Master’s Thesis

Ned Charles

Design of Three-Dimensional, Path Length Matched Optical Waveguides

Abstract

A method for designing physically path length matched, three-dimensional photonic circuits is described. These waveguides, with arbitrary endpoints, were fabricated via the femtosecond laser direct-write technique. The focus is specifically on the case where all waveguides are uniquely routed from the input to output; a problem which has not been addressed to date and allows for the waveguides to be used in interferometric measurements. Two iterative design methods were created for path length matched waveguides with adequate separation in three dimensions and minimized curvature. These algorithms could be used to calculate predicted radius of curvature, bend and transition loss in the waveguides, with results confirmed by computer simulation methods. Demonstrations via interferometric methods show that the fabricated circuits were indeed optically path length matched to within 45 μm which is well within the coherence length required for typical applications, including astronomical measurements.

Acknowledgements

This thesis wouldn't be complete without the acknowledgement of my colleagues who helped me tremendously along the way. Thanks to Nem Jovanovic at Macquarie University for helping coordinate all the design, fabrication work and testing, and also for feedback on my paper and thesis. Thanks to Paul Stewart for letting me bounce ideas, keeping the lab running in top form and for putting up with my occasional cursing while sharing an office. To my advisor Peter Tuthill, for doing a thorough review of this thesis, contributing edits to it, and for coordinating the Dragonfly project. To my co-advisor, John O'Byrne, for all of the feedback despite coming late to the process, and for also reviewing and contributing edits to this thesis. To Barnaby Norris, for helping make all of the data analysis happen and for good advice on portions of the design process.

I'd also like to thank two others at Macquarie University - Simon Gross for his expertise in the fabrication of the waveguides and Mick Withford for valuable feedback on our results. Also, a special thanks to Jon Lawrence at the AAO, who got the direct write waveguide design off to running start so I could pick up where he left off.

Statement of Student Contribution

The first section in this thesis is the introduction and is a review of existing relevant technologies and is mostly the work of others and cited appropriately. The theory section is also cited appropriately, however the review of the design requirements of the project was compiled by myself and based on a review by the Dragonfly team. The third section with the design of the waveguides using a cubic spline and developing all the software algorithms is my work, except for portions of the code that are cited appropriately. An extra acknowledgement is made to Jon Lawrence (at Macquarie University at the time) for originating the design using a type of spline for the concept. The concept of the sidestep design was created by the Dragonfly team and implemented by myself. The circular arc design in Section 5 was suggested by my advisor, Peter Tuthill, but implemented by myself.

The next two sections rely heavily on the theoretical calculations made by Snyder and Love in their important textbook and are cited appropriately. The power throughput and loss testing was all performed by myself and programmed in Python. The RSoft testing in Section 8 was all performed by myself. The right angle chip was designed by my colleague Nemanja Jovanovic from Macquarie University and appropriately acknowledged. The analysis of the path length from the interferomentric fringe data was performed by Barnaby Norris and Peter Tuthill. The conclusions were written by myself. The Python code in the appendix was all written by myself, except for particular algorithms which were borrowed and acknowledged.

All waveguides were fabricated by Simon Gross at Macquarie University along with design help from Nemanja Jovanovic who also contributed many hours grinding and polishing all of the chip designs. I would like to acknowledge the help of my advisors, Peter Tuthill and John O'Byrne, for making edits to this thesis.

I certify that this report contains work carried out by myself, except where otherwise acknowledged.

Signed……………………………………………………………

Date: 27 April 2012

Table of Contents

1. Introduction

Astrophotonics is a relatively new field that exploits new advances in optical photonic technology to solve problems in instrumental astronomy. Traditionally, advances in optical astronomy have been accomplished using large bulk optic approaches and ever-larger telescopes. Advances in Astrophotonics are allowing astronomers to create fundamentally new instrument architectures which interface to the traditional bulk optic telescopes, yielding better performance while reducing both size and cost [1]. Applications include fiber optics to link arrays of telescopes into interferometers [2], fiber Bragg gratings to suppress portions of the frequency spectrum [3], multimode/single-mode photonic lanterns [4], photonic frequency combs and spectrographs [5], with new applications emerging all of the time. These devices are allowing astronomers to make better spectral and angular resolution measurements with potential cost savings in the tens or hundreds of millions of dollars. With the higher angular resolutions and precision in signal fidelity, one of the great areas in which astronomers are hoping to make significant advances is the direct optical imaging of extrasolar planets.

For a given telescope, the theoretical maximum angular resolution is given by the diffraction limit of $\frac{1.22\lambda}{D}$ radians, where λ is the wavelength of the light and D is the diameter of the telescope. Telescope sizes have been increasing in a long historical trend since their invention, and the newest generation of extremely large telescopes presently under construction will have primary mirrors over 20 meters [6]. However, for terrestrial telescopes, the actual obtainable resolution is strictly limited not by aperture but by atmospheric turbulence. One way to correct for atmospheric turbulence is with adaptive optics, technology that senses the distortions in the optical wavefront incident on the primary mirror, and then adjusts a deformable mirror to correct for distortions due to turbulence. This gives a considerable improvement to the angular resolution obtained by the telescope [7].

Another technique to reduce the turbulent effects of the atmosphere is aperture masking interferometry. This requires a mask at the pupil plane of the telescope which only allows light to pass through it at several select small holes. An example of an aperture mask is shown below in Figure 1 [8]. Such a sparse array can be constructed with non-redundant baseline spacings, a configuration which results in a much reduced exposure to the unstable phase noise which comprises atmospheric seeing. This allows for recovery of each baseline's Fourier amplitude and phase without the addition of the extra redundancy "noise" [9]. To analyze the data, the spatial frequencies are extracted from the Fourier transform of the raw data frames and analyzed by extracting complex visibilities (the Fourier amplitudes and phases). One key observable which turns out to be particularly robust against atmospheric seeing is the *closure phase*, which is defined as the summed phase around a closed triangle of baselines [10]. Closure phases from a point source reference star should all be zero, while non-zero closure phases betray the presence of asymmetric structure (such as a faint binary companion or planet).

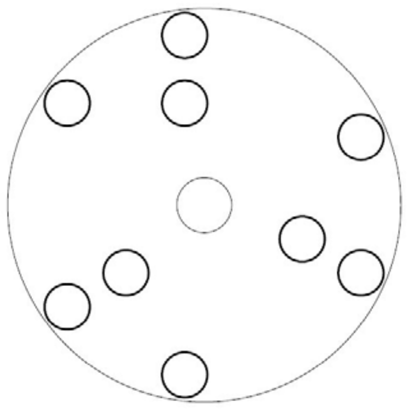


Figure 1 - Example of an aperture mask with nine non-redundant holes (Courtesy: *Tuthill et al*)

Aperture masking can be thought of as an evolution of an earlier method of speckle imaging, which takes multiple frames with very short exposure time, permitting statistical calibration of the effects of the atmospheric seeing over a large enough data sample. These frames are then analyzed for their visibilities and closure phase information with the results averaged over the entire data volume. To calibrate these data, the telescope is pointed at a known point-source reference star and complex visibilities recorded. Because these two sets of information are taken relatively close in time, the environment in the telescope, temperature, astronomical seeing, and other factors are hoped to remain constant [11]. Aperture masking interferometry enables observations near the diffraction limit for ground based telescopes, making it a powerful tool for astronomical observation. Use of aperture masking has made major breakthroughs in the observations of binary stars [12], MIRA variables [13], and Wolf-Rayet stars [14], to name a few, and continued observations on large telescopes will certainly bring more discoveries.

Another technology useful to Astrophotonics research is the single-mode optical fiber. If incoming starlight has been perturbed by the turbulent atmosphere, as is light from a stellar target without adaptive optics correction, the fiber only allows a single-moded optical wave to travel through it, filtering out all spatial disturbances in the wavefront [15]. This comes at a cost, as this smoothing of the wavefront means that all of the complex spatial modes causing the disturbances in the wavefront are absorbed, reflected, or dispersed. This can cause a dramatic drop in amplitude of the incoming signal. However, for many interferometric applications, the loss of light is tolerable and the benefit bestowed by the mode-cleaning to a single planar wavefront outweighs the lower signal levels.

The last technology that needs to be introduced for the research presented in this thesis is the creation of direct-write optical waveguides. These waveguides are created by focusing a high-power ultrashort pulsed laser into a single block of glass [16]. The energy deposited by the laser pulses creates a deformity in the glass at the spot of laser focus, resulting in a positive change in the local index of refraction. If the laser focal spot traverses through this block of glass, a tunnel of index change is created. This forms an optical waveguide that traps the light inside it in a manner entirely analogous to the more common method of a doped-core optical fiber. The light can then be inserted into one end of the formed waveguide and propagated through to the output. If the power and other pulse timing parameters are adjusted to the right levels, the created waveguide exhibits single mode properties. Such waveguides confer all of the spatial filtering advantages as single-mode optical fibers discussed above. In addition, the laser focal trajectory within the glass is able to be precisely controlled in space, meaning that with computer-controlled stages, a waveguide can be sculpted in the glass to a precision of a few micrometers, giving exquisite control over waveguide locus and length.

Direct write waveguide fabrication techniques have, at the time of this thesis, been used to create symmetrical circuits with optical path length matching [17]. These circuits have required fixed input and output points to construct the waveguides, but no design techniques currently exist that provide for flexible input and output positions. Flexible waveguide placement opens up various possibilities for design, and the optical path length matching of all waveguides allow for interferometric measurements to be made.

1.1 Dragonfly

All of the technologies discussed above are used in combination for the project that is central to this thesis: the Dragonfly instrument [18]. Dragonfly is an optical pupil remapping interferometer based on direct-write waveguides fabricated in a single block of glass. An image of the setup of the Dragonfly instrument is shown in Figure 2 below.

As shown in the diagram, light from a distant star falls on the telescope’s primary mirror. The telescope reduces the beam, which is then guided via mirrors, a lens and a prism to a segmented mirror. This segmented mirror has individual hex segments that are steerable in tip, tilt and piston under computer control. The beam from the mirror passes through a reducing telescope and is focused upon the face of a flat glass block by a lenslet array. This glass block initally had eight single-mode waveguides sculpted within it, using the laser direct write technique. The input pattern of these waveguides corresponds to the selected segments from the steerable mirror. The mirror segments are individually adjusted in tip and tilt to maximize the light coupling into the block. The light then travels through the individual waveguides, which all have a path length matched within several micrometers. When the starlight wavefronts at the inputs of the waveguides are all in phase, the path length equalization means that regardless of the path that each waveguide takes, if the difference in path length is within the coherence length of the starlight, the light at the output of all waveguides is still in phase. Thus, we may perform a coherent remapping of the input pupil to any desired output pupil.

