Dubins Paths for Waveguide Routing

by Quentin Wach • 26 min read • Feb 15, 2024

What does the perfect waveguide routing look like? Traditional waveguide interconnects lack efficiency and optimization, leading to suboptimal photonic integrated circuits (PICs). Addressing key concerns like minimizing internal and radiation losses, reducing cross-talk, and avoiding unnecessary crossings, I explored the concept of finding the shortest path between two points in a PIC given a minimum bending radius. Dubins paths, already well known in robotics and control theory, emerged as a simple and highly practical solution.

photonic integrated circuits | design automation | robotics | gds

1 Introduction

I'll try to keep this short and not ramble too much. I just want to get this out there since I found it rather useful and can't believe it isn't yet standard.

When I started my work on layouts for photonic integrated circuit in September 2023, it became quickly obvious that the typical waveguide interconnects provided by libraries are not well behaved nor optimal as to improve the performance and compactness of the photonic integrated circuit. Most commonly, we have simple straights, circular arcs, all sorts of splines, as well as Euler-bends, in short: They are all simple analytical functions or splines that (try to) optimize themselves. All of these are either simple building blocks that leave most of the work to the designer who has to plan out and specify every route, or behave extremely poorly and lead to overly bendy results that may even cross each other.

So in my first two weeks there, I was thinking about what we have to consider. Waveguides have to fullfil multiple criteria. They need to be…

- As short as possible as to minimize internal losses.
- As straight and continuos as possible as to minimize radiation losses.
- As far away from other waveguides and structures as to minimize cross-talk.
- Does not cross other waveguides unless close to a 90° angle if absolutely necessary.

It became quickly obvious that trying to find an optimum for all these design parameters is impossible unless we either weigh their importance using a complicated metric we have to justify, or make much harder constraints. Since it is common to have a fixed minimum bending radius for waveguides, it makes sense to translate this here. I then boiled all of the considerations down to the single, I believe, most important question: **"What is the shortest path between the pin or vector $\vec{a}$ and pin or vector $\vec{b}$ given a minimum bending radius?"**

As it turns out, the answer is rather simple and long established in the field of robotics / control theory. **Dubins paths!**

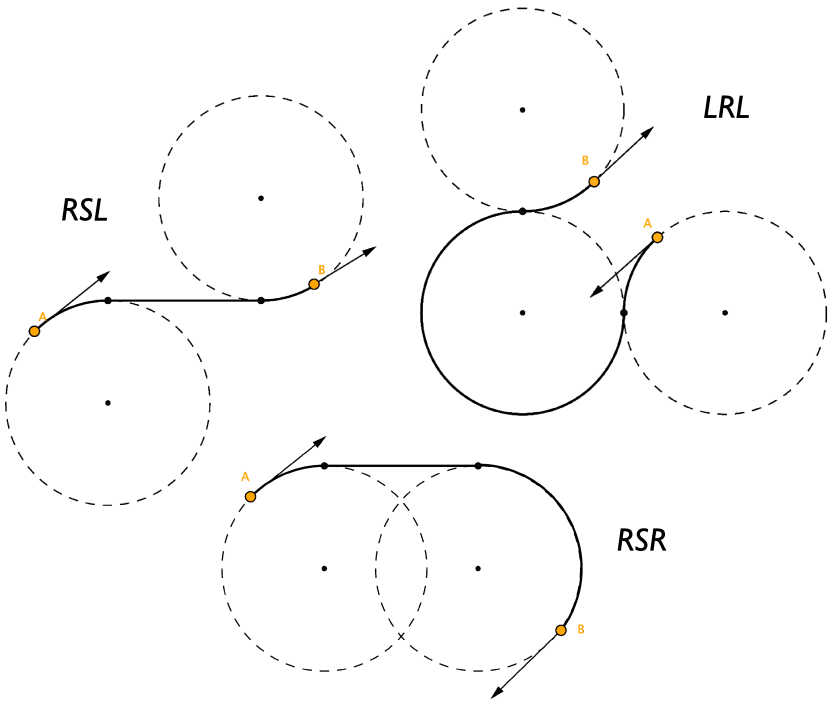
2 What are they?

Dubins paths are named after Lester Dubins, who introduced them in the 1950s[1]. They refer to the shortest paths that a vehicle can take from one point to another while constrained to move at a specific minimum turning radius. These paths are thus commonly studied in the field of motion planning for vehicles, particularly in robotics and aerospace engineering.

