JuTrack User Manual

MSU–Beam Dynamics Group
Jinyu Wan

March 28, 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. 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 for offline installation: `https://github.com/MSU-Beam-Dynamics/JuTrack.jl`.

Online installation is also available:

```julia
import Pkg
Pkg.add(url="https://github.com/MSU-Beam-Dynamics/JuTrack.jl")
```

Install Enzyme. v0.13.3 is preferred:

```julia
Pkg.add(name="Enzyme", version="0.13.3")
```

After installation, load JuTrack in your Julia script or REPL:

```julia
using JuTrack
```

If any missing dependency is reported, just install them in your current Julia environment.

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

```julia
using JuTrack
D  = DRIFT(name="D1", len=1.0)              # a 1 m drift
```

```julia
Q  = KQUAD(name="Q1", len=0.5, k1=0.2)       # a 0.5 m quadrupole
line = [D, Q, D]                             # simple beamline
beam = Beam([0.01 0.0 0.0 0.0 0.0 0.0], energy=1.0e9) # one electron with 1
    GeV energy
linepass!(line, beam)                        # track particle through line
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.

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:

```bash
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.

3 Coordinate system

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

$$\begin{array}{ll} x & [\mathrm{m}] \\ p_x & [1] \\ y & [\mathrm{m}] \\ p_y & [1] \\ z = -\beta c \tau & [\mathrm{m}] \\ \delta = \frac{p-p_0}{p_0} & [1] \end{array}$$

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

4 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.

4.1 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.

4.1.1 DRIFT

Create a drift space.

```
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:

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

4.1.2 QUAD

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

```
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^{-2}]
- Optional arrays:		
T1, T2 (6-element vectors)	Misalignment shifts at entrance/exit	[m]
R1, R2 (6×6 matrices)	Rotation transformations at entrance/exit	

Example:

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

4.1.3 KQUAD

Creates a quadrupole magnet using fourth-order symplectic integrator.

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

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k1:	Quadrupole strength	[m^{-2}]
- 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:

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

4.1.4 KSEXT

Creates a sextupole magnet using fourth-order symplectic integrator.

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

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k2:	Sextupole strength	[m^{-3}]
- 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:

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

4.1.5 KOCT

Creates an octupole magnet using fourth-order symplectic integrator.

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

- name:	Name of the element	
- len:	Length of the magnet	[m]
- k3:	Octupole strength	[m^{-4}]
- 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:

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

4.1.6 SBEND

Creates a sector dipole bending magnet.

```
SBEND(; name::String="BEND", len::Float64=0.0, angle::Float64=0.0,
      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 bend angle	[rad]
- e1:	Entrance edge angle	[rad]
- e2:	Exit edge 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:

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

4.1.7 RBEND

Creates a rectangular dipole bending magnet.

```
RBEND(; name::String="BEND", len::Float64=0.0, angle::Float64=0.0,
      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 bend 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:

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

4.1.8 RFCA

Creates an RF accelerating cavity.

```
RFCA(; name::String="RF", len::Float64=0.0, volt::Float64=0.0,
     freq::Float64=0.0, h::Float64=1.0, lag::Float64=0.0, philag::Float64=0.0,
     energy::Float64=E0)
```

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

Example:

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

4.1.9 CRABCAVITY

Creates a crab cavity.

```
CRABCAVITY(; name::String="CRAB", len::Float64=0.0, volt::Float64=0.0,
           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:

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

4.1.10 CORRECTOR

Creates a corrector.

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

- name:	Name of the corrector	
- len:	Length (typically 0)	[m]
- xkick:	Horizontal kick angle	[rad]
- ykick:	Vertical kick angle	[rad]

Example:

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

4.1.11 MARKER

Creates a marker element.

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

- name: Name of the marker

Example:

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

4.2 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 Nl and Nm 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:

```
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 $\times$ 13 mm rectangular vacuum pipe, using 15$\times$15 modes. Using _SC elements in the lattice will invoke space-charge kicks during tracking (the computational overhead scales with Nl*Nm and the number of particles).

4.3 Beam Construction and Analysis

4.3.1 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:

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

- r:	N×6 matrix of initial phase-space coordinates	
- energy:	Reference particle kinetic 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]
- atomnum:	Atomic number (optional)	

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

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

Example of creating a proton beam:

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

4.3.2 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:

```
get_emittance!(beam::Beam)
prinln(beam.emittance)
```

Computes:	RMS emittances (ε_x, ε_y, ε_z)
Updates	beam.emittance, beam.centroid, beam.beamsize
Formula:	$\varepsilon = \sqrt{\langle x^2\rangle\langle p_x^2\rangle - \langle xp_x\rangle^2}$

4.3.3 histogram1DinZ!

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

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

Computes:	Longitudinal charge distribution
Updates	beam.zhist (counts), beam.zhist_edges (bin edges)

4.4 Tracking Functions

4.4.1 linepass!