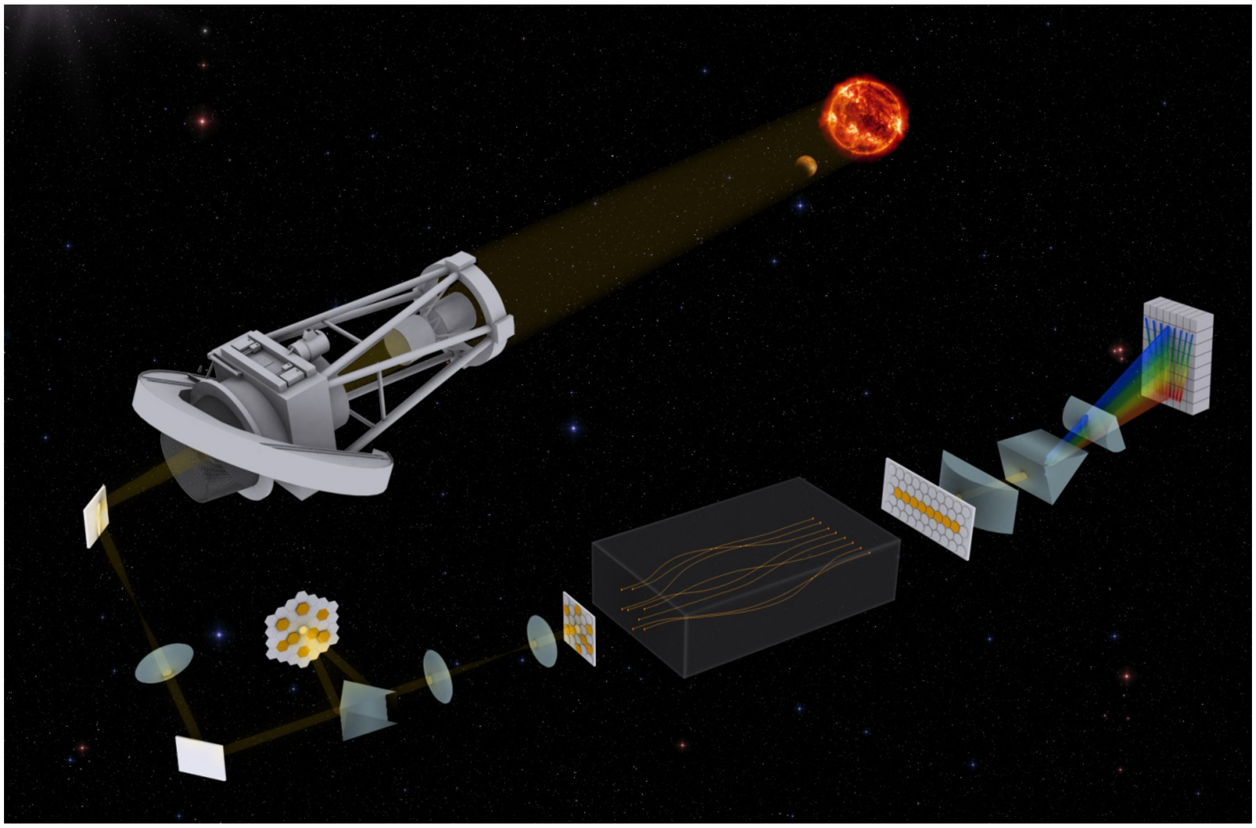


Figure 2 - Artist depiction of the Dragonfly instrument setup showing stellar target, main telescope and bulk optics that interface with the waveguide chip at center bottom of image

In the case discussed here, this remapping takes the two-dimensional input pattern of the waveguides and remaps the waveguides to a one-dimensional array. This allows the one-dimensional output beam to be put through a spectrograph, cross-dispersing the interferometric fringe pattern of the recombined outputs of the waveguides. Such a spectrograph arrangement yields information about the stellar target fringe pattern over a range of wavelengths, maximizing the science return. There is also an inverse relationship between the spectral width of the signal and the coherence length of the instrument, where a narrower signal makes for a longer coherence length. Thus, if the waveguides are path length matched within a smaller distance, a larger spectral width signal can be resolved.

The advantage of this pupil remapper, besides being able to remap from a two-dimensional array to a one-dimensional array, is that also confers the resistance to atmospheric seeing noise that aperture masking provides. In essence, this device is a photonic reformulation of an aperture mask, as it takes small portions of the incoming light pupil with little spatial turbulence over each portion. In addition, the light from each portion passes through its own single mode waveguide, which performs spatial filtering on the signal, further increasing its fidelity. Also, due to the precision of the laser fabrication system, the path lengths of all waveguides can be matched to within a few micrometers. This gives an enormous advantage over trying to match path lengths via bulk optics or optical fibers. It also gives the waveguides a small form factor, reducing the space need for optical coupling, as well as a uniform temperature over all waveguides, which removes thermal drifts within the apparatus which would otherwise hamper high precision measurements.

The design of this pupil remapper poses several challenges. This thesis describes the process of creating usable waveguide designs to perform optical interferometry. It illustrates the concepts of how to create the waveguides and achieve path length matching while simultaneously minimizing the curvature of each guide. It is also important to maximize the separation of neighbors at closest approach to minimize crosstalk between adjacent guides. This is achieved by rotation of the basic curve in three dimensions. As a final experimental constraint, the waveguides must all fit within a specified depth set by the laser focusing optics. This thesis examines the geometry of creating curved waveguides, methods for measuring the curvature value at each point in the waveguide, and usage of these curvature values to predict throughput power of the waveguides. The thesis will also discuss simulation methods to confirm power predictions, and the detailed discussion of several designs which were analyzed to compare theoretical, simulated, and physically measured results.

2. Theory

2.1 Optical Waveguides

The creation of optical waveguides relies on the basic optical principle known as Snell's Law, which relates the behavior of light travelling through a medium to a property known as the index of refraction [19]. The index of refraction refers to the speed of light travelling in that medium, with its value

$$\mathrm{n} = \frac{\text{Speed of Light in vacuum}}{\text{Speed of Light in material}} \tag{1}$$

Snell's Law states that when light passes from one medium to another, there is a relationship of the angle of incidence of the light and the angle of transmission. The equation for Snell's Law is

$$\mathrm{n_i} \sin\theta_\mathrm{i} = \mathrm{n_t} \sin\theta_\mathrm{t} \tag{2}$$

Normally, an incident light ray will pass from one medium to the other and be deviated by a specific angle. However, if $n_i > n_t$, it is possible at certain angles of incidence for the light to be completely reflected back into the incident medium. This "total internal reflection" occurs at angles equal to or greater than the critical angle, which is found by the equation

$$\theta_\mathrm{c} = \arcsin\left(\frac{\mathrm{n_t}}{\mathrm{n_i}}\right) \tag{3}$$

If a light wave is traveling in a specific medium with an index of refraction higher than the surrounding medium, at an angle of incidence greater than the critical angle, it will reflect completely back into the medium. An incident light beam entering, greater than the critical angle, will continually bounce off the surface boundary between the two media, while propagating down the length of it. This total internal reflection allows for the creation of optical waveguides, which is the main focus of the material in this thesis.

2.2 Interferometry

Also central to the work here, is the concept of interferometry, which is based on the principle of superposition where two waves can be summed together. When the two waves are superimposed on each other, the resulting signal can have greater amplitude (constructive interference) or lesser amplitude (destructive interference) depending on the phase shift between them. This principle is best illustrated with Young's double-slit experiment [20]. In this experiment, light from a point source passes through two slits and is projected onto a screen. On the screen, a fringe pattern can be observed from the light passing through the slits and undergoing interference. The light produces either constructive or destructive interference, depending on the distance from each slit at a particular point on the screen. The spatial frequency of the observed fringes will depend on the distance between the two slits. An equation relating the spatial frequency and the distance between slits can be written as

$$\Delta\Theta = \frac{\lambda}{b} \tag{4}$$

Where b is the distance between slits, and λ is the wavelength of the signal. This equation is illustrated in the left diagram in Figure 3 below.

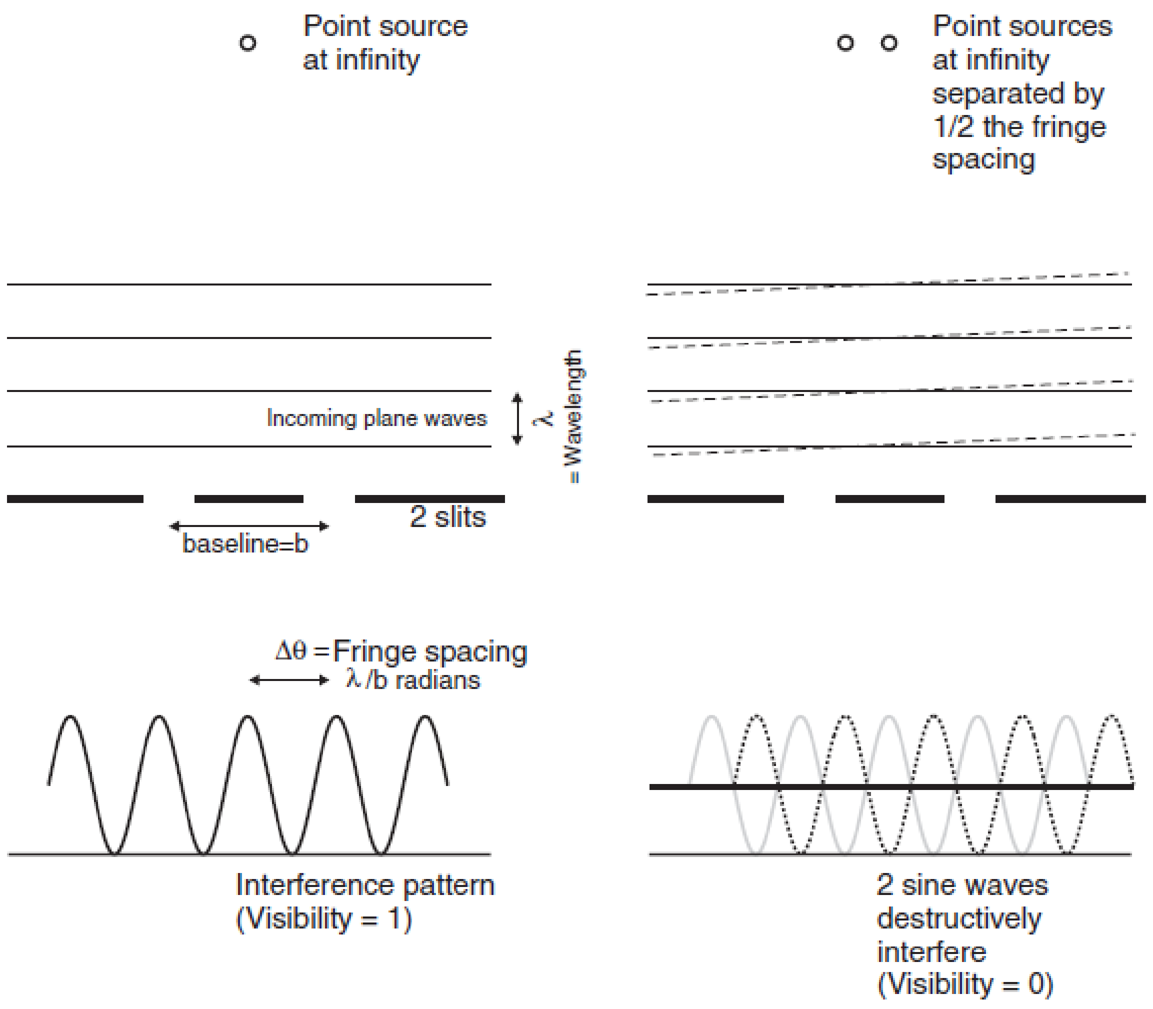

Point source at infinity
Point sources at infinity separated by 1/2 the fringe spacing
Incoming plane waves
λ = Wavelength
2 slits
baseline=b
Δθ = Fringe spacing λ/b radians
Interference pattern (Visibility = 1)
2 sine waves destructively interfere (Visibility = 0)

Figure 3 - Illustration of double slit experiment for one and two point sources, illustrating properties of interference (courtesy: *Monnier*)

As the right hand diagram shows, if a second point source of equal intensity is added at half the distance of the fringe spatial frequency $\frac{\lambda}{2b}$, the waves will combine 180° out of phase at the observing screen, leaving a constant, non-zero intensity, but no visibility. If the two slits are considered to be two input apertures of an interferometer, separated by a baseline b, and recombined via paths of identical optical path distance, then interference fringes on a stellar target can be observed. Information can be obtained about the size of a single star, the distance between binary stars, or the existence of a luminous object within close proximity to the star. The angular resolution of an interferometer is given by $\frac{\lambda}{2b}$, in contrast to the single telescope diffraction limit $\frac{1.22\lambda}{D}$ discussed previously. The largest existing optical telescope has a diameter of 10 meters. In contrast, an interferometer can be fabricated with two points placed much farther apart (several hundred meters), giving a much higher intrinsic resolving power.

