JMathAnim DSL Quick Reference

Note:

- All parameter keys are case-insensitive. `style:` accepts either a Map or a String (style name).
- Parameters that expect coordinates admit both `Coordinates` objects (`Vec`, `Point`, etc.) or a list like `[3,2]`.

Basics

`shape(...)` — ShapeDSL

Type	Required params	Optional params
`circle`	—	`segments`
`square`	—	—
`rectangle`	`from:[x,y]`, `to:[x,y]`	—
`triangle`	—	—
`polygon`	`points:[[x,y],...]`	—
`polyline`	`points:[[x,y],...]`	—
`segment`	`from:[x,y]`, `to:[x,y]`	`numpoints`
`arc`	`angle` (radians)	`segments`
`regularPolygon`	`sides`	—
`regularInscribedPolygon`	`sides`	—
`annulus`	`minRadius`, `maxRadius`	—
`logo`	`commands` (String)	—

Common params (all shapes): `style`, `layer`, `visible`, `transform`, `stack`

```
def s = shape(type: "circle", segments: 6,
              style: [color: "blue", thickness: 2],
              transform: [scale: 1.5, shift: [1, 0]])

def r = shape(type: "segment", from: [0,0], to: [1,1])
def p = shape(type: "polygon", points: [[0,0],[1,0],[0.5,1]])
```

`stack:` map — StackDSL

Exactly one of `to`, `screen`, or `point` must be present.

Key	Type	Description
`to`	Boxable	Destination object
`screen`	String	`CENTER` `LEFT` `RIGHT` `UPPER` `LOWER` `UPPER_LEFT` `UPPER_RIGHT` `LOWER_LEFT` `LOWER_RIGHT`
`point`	`[x,y]`	Fixed point destination
`originAnchor`	String	AnchorType of moving object
`destinyAnchor`	String	AnchorType of destination
`gaps`	double / `[h,v]`	Absolute gap
`relativeGaps`	double / `[h,v]`	Relative gap (mutually exclusive with `gaps`)

```
stack: [to: otherShape, originAnchor: "left", destinyAnchor: "right", gaps: 0.1]
stack: [screen: "upper_left"]
stack: [point: [1, 2], destinyAnchor: "center"]
```

AnchorType values (common reference)

`CENTER` `LEFT` `RIGHT` `UPPER` `LOWER` `LEFT_AND_ALIGNED_UPPER` `LEFT_AND_ALIGNED_LOWER` `RIGHT_AND_ALIGNED_UPPER` `RIGHT_AND_ALIGNED_LOWER` `LOWER_AND_ALIGNED_LEFT` `LOWER_AND_ALIGNED_RIGHT` `UPPER_AND_ALIGNED_LEFT` `UPPER_AND_ALIGNED_RIGHT` `DIAG1` `DIAG2` `DIAG3` `DIAG4` `ZTOP` `ZBOTTOM`

All keyword String values support **prefix matching** (case-insensitive). E.g. `"cen"` matches `CENTER` in `AnchorType`

`transform:` map — BasicTransformDSL

Key	Type	Description
`scale`	double / `[sx,sy]`	Uniform or non-uniform scale
`shift`	`[x,y]` / Coords	Translation
`rotate`	double / `[center, angle]`	Rotation (radians), optionally around a point
`center`	`true`	Center the object on screen

```
transform: [scale: 2, shift: [1, 0], rotate: 45*DEGREES]
transform: [rotate: [[0,0], PI/3]]   // rotate around [0,0]
```

Colors & Styles

`style:` map — StyleDSL

Key	Type	Values / Notes
`name`	String	Named style template (loaded first)
`color`	String	Color name or hex
`drawColor`	String	Stroke color
`fillColor`	String	Fill color
`thickness`	double	Stroke width
`drawAlpha`	double	Stroke opacity `[0,1]`
`fillAlpha`	double	Fill opacity `[0,1]`
`visible`	boolean	—
`layer`	int	Rendering layer
`lineCap`	String	`SQUARE` `ROUND` `BUTT`
`lineJoin`	String	`BEVEL` `ROUND` `MITER`
`strokeType`	String	`OUTSIDE` `INSIDE` `CENTERED`
`dotStyle`	String	`CIRCLE` `CROSS` `PLUS` `RING` `TRIANGLE_UP_HOLLOW` `TRIANGLE_UP_FILLED` `TRIANGLE_DOWN_FILLED` `TRIANGLE_DOWN_HOLLOW`
`dashStyle`	String	`SOLID` `DASHED` `DOTTED` `DASHDOTTED`

```
style: [name: "myTemplate", color: "red", thickness: 2, dashStyle: "dashed"]
style: "myTemplate"   // shorthand: loads named style only
```

