_e, orb::Vector{
                               Float64}=zeros(6))

```

<b>Inputs</b>	- seq:	A periodic lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order	
	- E0 [optional]:	Reference energy	[eV]
	- m0 [optional]:	Mass of the particle	[eV]
	- orb [optional]:	6-D Closed orbit	
<b>Returns</b>	- twi:	Twiss parameter struct	

### 8.6.4 twissring

Computes periodic Twiss parameters for a ring.

```

1 twi_list = twissring(lattice::Vector, dp::Float64, order::Int; E0::Float64=3e9,
2                      m0::Float64=m_e, orb::Vector{Float64}=zeros(6))

```

<b>Returns</b>	- lattice:	Closed lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order (0 or 1 is enough)	
	- E0 [optional]:	Reference energy	[eV]
	- m0 [optional]:	Mass of the particle	[eV]
	- orb [optional]:	6-D Closed orbit	
<b>Returns</b>	- twi_list:	List of Twiss parameters at each element position	

Example:

```

1 tw = twissring(ring, 0.0, 0) # On-momentum Twiss

```

### 8.6.5 fast\_find\_closed\_orbit\_4d

Find the 4-D closed orbit for a ring using numerical iteration. Avoid using this function in AD implementation due to the possible conflict with the linear algebra package.

```

1 orb4 = fast_closed_orbit_4d(ring::Vector; x0=zeros(4), energy::Float64=3.5e9,
                             mass::Float64=m_e)

```

<b>Returns</b>	- ring:	Closed lattice	
	- x0 [optional]:	Initial guess of the 4- D orbit	
	- energy [optional]:	Reference energy	[eV]
	- mass [optional]:	Mass of the particle	[eV]
<b>Returns</b>	- orb4:	4-D closed orbit	

Example:

```
1 orb4 = fast_find_closed_orbit_4d(ring, energy=275e9, mass=m_p)
```

### 8.6.6 fast\_find\_closed\_orbit\_6d

Find the 6-D closed orbit for a ring using numerical iteration. Ensure cavities are on before using this function. Avoid using this function in AD implementation due to the possible conflict with the linear algebra package.

```
1 orb6 = fast_closed_orbit_6d(ring::Vector; x0=zeros(6), energy::Float64=3.5e9,
    mass::Float64=m_e)
```

**Returns**

- ring: Closed lattice
- x0 [optional]: Initial guess of the 6- D orbit
- energy [optional]: Reference energy [eV]
- mass [optional]: Mass of the particle [eV]

**Returns**

- orb6: 6-D closed orbit

Example:

```
1 orb6 = fast_find_closed_orbit_6d(ring, energy=275e9, mass=m_p)
```

### 8.6.7 computeRDT

Computes resonance driving terms.

```
1 dtlist, s = computeRDT(ring::Vector, index::Vector{Int};
2     chromatic=false, coupling=false,
3     geometric1=false, geometric2=false,
4     tuneshifts=false)
```

**Inputs**

- ring: Lattice
- index: List of element indices to evaluate
- Flags: Select term categories

**Returns**

- dtlist: list of driving terms at specific locations
- s: locations of specific elements

Example of common RDTs:

- h21000
- h30000
- h10110
- h31000

Example:

```
1 d, s = computeRDT(ring, [12, 15], geometric1=true)
```

## 8.7 Convenient functions

### 8.7.1 findelem

findelem is a useful function to find elements with specific types, names or fields.

```
1 ids = findelem(ring::Vector, type::Type{<:AbstractElement})
```

**Inputs**

- ring: Lattice
- type: Element type

**Returns**

- ids: indices of elements matching type

```
1 ids = findelem(ring::Vector, field::Symbol, value)
```

**Inputs**    - ring:    Lattice  
              - field:    field name of the element  
              - value:    field value  
**Returns**   - ids:      indices of elements matching field values

### 8.7.2 plot\_lattice

plot\_lattice is a useful function to plot survey of the lattice. Only RF cavities, quadrupoles, sextupoles, octupoles, and dipoles will be plotted. Other elements will be ignored.