2.3 Aperture Masking and Closure Phases

Using these concepts from optical interferometry, aperture masking theory can be discussed. Considering the aperture mask depicted in Figure 1, if light from a stellar target is apodized with this on the mask, every pair of holes on the mask will now act like an interferometer with a specific baseline fringe frequency and phase, the latter represented by φ [21]. Starlight passing through the atmosphere will also carry accumulated perturbations due to refractive index fluctuations above each sub-aperture. These perturbations manifest themselves as observable phase delays in the signal, represented by ε. To remove these atmospheric phase delays from the signal, a quantity known as the closure phase is used. If three of the holes in a particular mask form a closed triangle, a closure phase can be derived. Fringe phase formed on a baseline spanning holes 1 and 2, along with the atmospheric phase delays of each hole, can be written as

$$\Psi_{12} = \varphi_{12} + \varepsilon_1 - \varepsilon_2 \tag{5}$$

The equations for the other two baselines can be written as

$$\Psi_{23} = \varphi_{23} + \varepsilon_2 - \varepsilon_3 \tag{6}$$

$$\Psi_{31} = \varphi_{31} + \varepsilon_3 - \varepsilon_1 \tag{7}$$

The closure phase can then be written as the sum of these three signals. By inspection, it can be seen that the atmospheric error terms will all cancel out when summed, with a closure phase, Φ, of

$$\Phi_{123} = \Psi_{12} + \Psi_{23} + \Psi_{31} = \varphi_{12} + \varphi_{23} + \varphi_{31} \tag{8}$$

Thus, when using the closure phase to get information about the celestial target, the phase delay errors introduced by the atmosphere can be ignored to first order. Each unique group of three holes will give its own closure phase, which is also the phase of the bispectrum, the complex quantity about the interaction between these individual frequency components. These bispectral values can then be analyzed to deliver information about the stellar target. Each hole-pair can be represented in an aperture mask as a vector with distance and direction. If each hole-pair has a unique distance and directon, and therefore spatial frequency, the mask is said to be non-redundant, with each baseline visibility being unique.

Imaging using aperture masking interferometry requires many repeated exposures or frames over small exposure times. The small exposure minimizes the atmospheric disturbances in time. To obtain information about the object, the dark frame intensities are subtracted and the Fourier transform taken for each frame. Summing these transforms over all frames gives an image of the power spectrum. This power spectrum then gives information about the intensity of the stellar target over wavelength. An example of a power spectrum is given later in Figure 61.

2.4 Single-Mode Waveguides

Single-mode optical fibers have achieved widespread use for telecommunication applications, because the physical structure of the fiber allows only the primary mode of a beam of light to propagate within it. This retains the fidelity of a light pulse traveling within the fiber, allowing the signal to be sent longer distances than in a multimode fiber. This fundamental mode propagation is what makes single-mode fibers useful in Astrophotonics applications, as any incoming light signal from an astronomical target is spatially filtered. This does come with a cost, as only allowing the fundamental mode to propagate leads to a considerable loss of light. A recent application of using single-mode fibers for spatial filtering is the FIRST project [22], which is similar to the Dragonfly project in that it also performs aperture masking AND pupil remapping. The same interferometric measurements as the Dragonfly instrument can be made, with the addition of better throughput through optical fibers and ease of adding more waveguides. This does come with a drawback in that path length matching is harder to achieve than direct write waveguides as it is difficult to add and subtract lengths of fibers alone on the scale of micrometers.

2.4.1 Direct Write Waveguides

As laser technology improved over the past few decades, the ability to create high powered laser pulses of the order of several femtoseconds has allowed for improvements in laser fabrication technology [23-25]. If a femtosecond laser is focused in a block of glass, the energy deposited at the point of focus can cause subtle structural changes which affect the refractive index. Because the pulse is short and tightly focused, the glass is only modified in a narrow region several micrometers wide, and the surrounding glass is not be affected. For the types of glass considered here, a positive change in the index of refraction is produced, resulting in a natural pathway to fabricate a waveguide. If the glass is placed on a mechanical stage and moved according to a specific locus of coordinates, the laser then traces a path in the glass, creating an optical waveguide. This allows for an optical waveguide to be created in any shape in two or three dimensions, giving a flexible framework for optical design [26].

To date, any path length matched designs in a single photonic circuit, two or three dimensional, have used route symmetry, where the paths are mirror images of each other [27,28]. The Dragonfly application is unique in that any design method must have the ability to cope with flexible placement of endpoints to meet the specific pupil remapping demands. Likewise, there has been at least one design [29] where three-dimensional waveguides have been created with flexible endpoints, but these designs don't have path length matching. This illustrates the complexity that the design of Dragonfly will involve.

2.5 Simulations of Optical Waveguides

To predict the optical performance of any given waveguide, the software design suite RSoft was used. RSoft has a CAD design environment, which allows for waveguides to be designed in two or three dimensions. When a waveguide is designed, the BeamPROP tool allows for simulation scenarios to be created. This entails setting up factors such as wavelength of the light, index of refraction change of the waveguide and waveguide propagation values. Calculating the intensity of light is extremely difficult due to the complex Helmholtz equation [19]. The BeamPROP tool works with the computational technique known as the Beam Propagation Method. This is an approximation method that allows for a massive reduction in the calculations that need to be made, by use of the slowly varying envelope approximation (SVEA) and only accounting for the local waveguide conditions. Using this method only requires the distribution of the refractive index in three (or two) dimensional space, n(x,y,z) and the input field u(x,y) at the start of the waveguide. The software then calculates the wavefront at any point z > 0 by the equation

$$\frac{\partial u}{\partial z} = \frac{i}{2\bar{k}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \left(\bar{k}^2 - k^2\right)u\right) \tag{9}$$

where u is a slowly varying envelope function and $\bar{k}$ is the reference wavenumber [30].

RSoft also has a specific work-around module for modeling curved waveguides by using a simulated bend. As a waveguide bends in space, the geometry of the above BPM equation becomes more complicated as the wavefront shifts with respect to the initial direction of propagation, z. The simulated bend technique significantly improves computation time by calculating the wave in the z direction, and approximating the bend in the waveguide as a change in the index of refraction. The basic equation governing the simulated bends technique is found in *Snyder and Love* [31] and is shown below:

$$n_e^2 = n^2(r) + 2n_{co}^2\left(\frac{r}{R_c}\right)\cos\Phi \tag{10}$$

Where n_e is the effective index of refraction, n_{co} the index of refraction in the waveguide core, n(r) is the index of refraction profile of a straight waveguide, r and Φ are the polar coordinates of the waveguide, translated from x and y, with respect to the z axis and R_c is the radius of curvature of the waveguide. With this approximation, calculation times for some of the three-dimensional waveguides in this thesis were reduced from over eight hours to less than two minutes.

2.6 Design Requirements

For the initial prototype of the Dragonfly project, to keep the form factor small while maximizing the Fourier coverage obtained, a design with eight waveguides was chosen. This would give 56 (select 3 out of 8) possible closure phase triangles. In the initial design process, along with the analysis of physical form factor, fabrication process, and desired input and output points, there were eight physical design requirements to be considered in this design. These requirements were as follows:

- All different waveguide path lengths must be matched
- Flexibility in placement of input and output coordinates
- Waveguides orthogonal to substrate surface at entrance and exit
- Waveguides follow a continuous, smooth locus
- Waveguide curvature minimized
- Adequate spatial separation between adjacent waveguides to avoid crosstalk
- All guides must occupy a limited depth set by the laser write
- For waveguides which cross more than once when viewed from above, the depth order must be preserved (see below).

Each of these design requirements will now be explained individually and the design process to meet each requirement will be discussed.

2.6.1 Path Length Matching of Waveguides

The most critical design requirement for the Dragonfly design is the matching of the path length among all waveguides. If a plane wave hits the inputs of all eight waveguides instantaneously, then the light traveling in all eight waveguides will be in phase after any fixed distance. If one of the waveguides has a considerably longer path length than the others, the light in that waveguide will take longer to get to its output, and will therefore be out of phase. When this path length mismatch grows large enough, the interference pattern will lose contrast and visibility loss will be severe. This maximum difference between any two paths is known as the coherence length, the equation for which is:

$$L = \frac{\lambda^2}{\Delta\lambda} \tag{11}$$

where L equals the coherence length, λ is the center wavelength of the optical signal and Δλ the full width half maximum spectral width of the optical signal [32].

For the first prototype of Dragonfly, the waveguides were fabricated to be optimized at a center wavelength of 1550 nm. This wavelength was chosen due to the high availability of test equipment at that wavelength due to it being in the optical communication C Band, the demonstrated ability of waveguides to be fabricated to transmit at 1550 nm using Eagle 2000 glass [33], longer atmospheric coherence time and atmospheric windows, and the fact that adaptive optics systems on telescopes operate better in the infrared region due to response times. Typical astronomical observations operate with fractional bandwidths between 1 and 15 %, giving a range for Δλ of 0.015 – 0.233 µm. Plugging these numbers into equation 11 gives a coherence length range of 10 - 160 µm, meaning that the difference in path length among all of our waveguides should be less than that range, depending on the target. Anticipating the small variations in physical path length that can occur during the fabrication process, a design goal of matching the mathematical locus of all waveguide path lengths to within 0.1 µm was chosen for the initial prototype. This length falls well within the needed coherence length, ensuring phase coherence over all waveguides. This is strictly for the calculations set by the algorithm and does not include the errors in fabrication, which may be much greater, hence the large margin.

2.6.2 Positioning Input and Output Coordinates

The next design requirement is the ability to place the input and output coordinates any desired location on the chip face. The ability to have flexible placement of the input and output points of the waveguides allows for variations in the sizing of the coupling lenslet arrays and gives the ability to couple into other optical chips with various spacing. This feature also enables the remapping of a redundant input pupil into a non-redundant output pupil.

The design also needed to meet the specification that each waveguide will have a specific input point matched to a specific output point. In the designs here, however, the primary concern is the layout at the input and output, with the mapping of which input leads to which specific output driven by convenience. This allows for added design flexibility in creation of the waveguide paths, which will be discussed later.

2.6.3 Orthogonal Endpoints

Another design requirement related to the creation of the optical waveguides is that each track must couple to both the input and output faces at a 90° angle (orthogonally). The main reason is to ensure phase coherence across all waveguides and to ensure maximally efficient injection of the beams. As shown in Figure 2, the light from the telescope falls on our steering mirror as a flat wavefront. This steering mirror then allows individual sections of the wavefront to be steered in tilt onto its respective input on the optical chip, via an individual lenslet in the array. Each waveguide input point is chosen so that the light falls on the optical chip as a mirror reflection, giving a relatively flat wavefront at the input to the chip. Because the input wavefront is flat, the input points of all waveguides should lie on a flat face, so that the wavefront arrives at all waveguide inputs at the same time. If the input face of the chip is at an angle, the light wavefront will hit the waveguides' inputs at different times, creating a phase slope across the array. The input and output faces of the optical chip are also polished as flat as possible, to prevent phase differences at the chip faces.