`createStyle(...)`

```
createStyle(name: "myStyle", style: [color: "red", thickness: 2])
```

Param	Required	Description
`name`	yes	Style name to register
`style`	yes	Map of style properties (same keys as `style:` map)

`linearGradient(...)`

Key	Type	Default	Description
`from`	`[x,y]` / Point	`[0,0]`	Start point
`to`	`[x,y]` / Point	`[1,0]`	End point
`relative`	boolean	`true`	Coords relative to shape
`cycle`	String	`"no_cycle"`	`no_cycle` `reflect` `repeat`
`0.0:`, `0.5:`, …	String / JMColor	—	Color stops

```
def lg = linearGradient(from: [0,0], to: [1,0],
                        relative: true,
                        0.0: "blue", 0.5: "red", 1.0: "#00FF00")
```

`radialGradient(...)`

Key	Type	Default	Description
`center`	`[x,y]` / Point	`[0.5,0.5]`	Gradient center
`radius`	double	`0.5`	Gradient radius
`focusAngle`	double	`0`	Focus point angle (radians)
`focusDistance`	double	`0`	Focus distance from center
`relative`	boolean	`true`	Coords relative to shape
`cycle`	String	`"no_cycle"`	`no_cycle` `reflect` `repeat`
`0.0:`, `1.0:`, …	String / JMColor	—	Color stops

```
def rg = radialGradient(center: [0.5,0.5], radius: 0.5,
                        0.0: "yellow", 1.0: "red")
```

`colorScale(...)`

```
def cs = colorScale(0.0: "blue", 0.5: "white", 1.0: "red")
def cs = colorScale((-2): "blue", 0: "white", 2: "red")   // negative keys: wrap in ()
```

Other Objects

`axes(...)`

Key	Type	Description
`xRange` / `yRange`	`[from, to]`	Primary tick range (or `range:` for both)
`xStep` / `yStep`	double	Primary tick step (default `1`)
`secondary`	Map	Shorthand: sets both `secondaryX` and `secondaryY`
`secondaryX` / `secondaryY`	Map `{from,to,step,maxWidth}`	Secondary ticks
`ticksX` / `ticksY`	List	Individual ticks (Number or Map `{at, label, type, maxWidth, scale}`)
`style`	Map	Global axes style
`xStyle` / `yStyle`	Map	Individual axis line style
`xTickStyle` / `yTickStyle`	Map	Tick mark style
`xLegendStyle` / `yLegendStyle`	Map	Tick label style

`tick.type`: `PRIMARY` (default) or `SECONDARY`

```
def ax = axes(xRange: [-3, 3], yRange: [-2, 2], xStep: 1, yStep: 1,
              secondaryX: [step: 0.5],
              ticksX: [[at: Math.PI, label: r"\pi"]])
```

`cartesianGrid(...)`

Key	Type	Default	Description
`center`	`[x,y]` / Coords	`[0,0]`	Grid center
`steps`	double / `[sx,sy]`	`[1,1]`	Primary grid step
`divisions`	int / `[dx,dy]`	`[1,1]`	Secondary subdivisions
`primaryStyle`	Map	—	Style for primary lines
`secondaryStyle`	Map	—	Style for secondary lines

```
def grid = cartesianGrid(steps: 1, divisions: 4,
                         primaryStyle: [color: "gray"],
                         secondaryStyle: [color: "lightgray"])
```