These paths simply connect circular arcs with straights which leaves us with a couple of characteristic Dubins paths:

- **LRL** (Left-Right-Left)
- **RLR** (Right-Left-Right)
- **LSL** (Left-Straight-Left)
- **RSR** (Right-Straight-Right)
- **LSR** (Left-Straight-Right)
- **RSL** (Right-Straight-Left)

You can see three examples of the possible paths in the figure I adapted here[2].



3 Geometric Construction

Dubins did indeed prove that these trajectories are the shortest paths mathematically. The geometric construction is quite intuitive and a nice toy problem to figure out on ones own but it can quickly explode into multiple pages of pen and paper calculations and diagrams which is why I am not going to go into it here. Instead, I refer to a great overview and explanation of the synthesis of Dubins paths given by David A. Anisi[3]. A wonderful and in-depth guide is also given by Andy G[4].

4 Code

Let's go through some code to generate them, though. There is an abundance of implementations of Dubins paths available on the internet[5]. What I present here is a hacked together version. First, it's useful to define a little helper to keep all the angles within the range of $[0, 2\pi)$:

```
import math as m


# normalizes an angle to the range [0, 2*pi)
def mod_to_pi(angle):
    return angle - 2.0*m.pi*m.floor(angle/2.0/m.pi)
```

Next, we actually find our Dubins paths using a planner function `general_planner()` which creates and compares the lengths of all six possible routes using `dubins_path_length()`. The data of the solution is then returned with `dubins_path()`.

```
import nazca as nd
from nazca.interconnects import Interconnect

# Function to find the optimal Dubins path between two points
def general_planner(planner, alpha, beta, d):
    """
    Calculates the Dubins path between two points defined by their
    angles and distance given a plan.

    Parameters:
        planner (str): Type of Dubins path. Can be one of 'LSL', 'RSR',
                       'LSR', 'RSL', 'RLR', or 'LRL'.
        alpha (float): Starting orientation angle in radians.
        beta (float): Ending orientation angle in radians.
        d (float): Distance between the two points.

    Returns:
        tuple or None: A tuple containing the Dubins path as a list of
        angles and distances, the mode of the planner,
        and the cost of the path.
        Returns None if the path is not feasible.
    """

    # Convert angles to sine and cosine
    sa = m.sin(alpha)
    sb = m.sin(beta)
    ca = m.cos(alpha)
    cb = m.cos(beta)
    c_ab = m.cos(alpha - beta)
    # Convert planner input to uppercase
    planner_uc = planner.upper()

    # Case LSL Dubins path
    if planner_uc == 'LSL':
        # Calculate intermediate values for LSL path
        tmp0 = d + sa - sb
        p_squared = 2 + (d * d) - (2 * c_ab) + (2 * d * (sa - sb))
        if p_squared < 0:  # Path not feasible
            return None
        tmp1 = m.atan2((cb - ca), tmp0)
        t = mod_to_pi(-alpha + tmp1)
        p = m.sqrt(p_squared)
        q = mod_to_pi(beta - tmp1)

    # Case RSR Dubins path (similar structure to LSL)
    elif planner_uc == 'RSR':
```

```
        # Calculate intermediate values for RSR path
        tmp0 = d - sa + sb
        p_squared = 2 + (d * d) - (2 * c_ab) + (2 * d * (sb - sa))
        if p_squared < 0:  # Path not feasible
            return None
        tmp1 = m.atan2((ca - cb), tmp0)
        t = mod_to_pi(alpha - tmp1)
        p = m.sqrt(p_squared)
        q = mod_to_pi(-beta + tmp1)

    # Case LSR Dubins path (similar structure to LSL)
    elif planner_uc == 'LSR':
        # Calculate intermediate values for LSR path
        p_squared = -2 + (d * d) + (2 * c_ab) + (2 * d * (sa + sb))