The other reason that the input and output points of the waveguide should be orthogonally is to maximize light coupling into and out of the chip. The incoming light wavefront is coupled into each waveguide via lenslets. These lenslets are placed in the longitudinal axis at a position where the focal point of the lens will fall at the input of the waveguide. When the focused light enters the waveguide, the propagating mode of the guide is excited by the incident radiation field. The mode does not instantaneously relax to the guided form, and therefore requires a specific distance for this to happen. So, the waveguide needs to have a straight section at the input to allow the light to settle into its propagation mode for the waveguide. If the light is not allowed to settle, but is subjected to a bend, more loss will occur than if the light was in the correct mode. Simulations were performed using the optical waveguide simulation tool, RSoft, and based on the results of these simulations, it was determined that each waveguide should have a straight section of 1 mm at the input before introducing any curvature. This 1 mm straight section is also required at the output, since the mode needs to settle to maximize coupling into the output lenslet.

For the initial Dragonfly prototypes, the input and output coordinates of the waveguides were on opposite faces of the optical chip. This means that the direction of the input and output straight sections will be parallel, even though they may be at different transverse (x,y) coordinates. The flexibility for designs can be incorporated for input and output coordinates that end up at different angles to each other (e.g. 45°, 90°, etc.), but such designs were not examined here.

2.6.4 Continuity of Waveguides

To maximize propagation of light through a waveguide, the entire trajectory needs to have a continuous locus. Any abrupt change, lateral shift, or gap between sections would cause major loss of light. Therefore, whatever mathematical function or computational algorithm used to create the waveguide must create a continuous curve.

2.6.5 Waveguide Curvature

A design factor closely related to continuity is the curvature of the waveguide. A continuous curve can be approximated by a finite (but large) number of discrete segments. Any two segments of this three-dimensional curve will lie on the same plane. If these segments are represented as three-dimensional vectors, and the direction of these vectors change in space, there will be a curvature value associated with these segments. This is given by a value known as the radius of curvature, which is the radius of a circle, known as the osculating circle, which best approximates the curve at a specific point.

The importance of this radius of curvature measurement, and how it pertains to a waveguide, will be extensively discussed throughout the rest of this thesis. A method for measuring the discrete radius of curvature of a given waveguide will be discussed in Section 3.5. The effect this radius of curvature value has on the optical power throughput of a waveguide will be analyzed in-depth in Section 6. With the ability to correlate curvature with optical power throughput, a design goal can be set for radius of curvature. For the waveguides to have enough output power to perform closure phase measurements, the radius of curvature should be large enough such that the total output power of each waveguide is greater than 25% of the measured input power.

2.6.6 Spatial Separation of Waveguides

If light is traveling in an optical waveguide, the region surrounding the core of the waveguide will be occupied by an evanescent electromagnetic field. This field extends outward in all directions orthogonal to the longitudinal axis of propagation, and exponentially decays as the distance from the waveguide increases [31]. If two waveguides are running parallel to each other, and are sufficiently close to allow overlap of their evanescent fields, the light from one waveguide can couple into the other and vice versa. In some applications, this is desirable, but for these remapping chips, it is to be avoided, as crosstalk between waveguides will cause degradation of phase information.

Another reason to keep sufficient spatial separation between neighboring waveguides is to avoid physical defects during the waveguide creation process. When the femtosecond laser writes its waveguide into the block of glass, it creates a permanent physical change in the index of refraction. If two waveguides are fabricated too close to each other, the process of writing the second can be impaired by the presence of the first, causing defects which could severely inhibit light propagation.

To avoid these factors, testing in the laboratory revealed that with these input and output spacing requirements, any two waveguides should be separated by 30 µm or more. Since the radius of the physical waveguides is about 5 µm, this means that the centers of any two waveguides should be separated by a distance of 40 µm or more. This guideline was used in the design of all waveguides discussed in this thesis.

2.6.7 Physical Chip Write Depth

The waveguides for the Dragonfly prototype, as mentioned earlier, are fabricated by a high powered femtosecond pulsed laser creating a physical positive index of refraction change in a glass chip. To create the waveguide, the laser focus is translated along the longitudinal or z axis as shown in Figure 4.

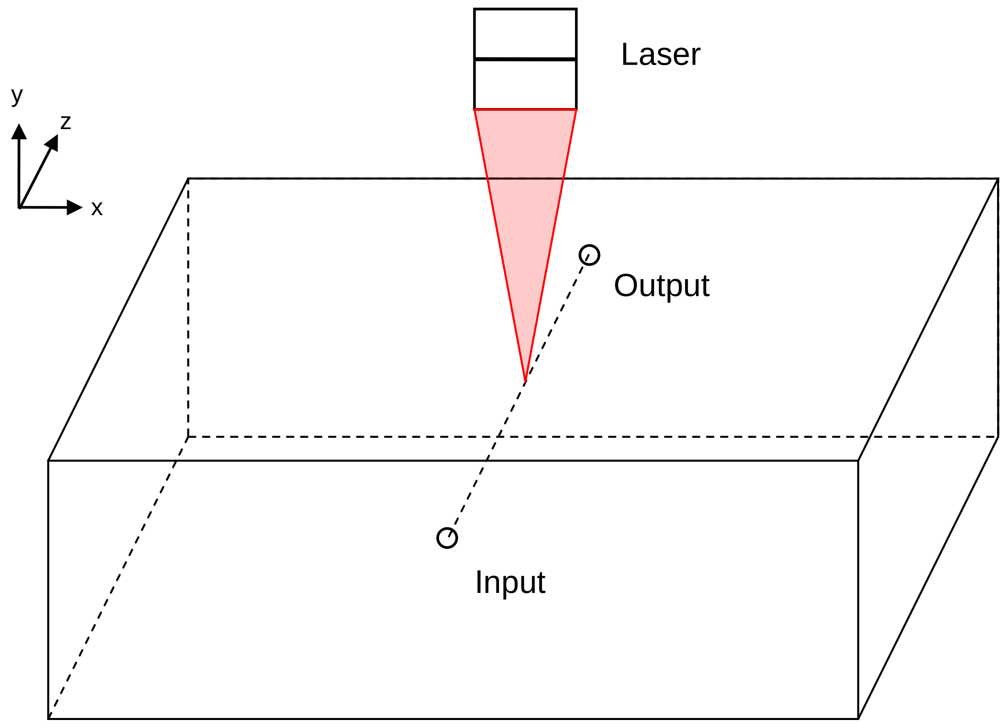


Figure 4 - Illustration of three-dimensional coordinate system for chip design and orientation of laser for waveguide fabrication

The laser then writes in incremental 25 µm steps in the z direction. At each step, the laser positions its focal spot based on the lateral x and y axis coordinates, giving a three-dimensional position array for the entire waveguide. The laser control software calculates the power settings to best fabricate the waveguide based on several factors. The laser is coupled to the glass via an index-matched oil immersion objective. When testing various writing depths on the waveguide, two physical factors were noted. Firstly, the laser requires a certain level of power focused at the intended x,y,z coordinates. As the coordinates are deeper or farther away from substrate surface, eventually a depth is reached where the laser focus does not yield sufficient energy density to precipitate the required index change in the glass. This means that there is a maximum depth of requirement for writing successful waveguides. The second depth requirement occurs at the other extreme when the laser is instructed to write at a very shallow depth near the surface of that face of the glass. High laser energy densities occur at a focal location close to where the oil couples the laser into the glass surface. A limit is reached as the heat absorbed by the oil causes it to begin to boil, which causes dramatic degradation in the laser beam quality. This gives a minimum depth that the waveguide can be written.

Based on testing the minimum and maximum writing depths, it was determined that the minimum depth from the surface that a waveguide could be written was 100 µm, while the maximum depth was measured at 450 µm, giving a depth restriction of 350 µm. This means that the maximum depth (or y coordinate) of any waveguide and the minimum depth of any waveguide must be within 350 µm of each other.

2.6.8 Waveguides crossing vertically

The previous section examined the requirements on the writing depth imposed by the laser. Because the ability to put the input and output coordinates of all waveguides at more or less arbitrary location in space is needed, it is inevitable that when looking from the top down perspective of the laser, two waveguides will cross each other. This causes no problems as long as the waveguide that crosses underneath is written first. If the upper waveguide is written first, when the laser tries to write the lower waveguide, the beam will pass through the top waveguide causing an aberration which will mar the focal spot at the lower guide. So, the waveguides need to be written in order from the bottom up. A unique problem can arise from this, however, in the process of meeting the other design requirements, a waveguide could cross one waveguide underneath and then at a later crossing, pass the same waveguide over the top. This then makes it impossible for the laser to write two such waveguides, as no matter what order they are written, the laser will pass through one already written waveguide while trying to write the second. This issue gives another design requirement that must be accounted for, and hereafter refers to configurations which do not meet this requirement as *entangled*.

3. Design of Three-Dimensional Waveguides

3.1 Physical Design of the Remapper

The steerable micromirror array used was an IRIS AO PTT111, which provided the fine-tuning ability to guide portions of the pupil to different positions on the face of the optical remapper block, has 37 individual micromirrors with the center of each mirror 606 µm from the neighboring mirrors. Using this design factor, the positions of the eight mirrors used were selected to maximize the spatial Fourier information recovered. After rescaling by beam compression optics, this pattern was reimaged on to the microlens array and so onto the face of the remapper chip. Thus, the centers of the waveguides were placed where the focal point from each mirror fell. This gave the input waveguide pattern, overlaid on the micromirror array for reference, as shown in Figure 5.

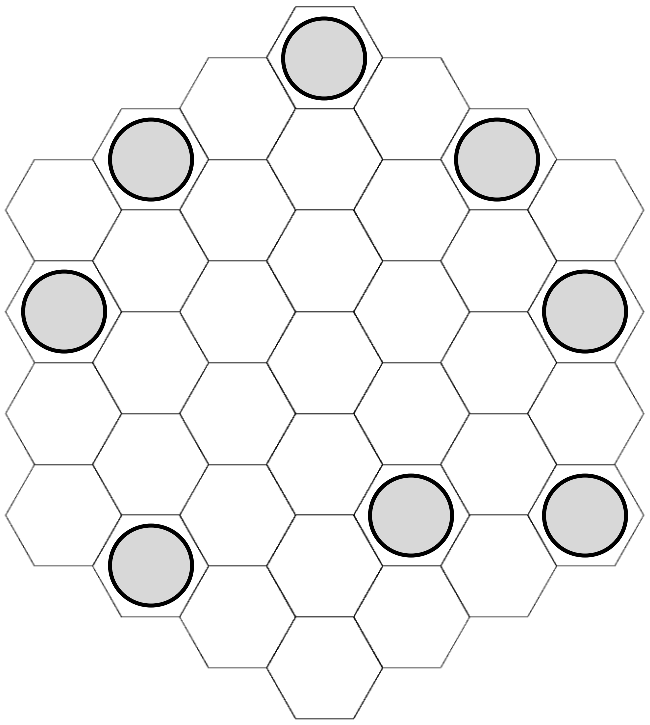


Figure 5 - Image of 37 Segment MEMS Array with the selected eight waveguides utilized in the prototype remapper chip indicated.

The output spacing of the remapper chip was chosen to be a linear array with adjacent guides separated by 250 µm for several reasons. This was found to offer convenient coupling to available lenslet arrays and downstream optics, and is also an industry standard which enables injection into planar photonic chips fabricated by photolithography [34]. The center of the eight outputs was at the same origin in the x and y axis as the center of the input waveguide positions. With the input and output positions chosen, the baseline requirement for the optical chip design looked like the image shown in Figure 6.