`contourPlot(...)`

Key	Type	Description
`function`	BiFunction	`(x,y) -> value` **required**
`levels`	List or Map `{min,max,step}`	Contour levels **required**
`xMin/xMax/yMin/yMax`	double	Domain (default: camera view)
`cols` / `rows`	int	Grid resolution (default `100`)
`colorScale`	ColorScale	Color scale instance
`style`	Map	Applied to all contour shapes

```
def cp = contourPlot(
    function: (x, y) -> x*x + y*y,
    levels: [min: 0.5, max: 3.0, step: 0.5],
    cols: 150, rows: 150,
    colorScale: colorScale(0.0: "blue", 1.0: "red"))
```

`arrow(...)`

Key	Type	Default	Description
`from`	`[x,y]` / Coords	**required**	Start point
`to`	`[x,y]` / Coords	**required**	End point
`type`	int / String	`ARROW1`	End arrowhead
`tail`	int / String	`NONE_BUTT`	Start arrowhead
`curvature`	double	—	Arrow curvature
`arrowThickness`	double	—	Arrowhead thickness
`startScale` / `endScale`	double	—	Arrowhead scale
`label`	String / Map	—	Text label
`lengthLabel`	String / Map	—	Length label
`vecLabel`	String / Map	—	Vector label
`style`	Map	—	Style

Arrow types: `NONE_BUTT` (0) `ARROW1` (1) `ARROW2` (2) `ARROW3` (3) `SQUARE` (4) `BULLET` (5)

```
def ar = arrow(from: [0,0], to: [1,1], type: "ARROW2", tail: "NONE_BUTT",
               label: [text: r"v", scale: 0.8, gap: 0.05])
```

Label sub-map keys: `text`, `format`, `color`, `drawColor`, `fillColor`, `thickness`, `scale`, `gap`, `upSide`, `rotation` (`FIXED` `ROTATE` `SMART`)

`connector(...)`

Same params as `arrow(...)` plus:

Key	Type	Description
`from`	MathObject	**required** — source object
`to`	MathObject	**required** — destination object
`startScaleBBox`	double	Bounding-box start scale
`endScaleBBox`	double	Bounding-box end scale

```
def con = connector(from: shapeA, to: shapeB, type: 2,
                    curvature: 0.3, label: "f")
```

`delimiter(...)`

Standard form:

Key	Type	Default	Description
`from`	`[x,y]` / Coords	**required**	Start point
`to`	`[x,y]` / Coords	**required**	End point
`type`	String	`BRACE`	`BRACE` `PARENTHESIS` `BRACKET` `INVISIBLE` `LENGTH_ARROW` `LENGTH_BRACKET`
`gap`	double	`0.1`	Distance from anchor to delimiter body

Stacked form:

Key	Type	Description
`stackedTo`	MathObject	**required** — object to stack to
`anchor`	String	`UPPER` (def) `LOWER` `LEFT` `RIGHT`
`type`	String	See above
`gap`	double	Distance from object

Common optional params:

Key	Description
`delimiterScale`	Scale of the delimiter shape
`amplitudeScale`	Scale of the delimiter amplitude
`labelGap`	Distance from label to delimiter shape
`label`	String or Map `{text, color, scale, gap, rotation}`
`lengthLabel`	String or Map `{format, color, scale, gap, rotation}`
`rotation`	`FIXED` `ROTATE` `SMART`
`style`	Style map

```
def d = delimiter(from: [0,0], to: [2,0], type: "BRACE", gap: 0.15,
                  label: [text: r"$L$", scale: 0.8, gap: 0.05])

def d = delimiter(stackedTo: myShape, anchor: "UPPER", type: "BRACKET",
                  lengthLabel: [format: "0.00", color: "blue"])
```

`labelTip(...)`

Key	Type	Default	Description
`path`	hasPath	**required**	Path object to attach to
`text`	String	`""`	Label text
`location`	double	`0.5`	Position along path `[0,1]`
`upSide`	boolean	`false`	Place above path
`gap`	double	—	Distance to shape
`gapRelative`	boolean	—	Gap relative to size
`rotation`	String	—	`FIXED` `ROTATE` `SMART`
`slope`	String	—	`POSITIVE` `NEGATIVE`
`anchor`	AnchorType	—	—
`style`	Map	—	Style properties

```
def lt = labelTip(path: myShape, location: 0.5, text: r"$A$",
                  upSide: true, gap: 0.1, rotation: "rotate")
```

