Visual Servoing Using ArUco Codes for Ackermann Vehicle Course Correction

Background

Daksh Raval | NODE Lab

February 5, 2026

Contents

1 PID Based Visual Servoing

Nutshell 1. *Go to each specific, predetermined waypoint to establish 'ground truth' of where ArUco codes should be in the image frame when at that waypoint. Then, using some pin-hole camera trigonometry, and by linearly approximating visual feed change to use the Jacobian matrix, we update the 'Twist' vector ($[v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]$)*

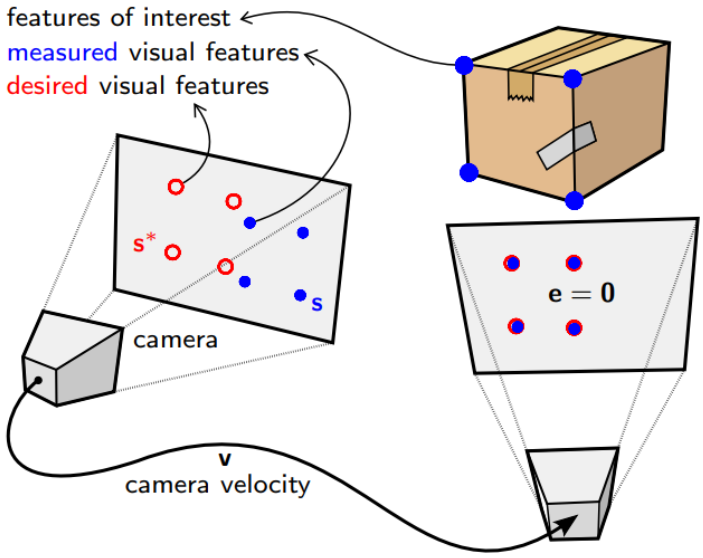

Figure 1: Using the feature image frame position error and linear algebra to correct camera position.

1.1 Background

Nutshell 2. *Relevant state variables:*

- $[\mathbf{x}, \mathbf{y}]$ *centered digital image frame coordinates.* ***Note image frame is focal length away from lens center!***
- $[\mathbf{X}, \mathbf{Y}, \mathbf{Z}]$ *feature coordinates w.r.t lens center.* ***Note that*** $[\dot{\mathbf{X}}, \dot{\mathbf{Y}}, \dot{\mathbf{Z}}]$ ***is exactly opposite to how the camera moves!!***
- $\mathbf{R} \in \mathbb{R}^{\mathbf{3x3}}$ *is the rotation matrix, and* $\mathbf{t} \in \mathbb{R}^{\mathbf{3x1}}$ *the translation vector, w.r.t 'world'/intertial frame*
- $[\mathbf{u}, \mathbf{v}]$ *are the pixel positions* ***on the physical camera's sensor grid****; can be centered or measured from corner.*
- $\rho_{\mathbf{x}}$ *and* $\rho_{\mathbf{y}}$ *are the sizes of each of the tiny physical camera sensors in a grid*
  - *ZEDMini documentation says 0.002mm [4]*

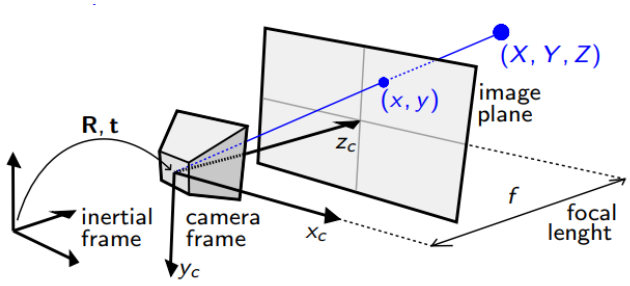

Figure 2: Caption

1.2 Jacobian Matrix

Nutshell 3. *If we assume a non-linear vector function is linear close to it's initial position, we can differentiate each output vector variable entry w.r.t each input variable* ***[1]****.*

$$Z\left(\mathbb{R}^2 \Rightarrow \mathbb{R}^2\right) = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix} = \begin{bmatrix} x + sin(y) \\ y + cos(x) \end{bmatrix} \xrightarrow{Jacobian} \begin{bmatrix} \partial f/\partial x & \partial g/\partial x \\ \partial f/\partial y & \partial g/\partial y \end{bmatrix}$$