        if p_squared < 0:  # Path not feasible
            return None
        p = m.sqrt(p_squared)
        tmp2 = m.atan2((-ca - cb), (d + sa + sb)) - m.atan2(-2.0, p)
        t = mod_to_pi(-alpha + tmp2)
        q = mod_to_pi(-mod_to_pi(beta) + tmp2)

    # Case RSL Dubins path (similar structure to LSR)
    elif planner_uc == 'RSL':
        # Calculate intermediate values for RSL path
        p_squared = (d * d) - 2 + (2 * c_ab) - (2 * d * (sa + sb))
        if p_squared < 0:  # Path not feasible
            return None
        p = m.sqrt(p_squared)
        tmp2 = m.atan2((ca + cb), (d - sa - sb)) - m.atan2(2.0, p)
        t = mod_to_pi(alpha - tmp2)
        q = mod_to_pi(beta - tmp2)

    # Case RLR Dubins path (similar structure to LSL)
    elif planner_uc == 'RLR':
        # Calculate intermediate values for RLR path
        tmp_rlr = (6.0 - d * d + 2.0 * c_ab + 2.0 * d * (sa - sb)) / 8.0
        if abs(tmp_rlr) > 1.0:  # Path not feasible
            return None
        p = mod_to_pi(2 * m.pi - m.acos(tmp_rlr))
        t = mod_to_pi(alpha - m.atan2(ca - cb, d - sa + sb) + mod_to_pi(p / 2.
        q = mod_to_pi(alpha - beta - t + mod_to_pi(p))

    # Case LRL Dubins path (similar structure to RLR)
    elif planner_uc == 'LRL':
        # Calculate intermediate values for LRL path
        tmp_lrl = (6. - d * d + 2 * c_ab + 2 * d * (- sa + sb)) / 8.
```

```
        if abs(tmp_lrl) > 1:  # Path not feasible
            return None
        p = mod_to_pi(2 * m.pi - m.acos(tmp_lrl))
        t = mod_to_pi(-alpha - m.atan2(ca - cb, d + sa - sb) + p / 2.)
        q = mod_to_pi(mod_to_pi(beta) - alpha - t + mod_to_pi(p))

    else:
        print("The given plan ", planner, " is false.")  # Invalid planner inp

    # Create the Dubins path as a list of angles and distances
    path = [t, p, q]

    # Adjust angles if planner segments are lowercase (for reverse motion)
    for i in [0, 2]:
        if planner[i].islower():
            path[i] = (2 * m.pi) - path[i]

    # Calculate the cost of the path (sum of absolute values of angles and dis
    cost = sum(map(abs, path))

    return (path, mode, cost)

# Function to calculate the length of a Dubins path
def dubins_path_length(start, end, radius):
    # Unpack start and end configurations
    (sx, sy, syaw) = start
    (ex, ey, eyaw) = end

    # Convert angles to radians
    syaw = m.radians(syaw)
    eyaw = m.radians(eyaw)

    # Define the turning radius
    c = radius

    # Calculate differences in coordinates
    ex = ex - sx
    ey = ey - sy

    # Project end point onto start orientation
    lex = m.cos(syaw) * ex + m.sin(syaw) * ey
    ley = - m.sin(syaw) * ex + m.cos(syaw) * ey
    leyaw = eyaw - syaw

    # Calculate the total distance
    D = m.sqrt(lex ** 2.0 + ley ** 2.0)
```

```
    return D

# Finds the Dubins path between two points
def dubins_path(start, end, radius):

    # Calculate the length
    D = dubins_path_length(start, end, radius)
    d = D / radius

    # Define important angles
    theta = mod_to_pi(m.atan2(ley, lex))
    alpha = mod_to_pi(- theta)
    beta = mod_to_pi(leyaw - theta)

    # Iterate through all possible paths
    planners = ['LSL', 'RSR', 'LSR', 'RSL', 'RLR', 'LRL']
    bcost = float("inf")
    bt, bp, bq, bmode = None, None, None, None
    for planner in planners:

        # find the solution for the Dubins path
        solution = general_planner(planner, alpha, beta, d)

        if solution is None:
            continue

        # Collect the data from the solution
        (path, mode, cost) = solution
        (t, p, q) = path
        if bcost > cost:
            # Best cost
            bt, bp, bq, bmode = t, p, q, mode
            bcost = cost

    return (list(zip(bmode, [bt*c, bp*c, bq*c], [c] * 3)))
```