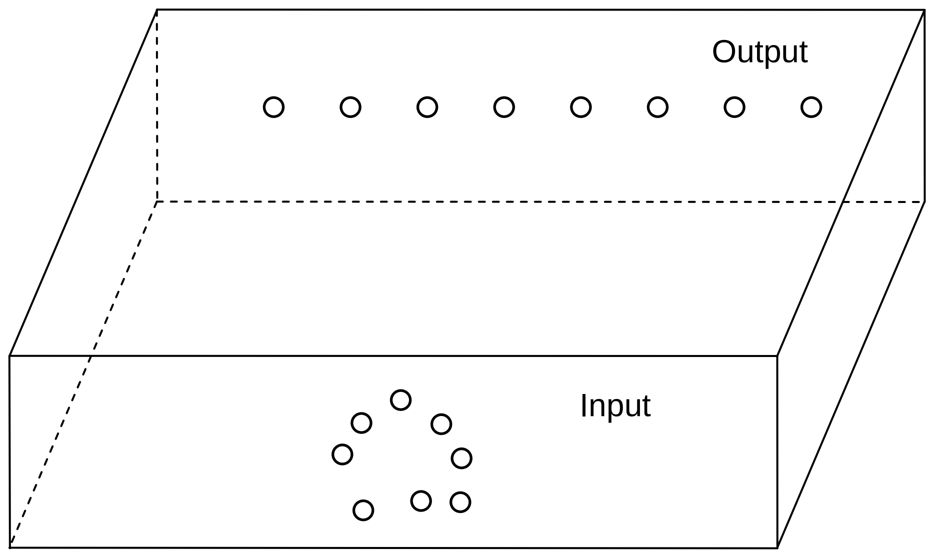


Figure 6 - Image of desired input and output points on a prototype optical chip

The length of the glass chip was selected to be 30 mm. This gave the best compromise between high curvatures due to the chip being too short, and too much absorption loss due to the chip being too long. All of the physical factors needed to create the waveguides were selected. Next, the method for creating the curved waveguides will be discussed.

3.1.1 The Interpolating Cubic Spline

The interpolating spline is a piecewise polynomial function that creates a curve based on a series of points or knots. The curve must pass through all knots specified and the function describing the curve between any two adjacent knots can be different over the entire length of the curve, giving the spline the flexibility to create any kind of curve needed.

After researching the various types of interpolating splines that one could use, the cubic interpolating spline was chosen for the waveguide creation algorithm. This is a third order polynomial function that gives us a lot of flexibility without the extra mathematical complexity that higher order equations would entail. To characterize a spline mathematically, the points that the curve should pass through are specified as (x_i,y_i) where i = 0,1,2…n for the function y = f(x). This gives a total of n knots with n – 1 line segments. With the knots selected, the equation for the curve between any two points is

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i \quad \text{for } x \in [x_i, x_{i+1}] \tag{12}$$

With this equation the entire spline can be defined in space, as the value of S(x) for any value of x can be found by using the specific spline equation between the two knots that bracket the interval containing x. However, description of the spline is not complete, as a,b,c and d are all unknown coefficients that need to be solved for. To help solve for these coefficients, a few additional numerical conditions that specify the creation of a cubic spline are used. One of the requirements for a cubic spline, as used for the waveguide design here, is that the entire spline function needs to be continuous. So, at each knot, the two spline equations joined at that knot must have the same value, i.e. $S_i(x_{i+1}) = S_{i+1}(x_{i+1})$. Another condition for the construction of a cubic spline is that it be as smooth as possible, meaning that at each knot joining two spline sections, we want the first and second derivatives to be equal to each other, so $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ and $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$. Equal first derivatives ensure that the slopes of the curves don't change abruptly at a joining knot. Equal second derivatives ensure that the curvature is smooth at a joining knot. Both of these were design requirements were enforced for the waveguides.

With the requirements that the spline function and its first two derivatives be equal at the joining knots, all requirements needed to solve all n - 1 spline equations were met, except for the conditions at the two end knots of the spline. At the end knots, the second derivatives can be specified to be equal to zero, meaning they have no curvature at those points, which is known as a natural spline. The other condition required specifying that the end segment has a particular slope, or first derivative, so that the spline curve will terminate at a particular angle. Because there is a requirement already discussed that the end points of the waveguide end orthogonally to the block surface, the second condition in the construction of the waveguides arises naturally from the basic problem statement.

3.1.2 Using the cubic spline to construct a waveguide

Thus far, the locations of the endpoints in three-dimensional space are known, and the requirement that the waveguide curves terminate orthogonally to the end face of the chip has been met. If the axis between the two endpoint faces is set as the z axis, the x,y location of the finishing point could be projected on to the front face and compared to the (x,y) location of the starting point. With the orientation shown in Figure 7 below, it was found to be convenient that the x and y points in Cartesian space could easily be translated to cylindrical coordinates of r and Θ.

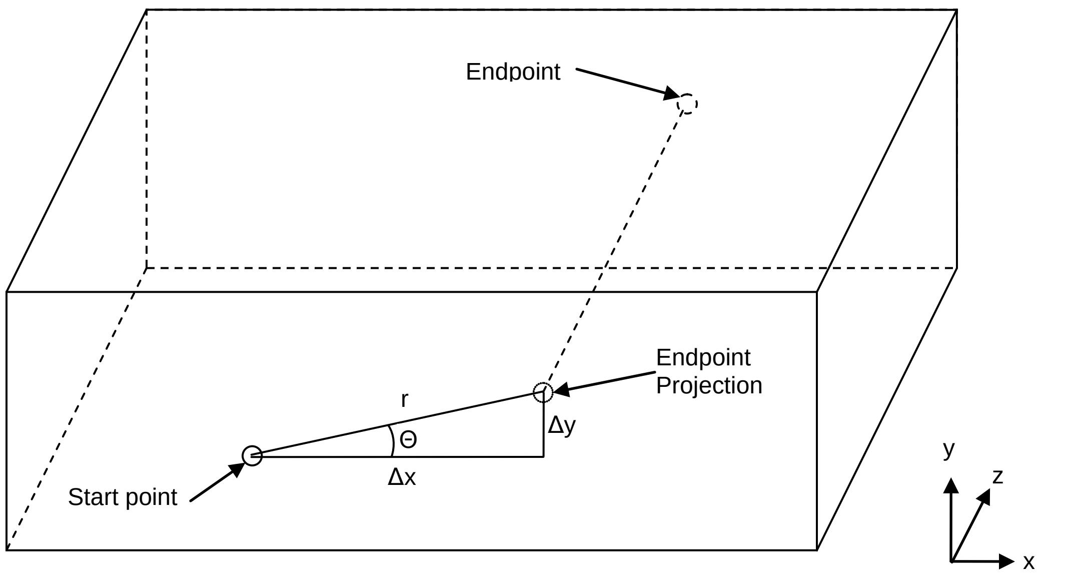


Figure 7 - Diagram of determining lateral offset for an input/output point pair via endpoint projection

This projection along the radial line segment meant that simpler two-dimensional splines could be used to create a three-dimensional waveguide. The lateral portion of the waveguide is the cylindrical projection of r at angle Θ and the longitudinal portion of the waveguide is the z axis, with r = S(z) being the spline function. After the z and r coordinates are solved for, the relationship between (x,y,z) and (r,Θ,z) could be used to translate the spline curve back to Cartesian coordinate in three-dimensional space.

With the ability to translate a two-dimensional cubic spline curve into three-dimensional space, the number of knots needed for the curve could be defined. The location of the z and r knots were defined by the positions of the start and end points of the waveguides, giving two knots. To make the math as simple as possible, only one additional knot was needed. To also minimize the curvature, the middle knot was placed at the halfway point of a line connecting the two endpoints. This gave a symmetrical curve about z and r. The other requirement that needed to be satisfied was the endpoint conditions. Because the curve should end parallel to the z axis, the slope of the line must be zero, so f'(z) = 0.

As a shorthand, until conversion back to three-dimensional (x,y,z) coordinates, x and y will represent the x and y of the two-dimensional spline function (called z and r above). To solve the integration coefficients for equation 12, the Lagrange interpolation formula was needed. Following the math detailed in Section 5.3 of *Numerical Methods in MATLAB* [35], the following equation is derived:

$$h_{i-1}m_{i-1} + 2(h_{i-1} + h_i)m_i + h_i m_{i+1} = 6\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right) \tag{13}$$

$$\text{for } i = 1, 2, \dots, n-1$$

where $h_i = x_i - x_{i-1}$ and $m_i = S_i''(x_i)$, the second derivative at point i

What this reduced equation gave was a way to solve all of the second derivatives based on the values for h and y, which were both known. Since there can be multiple sets of the equations, it was possible to solve for multiple knots. For equations with several knots, these could be solved using a matrix solution of **[h]*[m] = [u]**. For waveguide creation, there were only three knots, meaning that for equation 13 above, only one equation was needed for i = 1:

$$h_0 m_0 + 2(h_0 + h_1)m_1 + h_1 m_2 = 6\left(\frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0}\right) \tag{14}$$

This gives the known x and y positions of all three knots, and the requirement that the first derivatives of the end points should equal zero. The values for m_0 and m_2 depend on the end point constraints. Also from the above reference we get the equations for the end point constraints:

$$m_0 = \frac{3}{h_0}\big(d_0 - S'(x_0)\big) - \frac{m_1}{2} \tag{15}$$

$$m_N = \frac{3}{h_{N-1}}(S'(x_N) - d_{N-1}) - \frac{m_{N-1}}{2} \tag{16}$$

Substituting the values of m_0 and m_2 from these two equations into equation 14 above meant m_1 can be solved. With m_1 calculated, that value could be plugged back into the equations for m_0 and m_2 and could be solved. This gave all of the values for the second derivatives at each of the three knots. Section 3.3 from *Numerical Engineering Methods in Python* [36] gave the following equation for the spline curve

$$S_i(x) = \frac{m_i}{6}\left(\frac{(x - x_{i+1})^3}{x_i - x_{i+1}} - (x - x_{i+1})(x_i - x_{i+1})\right) - \frac{m_{i+1}}{6}\left(\frac{(x - x_i)^3}{x_i - x_{i+1}} - (x - x_i)(x_i - x_{i+1})\right) + \frac{y_i(x - x_{i+1}) - y_{i+1}(x - x_i)}{x_i - x_{i+1}} \tag{17}$$

This equation gave the ability to solve the value of the spline curve for any value of x. For the three knot design here, there were two equations to be used. If x fell between knot 0 and knot 1, then all values need to be solved in equation 17 with i = 0. If it fell between knot 1 and knot 2, equation 17 with i = 1 was used.

To fabricate a spline curve for the waveguide design, an arbitrary start point at x and y coordinates of (0,0) was set. Since this is at the start of the waveguide, the value for z was set at zero. The end point for this example had a value for x and y of 500 and 0 µm respectively and as previously mentioned, the length of the physical chip was set to be 30000 µm (30 mm). Because two 1 mm straight sections were needed either end, the total length in the z axis devoted to curve design was 28 mm. The value for the lateral offset of the waveguide was Δx or 500 µm. The last required specification was how many points were needed to interpolate in between each knot in the z axis, which was used as x in equation 17. The laser fabrication software was calibrated in this instance to accept a point every 25 µm in length. Meaning there were 28000/25 + 1 points or 1121 points that needed to be calculated.

The algorithm to create the spline was written using the Python language and based on the equations in this section. The code for this algorithm and the rest of the program, containing the various algorithms in this thesis, can be found in the Appendix. It first solved for the second derivative values using all of the information from the knots. It then took these second derivative values and plugged them into equation 17 above, evaluating the value of $S_i(x)$ based on where the value of x is. When the routine completed, the endpoints were extrapolated out for 1 mm on both ends, with the resulting curve shown in Figure 8.