1.3 Interaction Matrix

Nutshell 4. *We make the linear approximation that the 'change in visual feed' about a specific point in a short time step is, well, linear. We can then find the Jacobian matrix, using:*

- *focal length*
- *digit pixel position,* $(\mathbf{u}, \mathbf{v})$*, of feature of interest in image frame (focal length away from lens center)*
- *depth to point*
- *similar triangles*

to relate how features of interest move as the car moves, ie. rate of change in pixel position w.r.t to rate of change of camera position ***[2]****.*

1.3.1 Pinhole Camera Projection

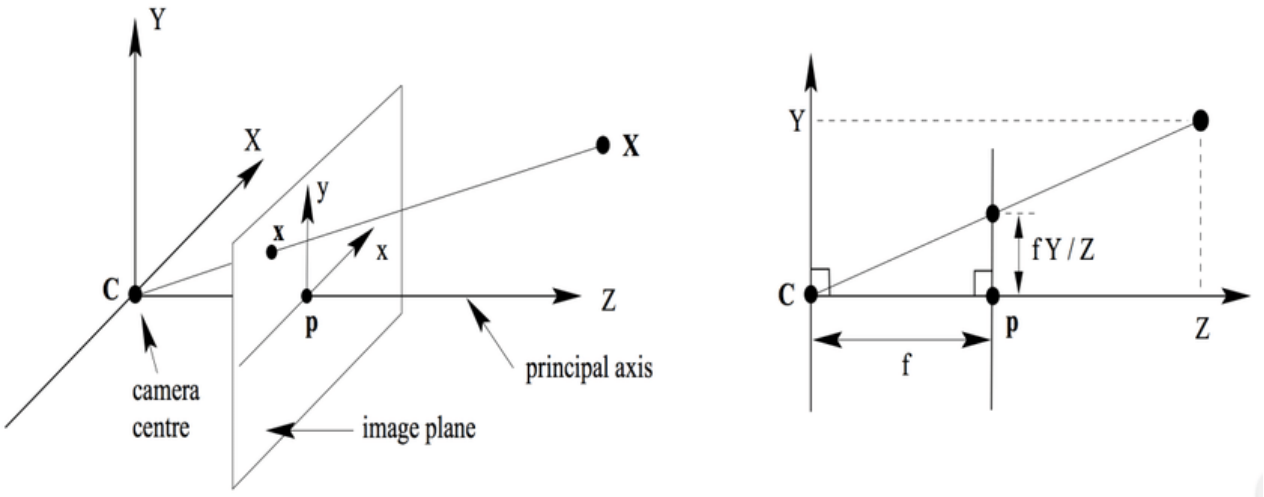

Figure 3: Note: 'f' means focal length; Z/principal-axis protrudes straight out from lens; X-axis different from $\tilde{\mathbf{X}}$ position vector of feature of interest in 3D frame (origin at lens center).

As you can see, after 'projecting' onto the YZ or XZ plane (orthogonal to image plane), you can simply use similar triangles to specify the 3D to pixel projection relation.

$$\frac{y}{f} = \frac{Y}{Z} \Rightarrow y = f \cdot \frac{Y}{Z} \qquad \frac{x}{f} = \frac{X}{Z} \Rightarrow x = f \cdot \frac{X}{Z}$$

Then differentiate w.r.t time, and as X, Y & Z all vary w.r.t time, just apply chain rule **[3]**:

$$\dot{y} = f \cdot \frac{Z \cdot \dot{Y} - Y \cdot \dot{Z}}{Z^2} = f \cdot \frac{\dot{Y}}{Z} - f \cdot \frac{Y}{Z} \cdot \frac{\dot{Z}}{Z} = f \cdot \frac{\dot{Y}}{Z} - y \cdot \frac{\dot{Z}}{Z} = \dot{y}$$

$$\dot{x} = f \cdot \frac{Z \cdot \dot{X} - X \cdot \dot{Z}}{Z^2} = f \cdot \frac{\dot{X}}{Z} - f \cdot \frac{X}{Z} \cdot \frac{\dot{Z}}{Z} = f \cdot \frac{\dot{X}}{Z} - x \cdot \frac{\dot{Z}}{Z} = \dot{x}$$