It's more difficult to then actually create the final curve but the `gds_solution()` function can easily be extended to draw Dubins paths with Matplotlib[6], too, for example. We can wrap it all up into a single, simple to use function just like any other provided by the design library you might be using. In this case, I have been using the Nazca library[7] which comes with several interconnects, including straights and circular arcs which are used often but very tedious and slow to work with alone. Using `dubin_p2p()` as shown below, we can simply define the start pin, the end pin, and our code will route a Dubins path between them using the straights and circular arcs provided by Nazca. Of course, this can be easily adapted to other tools like GDSFactory[8].

```python
# Generate a Nazca cell for a given Dubins path solution
def gds_solution(xs, pin1, pin2, solution):
    """
    Analogously to plotSolution() we draw the trajectory of a
    given solution for a Dubins path between
    two points, here, the pins, to be rendered in .gds!
    """
    # Get the pin vectors from nazca pin1 and pin2
    start = pin1.xya()
    end = pin2.xya()

    # Change the angle for the second pin
    new_end = list(end)
    new_end[2] = new_end[2] - 180
    end = tuple(new_end)

    # Define the xs for the .gds file
    ic = Interconnect(xs=xs)

    # Create the cell object for the path
    with nd.Cell("dubins-path") as C:

        # Define important points
        radius = solution[0][2]
        current_position = start
        (sx, sy, syaw) = start
        (ex, ey, eyaw) = end
        ex = ex - sx
        ey = ey - sy

        # Draw the S, L, or R elements from the solution
        for (mode, length, radius) in solution:
            if mode == 'L':
                # Find the center of the circle
                center = (
                    current_position[0]
                    + m.cos(m.radians(current_position[2] + 90)) * radius,

                    current_position[1]
                    + m.sin(m.radians(current_position[2] + 90)) * radius,
                )
                new_position = (
                    center[0] + m.cos(m.radians(current_position[2]
                    - 90 + (180 * length / (m.pi * radius)))) * radius,

                    center[1] + m.sin(m.radians(current_position[2]
```

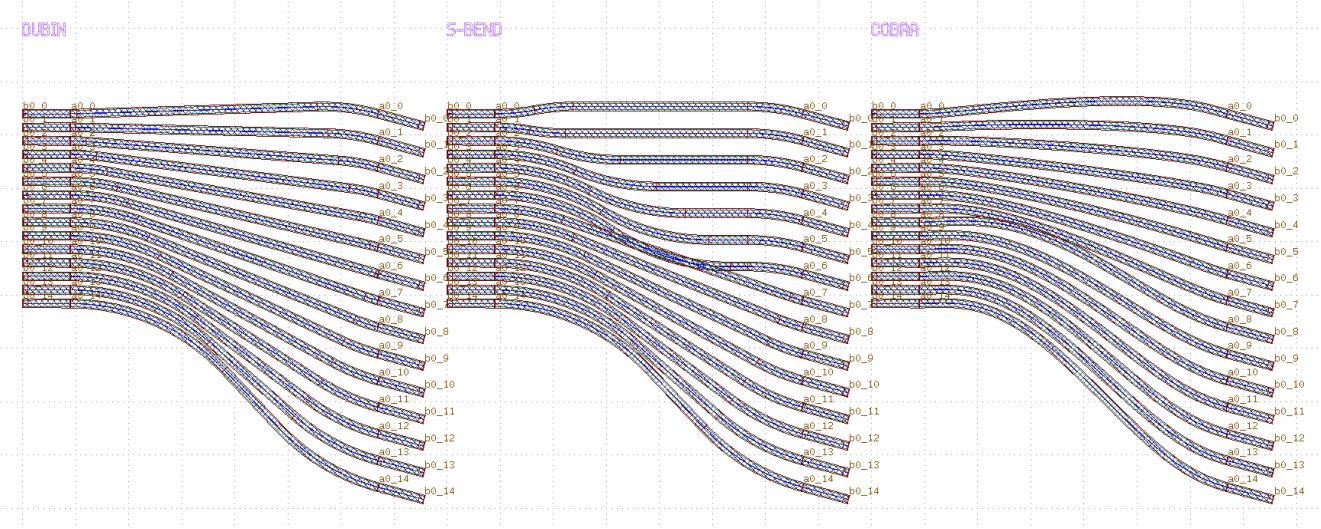
```
                - 90 + (180 * length / (m.pi * radius)))) * radius,

                current_position[2] + (180 * length / (m.pi * radius))
            )
            arc_angle = (180 * length / (m.pi * radius))
            # Change this line if you want to use the function
            # in, for example, Matplotlib
            ic.bend(radius=radius, angle=arc_angle).put()
        elif mode == 'R':
            # Find the center of the circle
            center = (
                current_position[0]
                + m.cos(m.radians(current_position[2] - 90)) * radius,

                current_position[1]
                + m.sin(m.radians(current_position[2] - 90)) * radius,
            )
            new_position = (
                center[0] + m.cos(m.radians(current_position[2]
                + 90 - (180 * length / (m.pi * radius)))) * radius,

                center[1] + m.sin(m.radians(current_position[2]
                + 90 - (180 * length / (m.pi * radius)))) * radius,

                current_position[2] - (180 * length / (m.pi * radius))
            )
            arc_angle = - (180 * length / (m.pi * radius))
            # Change this line if you want to use the function
            # in, for example, Matplotlib
            ic.bend(radius=radius, angle=arc_angle).put()
        elif mode == 'S':
            new_position = (
                current_position[0]
                + m.cos(m.radians(current_position[2])) * length,

                current_position[1]
                + m.sin(m.radians(current_position[2])) * length,

                current_position[2],
            )
            xl = current_position[0] - new_position[0]
            yl = current_position[1] - new_position[1]
            l = m.sqrt(xl**2 + yl**2)
            # Change this line if you want to use the function
            # in, for example, Matplotlib
            ic.strt(length=l).put()
```

```
            else:
                print("Something ain't right, buddy.")

            current_position = new_position
    return C

##############################################################################
# Use this when designing your PIC with Nazca!
##############################################################################

# Create Dubins path between two pins in Nazca
def dubin_p2p(xs, pin1, pin2, radius=500, width=4):
    """
    Finds and creates the shortest possible path between two vectors
    (pin1 and pin2) with a minimum bending radius,
    a so called "Dubins path". This Dubins path is made of two
    circular bends and a straight waveguide.
    Returns a cell containing these waveguides.

    IMPORTANT
    =========
    In this version, you NEED to specify to put the path at the starting
    pin so if pin1=IO.pin["a0"] you must add .put(IO.pin["a0"]).
    Else, the Dubins path will be generated correctly
    but possibly at the wrong position.

    PARAMETERS
    ==========
    xs:     Crosssection parameters.
    pin1:   The start pin to which the Dubins path attaches.
    pin2:   The end pin to where the Dubins path ends.
    radius: The minimum bending radius for the Dubins paths.
    width:  The width of the waveguides dubin_p2p creates.
    """
    # Get the pin vectors from pin1 and pin2
    START = pin1.xya()
    END = pin2.xya()

    # Change the angle for the second pin
    new_end = list(END)
    new_end[2] = new_end[2] - 180
    END = tuple(new_end)

    # Find the Dubins path between pin1 and pin2
    path = dubins_path(start=START, end=END, radius=radius)
```

```
    # Create the Dubins path with nazca using bends and straights
    return gds_solution(xs, pin1, pin2, solution=path)
```