Tracks beam through a sequence of elements.

```
linepass!(line, beam)
linepass!(line, beam, refpts)
```

Inputs	- line: Vector of lattice elements
	- beam: Initial beam
	- refpts: A list of integers. Optional element indices to record state
Updates	beam.r (final coordinates)
	beam.lost_flag (particle loss flags)

Example:

```
line = [D1, QF, D2, B1, D3, QD, D4]
beam = Beam(rand(1000,6)*1e-3, energy=3e9)
linepass!(line, beam)
```

4.4.2 ringpass!

Tracks beam through multiple turns in a circular lattice.

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

Inputs	- ring: Ring lattice
	- beam: Initial beam
	- nturn: Number of turns
	- save: Whether to save turn-by-turn data.
	If true, returns results: rout = ringpass!(...), else, no returns
Updates	beam.r (final coordinates)
	beam.lost_flag (particle loss flags)

Example:

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

4.4.3 Parallel Tracking

```
plinepass!(line, beam)
pringpass!(line, beam, nturn::Int)
```

Features:	- Multi-threaded version of linepass!/ringpass!
	- Requires Julia started with multiple threads
	- May conflict with auto-differentiation

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

4.5 Multivariate Truncated Power Series Algebra (TPSA)

4.5.1 Multivariate TPSA

Creates a TPS variable for multivariate TPSA.

```
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.

```
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 varaible.

```
CTPS(ctps::CTPS)
```

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

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

4.5.2 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],

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

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

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

Input	- line/ring: Lattice elements
	- rin: Vector of 6 CTPS variables (x,px,y,py,z,δ)
	- nturn: number of turns
Updates	rin with final TPS expressions

4.6 Optics and Analysis Functions

4.6.1 EdwardsTengTwiss

Create an EdwardsTengTwiss variable. Twiss parameters include:

- β_x, α_x, β_y, α_y (betatron functions)
- μ_x, μ_y (phase advances)
- D_x, D_{px}, D_y, D_{py} (dispersion)

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

Inputs	- 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]
Returns	- twi:	Twiss parameter struct	

4.6.2 twissline

Propagate the Twiss parameters through a sequence of elements.

```
tout = (tin::EdwardsTengTwiss,seq::Vector, dp::Float64, order::Int, refpts::
    Vector{Int})
```

Inputs	- tin:	initial Twiss parameters	
	- seq:	lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order (0 or 1 is enough)	
	- refpts:	List of element indices	
Returns	- tout:	List of Twiss parameter structs for specific element positions	

4.6.3 periodicEdwardsTengTwiss

Calculate Twiss parameters for a periodic structure.

```
twi = periodicEdwardsTengTwiss(seq::Vector, dp::Float64, order::Int)
```

Inputs	- seq:	A periodic lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order (0 or 1 is enough)	
Returns	- twi:	Twiss parameter struct	

4.6.4 twissring

Computes periodic Twiss parameters for a ring.

```
twi_list = twissring(lattice::Vector, dp::Float64, order::Int)
```

Returns	- lattice:	Closed lattice	
	- dp:	Momentum deviation	$[\delta p/p_0]$
	- order:	0=finite difference, > 0=TPSA order (0 or 1 is enough)	
Returns	- twi_list:	List of Twiss parameter structs for each element position	

Example:

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

4.6.5 computeRDT

Computes resonance driving terms.

```
dtlist, s = computeRDT(ring::Vector, index::Vector{Int};
            chromatic=false, coupling=false,
            geometric1=false, geometric2=false,
            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 (sextupole-driven)
- h30000 (sextupole-driven)
- h10110 (coupling)

- h31000 (octupole-driven)

Example:

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

4.7 Convenient functions

4.7.1 findelem

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

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

Inputs	- ring:	Lattice
	- type:	Element type
Returns	- ids:	indices of elements matching type

```
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

4.7.2 spos

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

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

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

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

Example:

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

4.8 Beam dynamics analysis functions

4.8.1 dynamic_aperture

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

```
DA, particles = dynamic_aperture(ring, nturns, amp_max, step_size,
                    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, π]	
	- E:	Beam energy	[eV]
	- dp:	Energy deviation	$\delta p/p_0$
Returns	- DA:	Envelop of DA	
	- particles:	coordinates of particles on DA envelop	

5 Automatic Differentiation (AD) Utilities

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

5.0.1 Core AD Functions

```
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

5.0.2 AD Wrappers

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

Example (Sensitivity Analysis with space charge):

```
function emittance_vs_quad(k1)
    # calculate emittance affected by space charge
    Q = KQUAD_SC(k1=k1, a=0.013, b=0.013, Nl=15, Nm=15)
    line = [Q, DRIFT(1.0)]
    beam = Beam(rand(1000,6), 1e9)
    ringpass!(line, beam, 10)
    get_emittance!(beam)
    return beam.emittance[1]
end

# Compute derivative at k1=2.0:
dE, E = autodiff(ForwardWithPrimal, emittance_vs_quad, Duplicated(2.0, 1.0))
```

5.1 jacobian

Compute the Jacobian of an array-output function f.

```
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:

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