You can express with linear algebra, which seems unnecessary now, but becomes useful later:

$$\begin{bmatrix} f \cdot \frac{\dot{X}}{Z} - y \cdot \frac{\dot{Z}}{Z} \\ f \cdot \frac{\dot{X}}{Z} - x \cdot \frac{\dot{Z}}{Z} \end{bmatrix} = \dot{X} \cdot \begin{bmatrix} f/Z \\ 0 \end{bmatrix} + \dot{Y} \cdot \begin{bmatrix} 0 \\ f/Z \end{bmatrix} + \dot{Z} \begin{bmatrix} -x/Z \\ -y/Z \end{bmatrix}$$

$$\boxed{\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} f/Z & 0 & -x/Z \\ 0 & f/Z & -y/Z \end{bmatrix} \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix}} \tag{1}$$

1.3.2 Rigid Body Dynamics

Nutshell 5. *The time derivative of position of some point, A, on a rigid body can be represented by the linear velocity of another point, B, plus angular velocity cross-product with the translation vector from B to A. Just like MEC E 250, but with no planar motion restrictions*

$$\boxed{\begin{gathered} \vec{v}_A = \vec{v}_B + \vec{\omega}_{RB} \times \vec{r}_{A/B} \\ \underbrace{\begin{bmatrix} \dot{x}_A \\ \dot{y}_A \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x}_B \\ \dot{y}_B \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \omega_Z \end{bmatrix} \times \begin{bmatrix} \Delta x_{A/B} \\ \Delta y_{A/B} \\ 0 \end{bmatrix}}_{PLANAR\ MOTION} \\ \underbrace{\begin{bmatrix} \dot{x}_A \\ \dot{y}_A \\ \dot{z}_A \end{bmatrix} = \begin{bmatrix} \dot{x}_B \\ \dot{y}_B \\ \dot{z}_B \end{bmatrix} + \begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \times \begin{bmatrix} \Delta x_{A/B} \\ \Delta y_{A/B} \\ \Delta z_{A/B} \end{bmatrix}}_{3D\ MOTION} \end{gathered}}$$

$$\dot{P}_A = V_B + \omega_{AB} \times P_{A/B}$$

We want to know how the camera moves relative to the world, and by extension the feature of interest. This is exactly opposite to how the feature of interest moves relative to the camera, so we add a negative:

$$\boxed{\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix}_A = - \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}_B - \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}_{AB} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{\textbf{B to A}}} \tag{2}$$

This is saying: the 3D velocity of a world point (as seen by the camera) is the sum of the negative camera translation velocity and the apparent motion from camera rotation.

1.3.3 Angular Velocity Isolation

Nutshell 6. *Using some sleight of hand, we can start to isolate the angular velocity part of the 'Twist' vector.*

Instead of doing the cross product the regular way, you can do matrix-vector multiplication, with the matrix being the skew-symmetric matrix of the first vector in the cross product.

It took me forever to figure out what they were doing lol, but in short:

$$\omega \times P = \begin{bmatrix} 0 & -w_Z & w_Y \\ w_Z & 0 & -w_X \\ -w_Y & w_X & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = X \cdot \begin{bmatrix} 0 \\ w_Z \\ -w_Y \end{bmatrix} + Y \cdot \begin{bmatrix} -w_Z \\ 0 \\ w_X \end{bmatrix} + Z \cdot \begin{bmatrix} w_Y \\ -w_X \\ 0 \end{bmatrix} = \begin{bmatrix} Z \cdot w_Y - Y \cdot w_Z \\ X \cdot w_Z - Z \cdot w_X \\ Y \cdot w_X - X \cdot w_Y \end{bmatrix}$$

If you replace the w_Z with Z, but flip the positions of X and Y, you get the same result, but negative, quite beautifully:

$$-\omega \times P = \begin{bmatrix} 0 & -Z & Y \\ Z & 0 & -X \\ -Y & X & 0 \end{bmatrix} \begin{bmatrix} w_X \\ w_Y \\ w_Z \end{bmatrix} = w_X \cdot \begin{bmatrix} 0 \\ Z \\ -Y \end{bmatrix} + w_Y \cdot \begin{bmatrix} -Z \\ 0 \\ X \end{bmatrix} + w_Z \cdot \begin{bmatrix} Y \\ -X \\ 0 \end{bmatrix} = - \begin{bmatrix} Z \cdot w_Y - Y \cdot w_Z \\ X \cdot w_Z - Z \cdot w_X \\ Y \cdot w_X - X \cdot w_Y \end{bmatrix}$$

Don't ask me how this works, it just does.