```
1 plot_lattice(line::Vector, scale=0.15, axis=true)
```

**Inputs**    - line:    Lattice  
              - scale:    Scale of the element size in the plot  
              - axis:    true or false: show or not show the axis

### 8.7.3 spos

```
1 s = spos(ring::Vector{AbstractElement}, [idx::Vector])
```

**Inputs**    - ring:    Lattice  
              - idx:    Indices of elements  
**Returns**   - s:       longitudinal positions of specific elements

```
1 spos(ring::Vector{AbstractElement})
```

**Inputs**    - ring:    Lattice  
**Returns**   - s:       longitudinal positions of all elements

Example:

```
1 quad_indices = findelem(ring, KQUAD)
2 quad_positions = spos(ring, quad_indices)
```

### 8.7.4 total\_length

Calculate the total length of the lattice.

```
1 totL = total_length(ring)
```

**Inputs**    - ring:    Lattice  
**Returns**   - length:   Total length

### 8.7.5 symplectic

Check if a transfer matrix is symplectic

```
1 max_delta = symplectic(M66::Array{Float64,2})
```

**Inputs**    - M66:        6\*6 transfer matrix  
**Returns**   - max\_delta:   Maximum deviation from a symplectic matrix

### 8.7.6 rad\_on, rad\_off

Turn on/off synchrotron radiation for all elements in the lattice.

```

1 rad_on!(ring)
2 rad_off!(ring)

```

**Inputs** - ring: Lattice

## 8.8 Beam dynamics analysis functions

### 8.8.1 dynamic\_aperture

A simple function to calculate dynamic aperture of a ring by scanning rays on multiple angles.

```

1 DA, particles = dynamic_aperture(ring, nturns, amp_max, step_size,
2                                angle_steps, E, dp)

```

**Inputs**

- ring: Lattice
- nturns: Number of turns
- amp\_max: Maximum scanning amplitude [m]
- step\_size: Step size of the scanning [m]
- angle\_steps: Number of rays in  $[0, \pi]$
- E: Beam energy [eV]
- dp: Momentum deviation

**Returns**

- DA: Envelop of DA
- particles: coordinates of particles on DA envelop

## 9 Automatic Differentiation (AD) with fast TPSA

A fast TPSA module **DTPSAD** is implemented for AD calculation in pure Julia. Note that this is not the same as the TPSA module in Section 7.5. This module is designed for fast AD and only supports first-order calculation. To implement this module, we need to define elements with **DTPSAD** variables instead of standard Float64 numbers. JuTrack support calculations with these elements and all the variables in the calculation are represented in a first-order series, therefore result in first-order derivatives for all parameters with respect to the TPS variables.

### 9.1 Set number of variables

```

1 set_tps_dim(n)

```

**Inputs** - n: Number of variables

### 9.2 Create a fast TPSA variable

```

1 x = DTPSAD(a)

```

**Inputs** - a: constant of variable

**Returns** - x: a fast TPS variable with constant a

```

1 x = DTPSAD(a, n)

```

**Inputs**

- a: constant of variable
- n: index of variable

**Returns** - x: a  $n$ -th fast TPS variable with constant a



### 9.3 Construct Beam element

Similar to `Beam()`,

```
1 beam = Beam(r::Matrix{DTPSAD{N, T}}; energy::Float64=1e9,  
2           np::Int=size(r,1), charge::Float64=-1.0,  
3           mass::Float64=9.11e5, current::Float64=0.0, ...) where {N, T}
```

<b>Inputs</b>	- r:	N×6 coordinate matrix with type of <b>DTPSAD</b>
	- other parameters	the same as normal Beam object
<b>Returns</b>	- beam:	a beam objective with all parameters described in TPSA

For example:

```
1 rin = rand(100, 6)  
2 beam = Beam(DTPSAD.(rin))
```

The difference is that all variables are represented by **DTPSAD** objects instead of the Float64 numbers, thereby the derivative information is reserved. When any of the parameters are **DTPSAD** objects, JuTrack will automatically create a **DTPSAD** element, i.e., `Beam{DTPSAD{N, T}}`, instead of `Beam{Float64}`.

### 9.4 Construct lattice element

Similar to standard lattice elements such as `DRIFT()`, `QUAD()`, `KSEXT()`, .... An example of creating an FODO cell in **DTPSAD** objects domain,