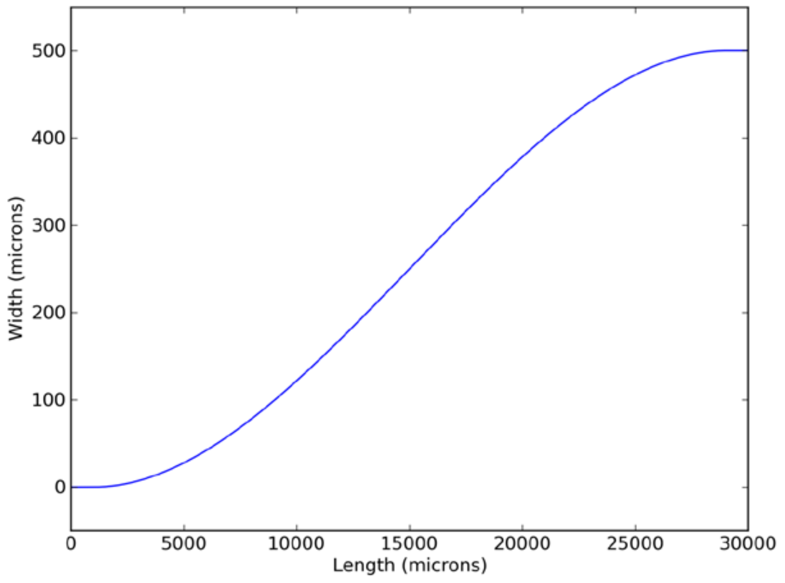


Figure 8 - Graph showing a simple planar waveguide design based on a cubic spline

3.1.3 Path Length Matching Multiple Waveguides

It has been shown that a three-dimensional waveguide can be created using a two-dimensional spline based on a direct line between the projection of the endpoint on to the x,y plane at the start face. Because the flexibility to put the input and output points at any place on the opposing faces is a requirement, it is inevitable that the direct line distance between start and end points will not be the same. Two waveguides can be designed with a lateral offset of 500 for Waveguide 1 and 250 for Waveguide 2. Creating two waveguide curves for each set of coordinates and translating the start point of Waveguide 2 to (0,0), gives the curves shown in Figure 9.

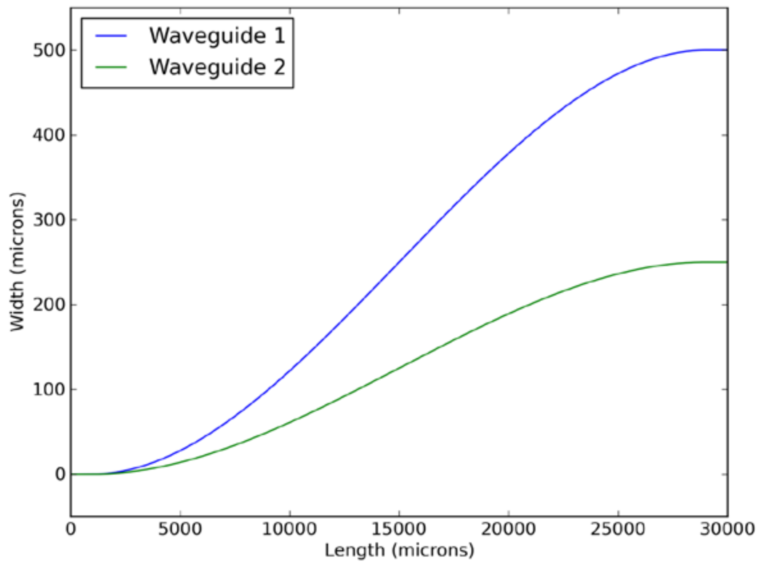


Figure 9 - Graph showing design of two spline based curves with different endpoints

This shows that the spline curve algorithm when used for various offset positions of the output has created a smooth spline curve for both. Intuitively, one can see that the length of waveguide 2 is less than waveguide 1. To match the path length of waveguide 2, since the end knots are fixed at the end points, the center knot was moved vertically in small increments until the curve length of waveguide 2 equals waveguide 1. Figure 10 graphically illustrates this principle.

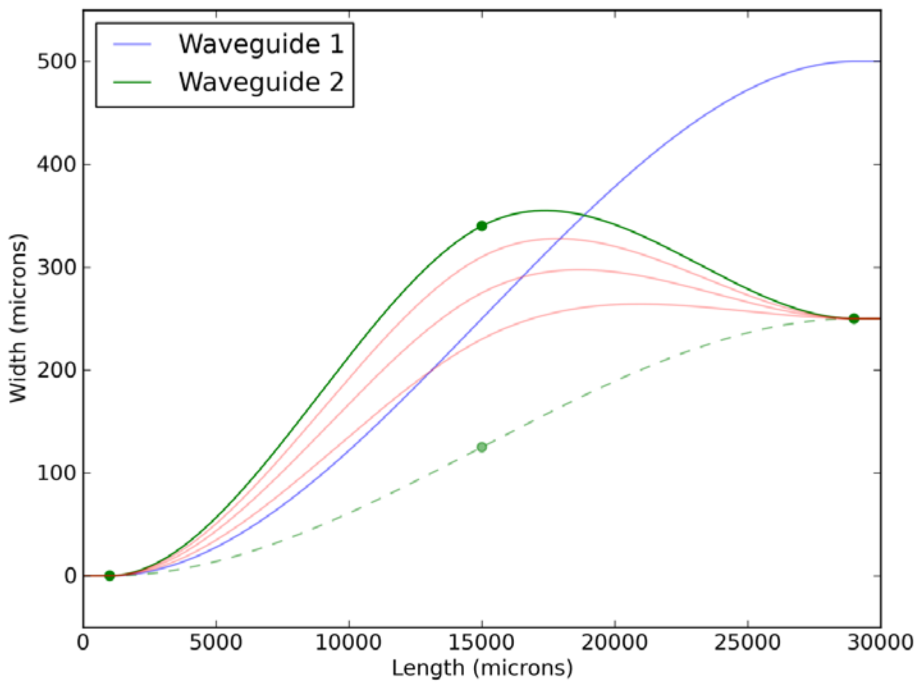


Figure 10 - Graph illustrating how waveguide curve path lengths are matched. Blue curve is original design of waveguide 1, dashed green curve is origin location for waveguide 2, red lines show increase of path until the solid green line, which is the path length match for waveguide 2

In this graph, the dashed green line represents the initial curve for waveguide 2. The center knot was moved vertically until it reaches the position at the solid green line, which represents the new curve for waveguide 2 with the length of its curve matched to the length of waveguide 1 to within 0.1 µm. The red lines show intermediate curves representing waveguide 2 as the knot is moved vertically. This process achieves path length matching of waveguide 2 to waveguide 1 which satisfies our design requirement. However, if there is a set of eight waveguides, a method is needed to match the path length of all eight. To do this, an array is created which finds the direct x,y distance of the start point of the waveguide and the projection of the end point for all eight waveguides. Path length can be added to a waveguide with a smaller offset than another one, but since the center knot of the waveguide with the larger offset is already placed at the midpoint, there is no way to reduce the distance of the larger waveguide. So, out of the eight waveguides, the one with the largest direct x,y offset was designated as the *primary waveguide*. The trajectory for this waveguide was solved first and the path length of that waveguide used as a reference for all others. The other seven waveguides, which were designated the *secondary waveguides*, each had their path length increased using the above method all had been path length matched. This gave a full set of eight path length matched waveguides.

3.2 Algorithm to Calculate Path Length of a Curve

Before proceeding further on meeting the remaining design requirements, there needs to be an explanation of the method for path length metrology in the previous section. As discussed previously, to create the waveguide prototypes the three-dimensional position of a locus of points needs to be provided every 25 µm in the longest direction (z axis). This value is used in the calculation and construction of the spline curves. However, an accurate measurement to calculate the path length of these spline curves, or any other discrete curves, was needed. The method used here was straightforward. When the curves were created in three-dimensional space, the length was calculated by summing the lengths between each adjacent three-dimensional point.

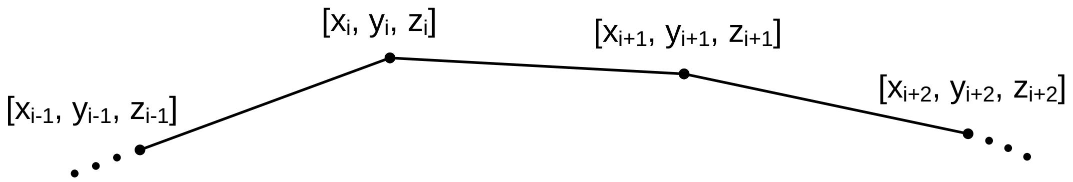

$[x_{i-1}, y_{i-1}, z_{i-1}]$
$[x_i, y_i, z_i]$
$[x_{i+1}, y_{i+1}, z_{i+1}]$
$[x_{i+2}, y_{i+2}, z_{i+2}]$

Figure 11 - Measurement of length between any two three-dimensional points

The formula to find the length between points is straightforward:

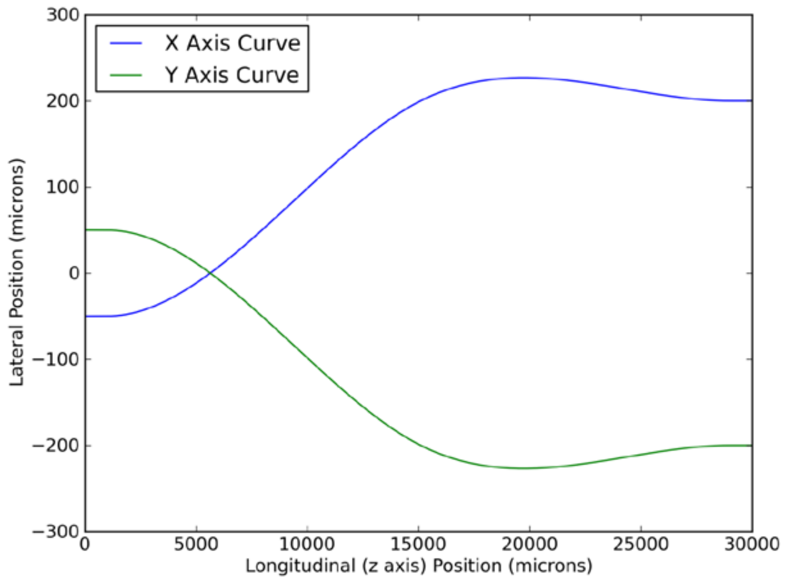
$$L_{i,\ i+1} = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2} \qquad (18)$$

The total length of the waveguide was the sum of all individual segment lengths. However, because the laser traces its own smooth path between the points given by the algorithm, on the still finer grid of 1 µm spacing, a comparison of how the length computed assuming 25 µm spacing compares to 1 µm spacing was made. With 1 µm spacing, there were 28000 data points that defined the spline curve, plus the 1mm section on either end. Using the previous example offset of 500 µm against the 30000 µm length, the results were a total curve length of 30005.339 µm for the 25 µm spacing and 30005.356 µm for 1 µm spacing. This gives a difference of only 0.02 µm, which was five times less than the 0.1 µm path-matching tolerance. If the offset was increased to 5000 µm, the result is 30528.6049 for 25 µm spacing and 30528.6052 µm for 1 µm spacing, giving a difference of less than 0.001 µm. This comparison shows that using a 25 µm spacing with the algorithms for creating the waveguides and calculating the path length of spline based waveguides is easily sufficient to meet the path length design tolerance of 0.1 µm.

3.3 Translating a Two-Dimensional Spline Curve into Three Dimensions

Thus far the examples examined here on translating the two-dimensional spline curve data into a three-dimensional locus of waveguide points have only had an offset in one axis, the x axis. However, because there was the requirement that input and output points could lie anywhere on the two respective surfaces, the angle formed by the input and output points could lie anywhere on a circle of 360°. To calculate the angle to rotate the spline, the arc tangent was used, as in Figure 7. This value of Θ gave a positive or negative orientation for x and y, meaning that the angle would lie in particular quadrant. Depending on the quadrant, the x and y values of the curve would be scaled appropriately. To avoid complication from the x and y values of the input point, the splines were created at a start point of (0,0) and then shifted to the start point after creation.