1.3.4 Simplying to One Matrix-Vector Multiplication

Nutshell 7. *Plug $[\dot{X}, \dot{Y}, \dot{Z}]$ from equation (2) into equation (1), and after a few more steps, you have the interaction matrix*

No, I'm not writing out this matrix multiplication lol. Now we use the same similar triangles conclusion from before:

$$X = x \cdot \frac{Z}{f} \qquad \textit{and} \qquad Y = y \cdot \frac{Z}{f}$$

Plugging this into the previous matrix we have:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} -\frac{f}{Z} & 0 & \frac{x}{Z} & \frac{xy}{f} & -f - \frac{x^2}{f} & y \\ 0 & -\frac{f}{Z} & \frac{y}{Z} & f + \frac{y^2}{f} & -\frac{xy}{f} & -x \end{bmatrix} \begin{bmatrix} v_X \\ v_Y \\ v_Z \\ \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix}$$

This deceptively seems like the 'end goal', as (x,y) are in the 'image frame', but:

Remark 1. *The image frame is the hypothetical 2D projection of the scene onto an imaginary physical plane perpendicular to the camera lens, a distance equal to the focal length away.* ***The image frame isn't the same as the pixelated digital image.***

To get around this, we convert. The principal point, (u_0, v_0), is located at the 'center' of the digital image. We normalize/convert from physical to digital image frame by:

$$u = u_0 + \frac{x}{\rho_w} \qquad v = v_0 + \frac{y}{\rho_h} \qquad \Rightarrow \qquad \dot{u} = \frac{\dot{x}}{\rho_w} \qquad \dot{v} = \frac{\dot{y}}{\rho_h}$$

Where:

- u is the 'x' pixel position from the bottom right corner
- v is the 'y' pixel position from the bottom right corner
- ρ_h and ρ_w are the 'height' and 'width' of each individual physical camera sensor in the grid/array

Isolating x and y so we can substitute them, and representing $(u - u_0)$ and $(v - v_0)$ as $\bar{u}$ and $\bar{v}$:

$$x = \bar{u} \cdot \rho_w \qquad y = \bar{v} \cdot \rho_h$$

Plugging this in, we get our interaction matrix finally:

$$\underbrace{\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix}}_{pixel\ motion} = \underbrace{\begin{bmatrix} -\frac{f}{\rho_w Z} & 0 & \frac{\bar{u}}{Z} & \frac{\bar{u}\bar{v}\rho_h}{f} & -f - \frac{\bar{u}^2\rho_w}{f} & \bar{v} \\ 0 & -\frac{f}{\rho_h Z} & \frac{\bar{v}}{Z} & f + \frac{\bar{v}^2\rho_h}{f} & -\frac{\bar{u}\bar{v}\rho_h}{f} & -\bar{u} \end{bmatrix}}_{Relates\ real\ motion\ to\ motion\ in\ video\ feed} \underbrace{\begin{bmatrix} v_X \\ v_Y \\ v_Z \\ \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix}}_{real\ motion} \tag{3}$$

1.4 Control Feedback

To simplify this, and bring this closer to what we'll actually compute, we'll define the following:

- $\tilde{\mathbf{s}}$: $[u, v]$
- $\mathbf{s}^*$: 'desired' $[u, v]$
- $\mathbf{e} = \mathbf{s} - \mathbf{s}^*$

Nutshell 8. *To make the error stably approach 0, we 'scale' the time derivative of the error by how big the error is. This is a P controller, or in another sense, they're eigenvectors.*

$$\vec{s} = \begin{bmatrix} u \\ v \end{bmatrix} \Rightarrow \dot{s} = L \cdot t \qquad \dot{e} = \dot{s} - \dot{s}^* = -\lambda e,\ \lambda > 0$$

λ is the 'control/proportional' gain parameter. Now, assuming you have a stationary target, you use feedback control only (ie. $\dot{s}^* = 0$).

$$\dot{e} = -\lambda e = \dot{s} = L \cdot t \Rightarrow \boxed{t = -\lambda L^+ e} \tag{4}$$

Where v is the new suggested twist vector, and L^+ is the inverse of the interaction matrix.