```
1 D1 = DRIFT(len=DTPSAD(0.2))  
2 D2 = DRIFT(len=DTPSAD(0.4))  
3 D3 = DRIFT(len=DTPSAD(0.2))  
4 Q1 = KQUAD(len=DTPSAD(0.1), k1=DTPSAD(-29.6, 1)) # k1 is the first variable to  
           differentiate  
5 Q2 = KQUAD(len=DTPSAD(0.1), k1=DTPSAD(29.6, 2)) # k1 is the second variable  
           to differentiate  
6 FODO = [D1, Q1, D2, Q2, D3]
```

The difference is that all variables are represented by **DTPSAD** objects instead of the Float64 numbers, thereby the derivative information is reserved. When any of the parameters are **DTPSAD** objects, JuTrack will automatically create a **DTPSAD** element, e.g., `DRIFT{DTPSAD{N, T}}`, instead of `DRIFT{Float64}`.

### 9.5 Particle tracking

The same as standard particle tracking, e.g.,

```
1 linepass!(FODO, beam) # use previously created FODO and beam  
2 ringpass!(FODO, beam, 1000)
```

### 9.6 Gradient

Obtain gradients of a single-output function **f**.

```
1 g = Gradient(f, X)
```

<b>Inputs</b>	- f:	Julia function $f(x_1, x_2, \dots, x_n)$ with $n$ arguments $x_1, x_2, \dots, x_n$
	- X:	A vector of the input arguments ( <b>DTPSAD</b> type)
<b>Returns</b>	- g:	gradients of $f$ with respect to $x_1, x_2, \dots, x_n$ at $X$

```
1 g, Y = Gradient(f, X, true)
```

Using example,

## 9.7 Jacobian

```
1 J = jacobian(f, X)
```

```
1 J, Y = Jacobian(f, X, true)
```

Using example,

Output is:

## 9.8 Number2TPSAD

```
1 line_TPSA = Number2TPSAD(line_standard)
```

18

## 9.9 TPSAD2Number

Convert a lattice with Telements to standard elements.

```
1 line_TPSA = Number2TPSAD(line_standard)
```

**Inputs** - line\_TPSA: A lattice sequence with **DTPSAD** elements  
such as DRIFT{DTPSAD{N, T}} and QUAD{DTPSAD{N, T}}.  
**Returns** - line\_standard: A lattice sequence with standard elements  
such as DRIFT{Float64} and QUAD{Float64}.

## 10 Automatic Differentiation (AD) with Enzyme

JuTrack utilizes AD capability of Enzyme.jl. For detailed information and more usage examples, please refer to <https://enzyme.mit.edu/julia/stable/>.

### 10.0.1 Core AD Functions

```
1 results = autodiff(mode, f, Duplicated(x, 1.0), ...args)
```

**Inputs** - mode: AD mode (only supports Forward or ForwardWithPrimal)  
- f: Differentiable function  
- Duplicated/Const: AD wrapper  
**Returns** - results: Derivatives or [derivatives, function values] for Primal mode

### 10.0.2 AD Wrappers

```
1 Duplicated(x, seed) # Marks variable for differentiation
2 Const(x)           # Marks constant parameter
```

Example (Sensitivity Analysis with space charge):

```
1 function emittance_vs_quad(k1)
2     # calculate emittance affected by space charge
3     Q = KQUAD_SC(k1=k1, a=0.013, b=0.013, Nl=15, Nm=15)
4     line = [Q, DRIFT(1.0)]
5     beam = Beam(rand(1000,6), 1e9)
6     ringpass!(line, beam, 10)
7     get_emittance!(beam)
8     return beam.emittance[1]
9 end
10
11 # Compute derivative at k1=2.0:
12 dE, E = autodiff(ForwardWithPrimal, emittance_vs_quad, Duplicated(2.0, 1.0))
```

## 10.1 jacobian

Compute the Jacobian of an array-output function f.

```
1 result = jacobian(mode, f, X)
```

**Inputs** - mode: AD mode (only supports Forward or ForwardWithPrimal)  
- f: Differentiable function  
- X: Array-like inputs of f  
**Returns** - results: Derivatives or [derivatives, function values] for Primal mode

Example:

```
1 function AD_jacobian(x)
2     beam = Beam([x[1] x[2] x[3] x[4] 0.0 0.0])
3     ringpass!(RING, beam, 5)
4     return beam.r
5 end
6 g = jacobian(Forward, AD_jacobian, [0.01, 0.0, 0.01, 0.0])
7 println("Jacobian matrix: ", g)
```

## References

[Forest(2018)] Étienne Forest, *Beam Dynamics*, CRC Press, 2018.