Although this method proved successful, an easier way was developed. Instead of creating one two-dimensional spline along the plane at angle Θ, two separate splines could be created individually for the x-z and y-z planes, giving the same result. This worked for the primary spline with no additional offset to the middle knot to increase path length. For the secondary splines, however, there still needed to be additional length added to the curve, with the middle knot of the spline shifted at the correct angle Θ. It was straightforward to split the offset into a shift in x and a shift in y by letting Δx = OffsetValue * cos(Θ) and Δy = OffsetValue * sin(Θ). As the offset is increased incrementally the x and y values were also scaled incrementally until the path length of the entire three-dimensional curve was equal to the primary curve length. Figure 12 shows an example of the creation of a secondary waveguide with input point at (-50,50) and output point of (200,-200), and a z of 30000, which has been path length matched to a waveguide with an offset of 500. The graph on the upper panel shows the separate x and y splines, while the lower panel shows a 3D rendering of the complete waveguide viewed from above.



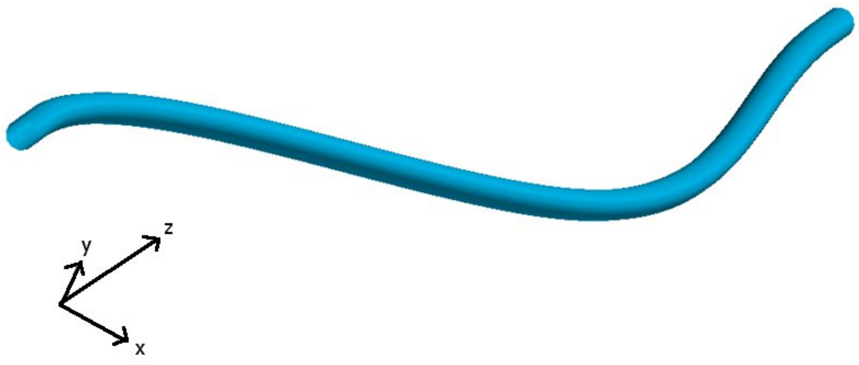


Figure 12 – (top) two two-dimensional spline curves in the x and y axes combine to create a three-dimensional waveguide (bottom)

3.4 Spatial Separation of Waveguides

With the tools to create waveguide paths developed, the start and end points for the eight waveguides discussed in Section 3.1 could be input into the computer application. When finished, the application displayed a three-dimensional picture of the eight waveguides using the library Mayavi2 in Python [37]. This gave the ability to rotate the image of the waveguides, or zoom into particular areas. Upon closer inspection of the 3D image of these eight waveguides, there were two points where the waveguides were actually intersecting, shown by the red circles in Figure 13. The width of the waveguides in this image was set at 10 µm, so by intersecting, the centers of these two waveguides were less than 10 µm apart. As discussed previously in Section 2.6.6 on waveguide separation, the waveguides needed to be separated by 30 µm or more. In the following sections, methods for measuring the separation of waveguides will be discussed together with methods for routing the waveguides to keep the required separation.

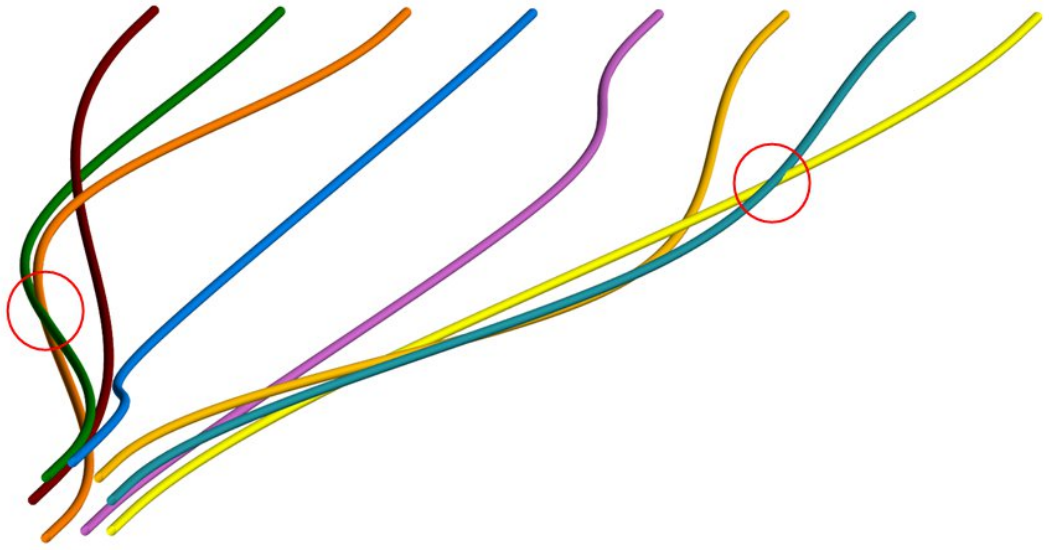


Figure 13 - Initial eight waveguide design with two red circles highlighting areas where two waveguides intersect each other

3.4.1 Determining Proximity between Neighboring Waveguides

The method to determine the separation between waveguides was similar to the method to determine the path length of each waveguide. This time, however, the algorithm went through each waveguide individually, and for each point on that waveguide, checked the distance to all other waveguides using the same length equation (18). For each waveguide, the algorithm goes incrementally through each point on the waveguide and compares the distance from that point to the closest points lengthwise on all of the other waveguides. Figure 14 gives an illustration of how the proximity is computed.

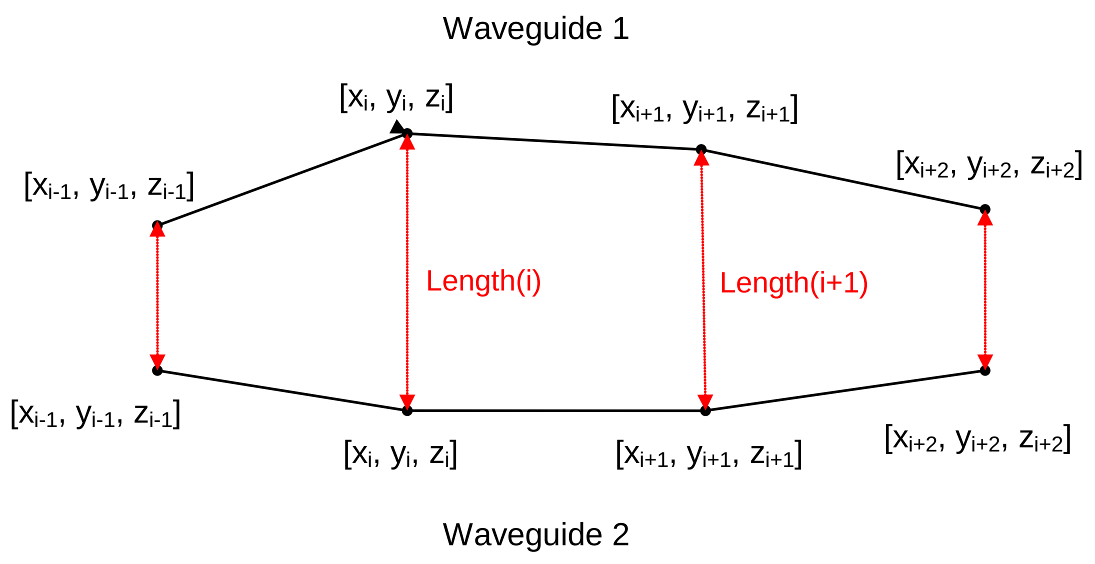

Waveguide 1
$[x_{i-1}, y_{i-1}, z_{i-1}]$
$[x_i, y_i, z_i]$
$[x_{i+1}, y_{i+1}, z_{i+1}]$
$[x_{i+2}, y_{i+2}, z_{i+2}]$
Length(i)
Length(i+1)
$[x_{i-1}, y_{i-1}, z_{i-1}]$
$[x_i, y_i, z_i]$
$[x_{i+1}, y_{i+1}, z_{i+1}]$
$[x_{i+2}, y_{i+2}, z_{i+2}]$
Waveguide 2

Figure 14 – Illustration of measurement proximity of distance of two waveguides (red) by comparison of their three-dimensional points in space

At a specific point i on waveguide 1, the distance is measured to point i on waveguide 2, where z_i is equal for both waveguides, since they are calculated in the same incremental value in the z direction. This process is repeated at all points from start to finish between waveguide 1 and waveguide 2 with that minimum value. The same proximity is then verified individually with waveguides 3 through 8. The algorithm keeps track of the minimum distance found along that waveguide to any of the other waveguides. The process is repeated for all eight waveguides, and after verifying the proximity distances for all waveguides, reports the closest distance found globally, the two waveguides that make up this distance, and at what point on those waveguides the distance was measured. This information becomes useful in the next section on arranging the waveguides to meet proximity requirements.

3.4.2 Adjusting the Curves to Meet Proximity Requirements

The next step was to design a way to adjust the position of our curve in space to spatially separate the waveguides. A radical change to the locus at this point is undesirable, since it has already met all of the previous requirements and constraints. Since these waveguides were created in essentially a polar coordinate structure, an additional angle of rotation about its longitudinal axis was added to each waveguide. Looking at Figure 13, imagine that each waveguide is twisted around its endpoints until each is arranged so that all spatial separation requirements are met. Although at first sight this seems straightforward, the problem is that the endpoints are laterally offset so the waveguides do not have an axis of rotation parallel to the z axis which simultaneously preserves their input and output points and endpoint orthogonality.

Because the primary waveguide is used to set the distance for the other secondary waveguides, it should remain unaltered while the secondary waveguides are re-oriented around it. Looking back at Figure 7, the start and end points form an angle θ with respect to the x axis. Figure 10 also shows that the largest spatial deflection for a secondary waveguide occurs around the middle knot. This may offer a potential way forward: instead of rotating the entire curve around the endpoints, the angle that the middle knot stretches with respect to the start point could be rotated instead. The middle knot could then be extended along this new angle until the resulting three-dimensional waveguide curve is path length matched to the same primary waveguide length. Figure 15 illustrates this concept.

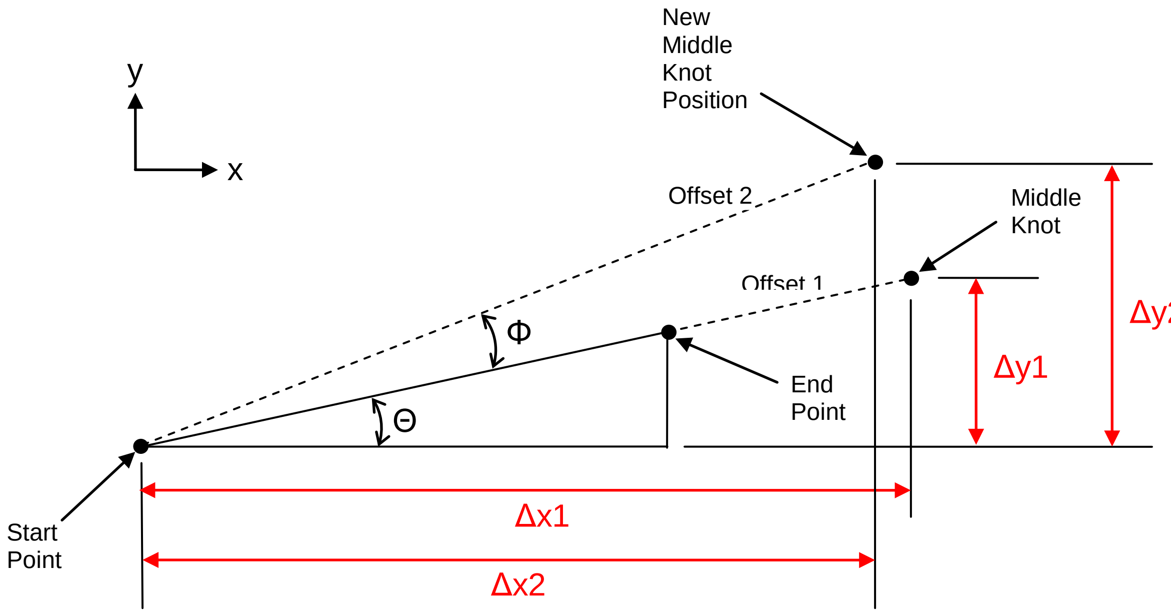


Figure 15 - Diagram illustrating middle knot rotation. The remapper block is viewed from end-on in this drawing, with the waveguide penetrating into the paper.