5 Conclusion

In the figure below, a comparison between a dense array of Dubins paths and arrays using Nazcas s-bends and cobra splines is made:



As one can see, not only do other interconnects lead to over-bending of the waveguides and thus a longer path and greater losses, they are also less reliable, predictable, **often break the design rules to not violate the minimum bending radius** as is indeed the case here. They **even intersect each other!** Meanwhile, the Dubins paths behave extremely predictably. They clearly show the shortest path without unnecessary bends and they do not intersect each other which allows for much denser layouts than would be possible with the other interconnects.

There is a list of improvements one may make based on this. For one, the curvature of Dubins paths are not smooth which may lead to higher radiation losses. Still, the hours of headaches I personally avoided just by using Dubins paths are insane. It is also simply much more enjoyable to use.

That's my little tip for those working on photonic integrated circuit layouts. I hope it helps!

Citing

If so, you can cite:

```
@article{QWachDubin2024,
    author = {Quentin Wach},
    title = {Dubins Paths for Waveguide Routing},
    year = {2024}
}
```

References

1. Dubins, L. E., *"On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents"*. American Journal of Mathematics. 79 (3): 497–516, 1957 ↩

2. Wikipedia: Dubins Paths (Accessed: Feb 20, 2024) ↩
3. David A. Anisi, *"Optimal Motion Control of a Ground Vehicle"*, 2003 ↩
4. Andy G, *"A Comprehensive Step by Step Tutorial to Computing Dubins Paths"* ↩
5. Atsushi Sakai, *"Python Robotics: Dubins Path Planning"*, GitHub. (Accessed: Feb 20, 2024) ↩
6. J. D. Hunter, *"Matplotlib: A 2D Graphics Environment"*, Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007. ↩
7. Nazca Design: Photonic IC Design Framework ↩
8. GDSFactory, GitHub ↩