Constructible Objects

Points

Method	Required	Optional
`ctPoint(at:[x,y])`	`at`	style, layer, visible, transform, `name`
`ctMidPoint(A:, B:)`	`a`, `b`	—
`ctMidPoint(segment:)`	`segment` (CTSegment)	—
`ctIntersectionPoint(A:, B:)`	`a`, `b`	`index` (0/1)
`ctMirrorPoint(point:, axis:)`	`point`, `axis` (CTAbstractLine)	—
`ctMirrorPoint(point:, A:, B:)`	`point`, `a`, `b`	—
`ctMirrorPoint(point:, center:)`	`point`, `center`	—
`ctRotatedPoint(point:, center:, angle:)`	`point`, `center`, `angle`	—
`ctTranslatedPoint(point:, vector:)`	`point`, `vector` (CTVector)	—
`ctPointOnObject(on:)`	`on` (PointOwner)	—

> All CT methods accept: `style`, `layer`, `visible`, `transform`, `name` (CT name)

```
def A = ctPoint(at: [0, 0])
def M = ctMidPoint(A: A, B: B)
def I = ctIntersectionPoint(A: line1, B: circle1, index: 0)
def R = ctRotatedPoint(point: A, center: O, angle: Math.PI/3)
```

Lines

Method	Forms
`ctLine`	`A:, B:` or `A:, dir:` (HasDirection)
`ctSegment`	`A:, B:`
`ctRay`	`A:, B:` or `A:, dir:`
`ctVector`	`A:, B:`
`ctAngleBisector`	`A:, B:` (vertex), `C:`
`ctPerpBisector`	`A:, B:` or `segment:` (CTSegment)
`ctLineOrthogonal`	`A:, B:` or `A:, dir:`
`ctTransformedLine(line:, ...)`	`mirrorAxis:` / `mirrorCenter:` / `vector:` / `center:+angle:`

```
def l  = ctLine(A: A, B: B)
def s  = ctSegment(A: A, B: B, style: [color: "blue"])
def pb = ctPerpBisector(segment: s)
def tl = ctTransformedLine(line: l, mirrorAxis: l2)
def tl = ctTransformedLine(line: l, center: O, angle: Math.PI/4)
```

Circles

Method	Forms
`ctCircle`	`center:+radius:` / `center:+through:` / `A:+B:+C:`
`ctCircleArc`	`center:, A:, B:`
`ctCircleSector`	`center:, A:, B:`
`ctSemiCircle`	`A:, B:`
`ctEllipse`	`focus1:, focus2:, A:`
`ctTransformedCircle(circle:, ...)`	`mirrorAxis:` / `mirrorCenter:` / `vector:` / `center:+angle:`

```
def c  = ctCircle(center: O, radius: 1.5)
def c  = ctCircle(A: P, B: [1,1], C: Q) //Circle through P, (1,1) and Q
def e  = ctEllipse(focus1: F1, focus2: F2, A:
         P) //Ellipse with focus F1, F2 that passes through P
def tc = ctTransformedCircle(circle: c, vector: v)
```

`ctLatex(...)`

Param	Type	Default	Description
`text`	String	**required**	LaTeX expression
`anchor`	`[x,y]` / Coords	**required**	Anchor point
`anchorType`	String	`"DIAG4"`	Anchor side (see AnchorType)
`gap`	double	`0.3`	Gap from anchor
`scale`	double	`0.5`	Size scale
`style`	Map	—	Style properties

```
def lbl = ctLatex(text: r"\alpha", anchor: [1, 0],
                  anchorType: "left", gap: 0.15, scale: 0.7)
```

Other constructibles

Method	Forms
`ctPolygon`	`points:[A,B,C,...]` (free) or `A:+B:+sides:` (regular)
`ctAngleMark`	`center:, A:, B:` + optional `radius:`, `isRight:`
`ctTangent`	`point:+circle:` + optional `index:` or `circle1:+circle2:` + optional `index:`

```
def poly = ctPolygon(points: [A, B, C, D])
def hex  = ctPolygon(A: A, B: B, sides: 6)
def am   = ctAngleMark(center: O, A: P1, B: P2, radius: 0.15, isRight: true)
def t    = ctTangent(point: P, circle: c, index: 0)
def t    = ctTangent(circle1: c1, circle2: c2, index: 2)
```