1.4.1 Real-Time Inverse Computation of Interaction Matrix

Nutshell 9. *We take a shortcut that ROS1's Eigen's library has optimized a ton.*

The specifics are:

- compute the **Moore-Penrose pseudoinverse**
- use Singular Value Decomposition

ROS handles this, but long story short, it gives you a good enough inverse, cause full Gauss Jordan elimination for this is like trying to do a Mad Minute multiplication table with ChatGPT.

1.4.2 Stacking Information

Nutshell 10. *One point isn't enough; to use more points, you just stack. Pseudo inverse*

$$L = \begin{bmatrix} L_1 \\ \vdots \\ L_N \end{bmatrix} \in \mathbb{R}^{2N\times 6} \Rightarrow L^* = \begin{bmatrix} L_1 & \dots & L_N \end{bmatrix}^* \in \mathbb{R}^{6\times 2N} \qquad e = \begin{bmatrix} e_1 \\ \vdots \\ e_N \end{bmatrix} = \begin{bmatrix} u_1 - u_1^* \\ v_1 - v_1^* \\ \vdots \\ u_N - u_N^* \\ v_N - v_N^* \end{bmatrix} \in \mathbb{R}^{2N\times 1}$$

1.5 Ackerman Model Integration

Nutshell 11. *Updated twist has 6 degrees of freedo, RC car has two, throttle and steering. Using some sleight of hand we reduce it.*

1.5.1 Angular Velocity to Steering Command

Nutshell 12. *Rearrange the following:*

$$\dot{\theta} = \omega = \frac{v}{L} \cdot tan(\delta) \Rightarrow \delta = arctan(\frac{\omega \cdot L}{v}) \xRightarrow{small\ \theta\ approx\ if\ you\ want} \delta = \frac{\omega \cdot L}{v}$$

1.6 PI Implementation

Nutshell 13. *P easy. For I: summing numerically integrated (error × time) error over last n frames with successful detection*

$$\boxed{Twist = -L^+(\lambda_P \cdot e_k + \lambda_I \cdot \sum_{k=-2}^{0} (e_k \cdot \Delta t_k)}$$

Remark 2. *You could try it either for only the previous timestamp, or the full error integral:*

$$Twist = -L^+(e_k \cdot (\lambda_P + \lambda_I \cdot \Delta t_{k-1\ to\ k})$$

$$Twist = -L^+(\lambda_P \cdot e_k + \lambda_I \cdot \sum_{n=0}^{k} (e_k \cdot \Delta t_k)$$

but neither of these make sense because:

- *using only the error from the previous timestamp essentially just adds another term in the servoing error with no real memory, just scaled by different/more things*
- *The full integral means the error from all frames since the very start is included, ie. no memory loss. So it doesn't "forgive you", even when you've reached the waypoint.*

1.7 Heading to Steering Angle Conversion

Nutshell 14. *Using kinematic bicycle model (no tire slip considerations). Heading is angle the center of gravity is "going", steering is the angle the front wheel is turned to.*

The **kinematic bicycle model** whose state variables are shown in this diagram [5]:

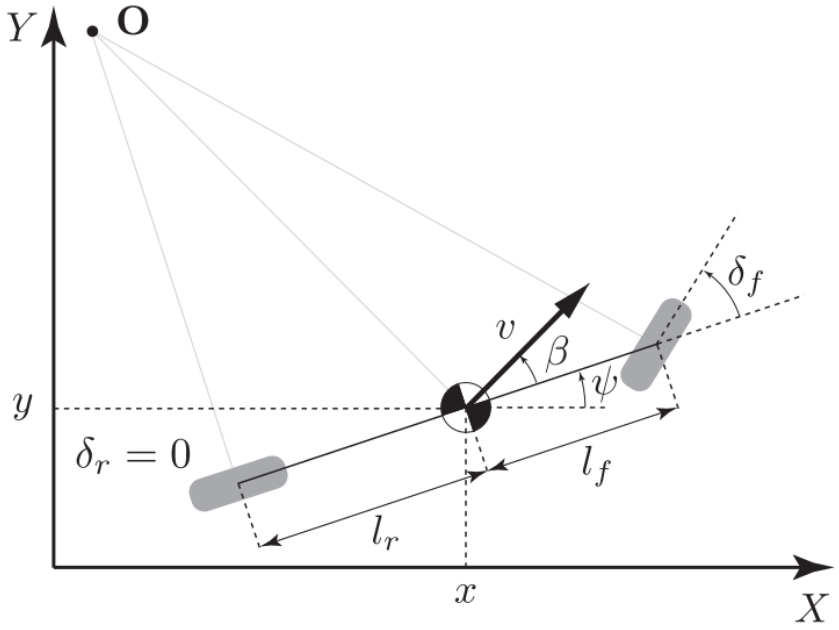

Figure 4: Diagram from ETH Paper

The equations of motion relevant here are [5]:

$$\dot{\psi} = \omega = \frac{v}{l_r} \cdot sin(\beta) \Rightarrow \beta = arcsin(\frac{\omega \cdot l_r}{v})$$

$$\beta = arcsin(\frac{\omega \cdot l_r}{v}) = arctan(\frac{l_r}{l_f + l_r} \cdot tan(\delta)) \Rightarrow \boxed{\delta = arctan(\frac{l_f + l_r}{l_r} \cdot tan(arcsin(\frac{\omega \cdot l_r}{v})))}$$

Remark 3. *This looks like an extremely fragile calculation because of all the singularities and domain limitations of arcsine, tangent and arctangent, but I clamp it, as explained below.*

Arcsine(x) is only defined $x \in [-1, 1]$, which makes sense because

$$\omega = \frac{v}{R} = \frac{v}{l_r} \cdot sin(\beta) \Rightarrow sin(\beta) = \frac{l_r}{R} = 1 \textbf{ only when } \beta = \frac{\pi}{2}$$

That means the front wheel is turned 90 degrees and the bicycle is only moving rotationally. I'll clamp the argument of arcsine so tangent doesn't blow up to infinity:

$$|\frac{\omega \cdot l_r}{v}| < 0.99999 \textbf{ because } arcsin(0.99999) \approx 1.566 < \frac{\pi}{2} \Rightarrow tan(1.566) \approx 200$$

Arctangent is defined for $x \in [-\infty, \infty]$. Long story short, our graph looks almost linear with some grooves, something like this:

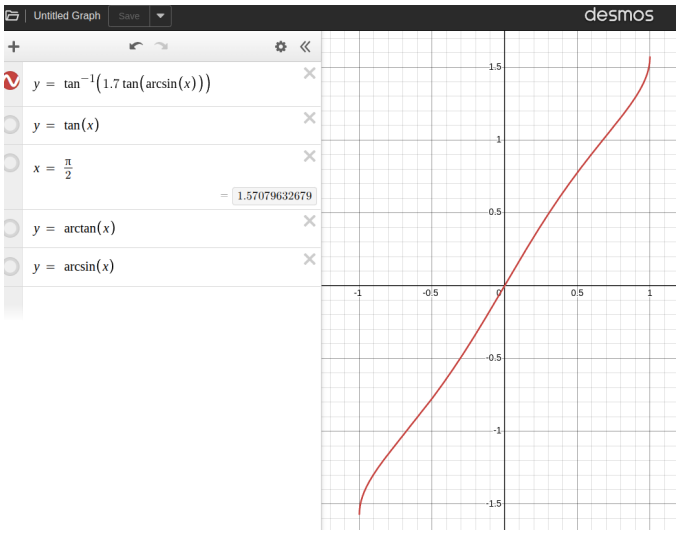

Figure 5: Caption

Remark 4. *The scaling factor means we can't simplify, ie.*

$$arctan(tan(x)) = x \qquad arctan(2tan(x)) \neq x$$

2 Appendix

Remark 5. *All references are clickable links*

[1] 3 Blue 1 Brown at Khan Academy explaining Jacobian matrices
[2] Runway Detection Paper on ResearchGate
[3] Visual Servoing for Steering & Manipulation by Swiss Researcher Antonio Paolillo
[4] ZEDMini Official Documentation
[5] Kinematic & Dynamic Bicycle Model Paper from ETH