As discussed in Section 3.1.3, path length was added to the spline curve by offsetting the middle knot. If the start and end points lie at an angle Θ with respect to the x axis, then the middle knot was stretched with the relationship based on that angle, shown in Figure 15 above as Δx1 = OffsetValue * cos(Θ) and Δy1 = OffsetValue * sin(Θ). To avoid a spatial clash with a second guide, rotation of the angle at which the middle knot lies, as shown by the angle Φ above, with a final offset for the middle knot at an angle of Θ+Φ. This gave a new position for the middle knot shown in Figure 15 as Δx2 = OffsetValue * cos(Θ+Φ) and Δy2 = OffsetValue * sin(Θ+Φ). With the new middle knot position, the spline curve algorithm automatically calculated the new trajectories.

Figure 16 below shows the x, y and three-dimensional renderings of two waveguides. The blue curves show the creation of a secondary waveguide with input point at (0,0) and output point of (250, 50), and a length of 30000, being path length matched to a primary waveguide with an offset of 500. This waveguide was projected along an angle Θ of arctan(50/250) = 11.3° or 0.197 radians. The green curves show the design of a waveguide with the same endpoints and with the addition of middle knot rotation angle Φ of 10° or 0.175 radians.

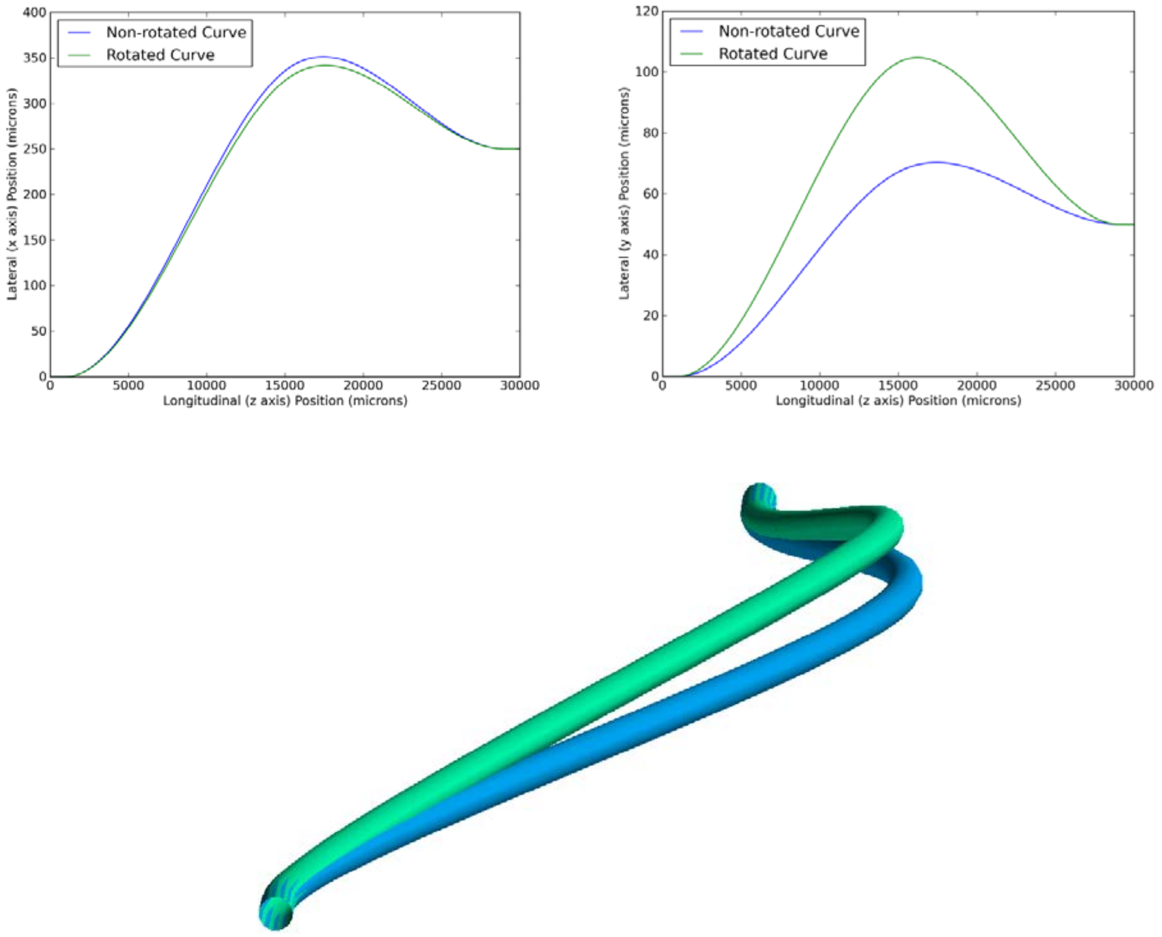


Figure 16 - X axis (top left) and Y axis (top right) isometric view and 3D view of non-rotated (blue) and rotated (green) waveguides (bottom)

The image illustrates the ability to position the waveguides in space by a rotation of the middle spline knot, while keeping the endpoints the same and the integrity of the waveguide curve intact. The mathematical machinery described above could now be used to successfully separate all eight waveguides in space.

3.4.3 Spatially Aligning Waveguides

Using the three-dimensional image of the waveguides, the user can examine images of the waveguides from every angle as well as zoom into particular sections of the image.

Using this tool, and varying the angle adjustment of the waveguides, as discussed in the previous section, it is possible with several cycles of adjustments, and using visual feedback as well as the proximity distance calculations, to create a set of eight waveguides. These guides were made using the previously stated endpoint coordinates, shown in Figure 6, now meeting also the proximity tolerance. Figure 17 below shows this design.

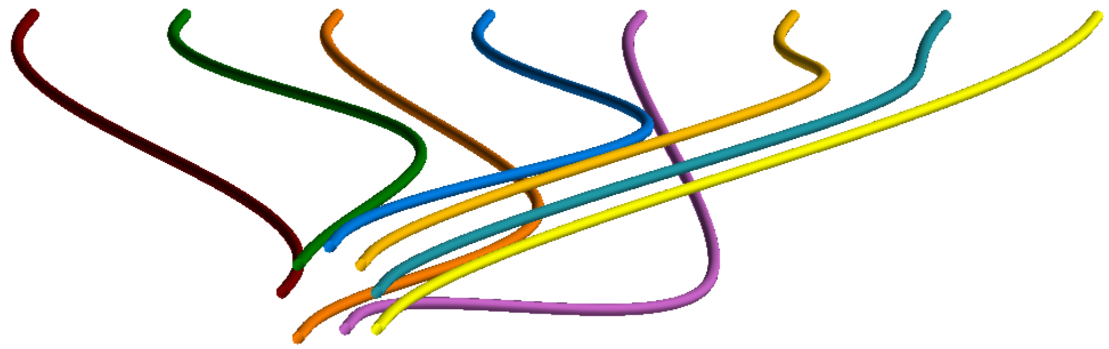


Figure 17 - Image of eight three-dimensional waveguides now meeting spatial separation requirement

Comparing this image to Figure 13, one can see that the two conflict areas that were found previously have been resolved. In fact, the orange waveguide has been completely re-oriented to the opposite direction. This was achieved simply by specifying a rotation angle of the middle knot of 180°.

This set of eight waveguides now meets all but two requirements discussed thus far. Using the image it was visually verified that no waveguides were entangled (see 2.6.8). The last requirement that needs to be confirmed is the writing depth constraint. Here a similar algorithm to the proximity one described in Section 3.2 was used. For each waveguide, the minimum and maximum values of depth in the y axis were found. All eight waveguides were compared and the absolute maximum and minimum height values measured against the origin at the center of the input array are found. For the design in Figure 17, the minimum height value was –213.5 µm and the maximum value was +105 µm, giving a height differential of 318.5 µm. This value was under the physical writing depth requirement of 350 µm. These values were created with the x and y origin at the center of the input pattern. To ensure that these values fit within the 100 µm to 450 µm depth requirements, the y values of the entire array are shifted to fit within those coordinates. These waveguides were now designed to meet all requirements detailed in Section 2.6.

3.5 Determining Radius of Curvature

The waveguides designed in Figure 17 were discrete curves and required an algorithm to calculate the radius of curvature from many individual line segments. As mentioned in Section 2.6.5, two adjacent line segments can be represented by two three-dimensional vectors. It is known that the angle between two vectors **a** and **b** can be calculated by using the dot product of two vectors, which is

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta) \tag{19}$$

Rearranging for θ gives

$$\theta = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right) \tag{20}$$

For any two three-dimensional vectors a and b, the dot product of the vectors can be found by the equation

$$\mathbf{a} \cdot \mathbf{b} = (a_x b_x + a_y b_y + a_z b_z) \tag{21}$$

So, looking at Figure 18 below, **a** can be represented by the vector from i-1 to i and **b** as the vector from i to i+1. This gives values for the vectors of

$$a_x = x_i - x_{i-1}\ ,\ a_y = y_i - y_{i-1}\ ,\ a_z = z_i - z_{i-1} \tag{22}$$

$$b_x = x_{i+1} - x_i\ ,\ b_y = y_{i+1} - y_i\ ,\ b_z = z_{i+1} - z_i$$

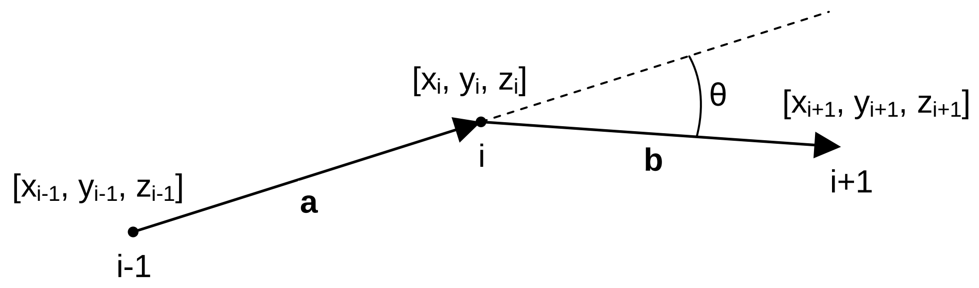

[x_i, y_i, z_i]
θ
[x_{i+1}, y_{i+1}, z_{i+1}]
i
b
i+1
[x_{i-1}, y_{i-1}, z_{i-1}]
a
i-1

Figure 18 – Diagram of two three-dimensional vectors in space illustrating curvature calculation

The value for the dot product is then given by

$$(x_i - x_{i-1})(x_{i+1} - x_i) + (y_i - y_{i-1})(y_{i+1} - y_i) + (z_i - z_{i-1})(z_{i+1} - z_i) \tag{23}$$

The magnitude of each vector **a** and **b** can be found by using the length value calculated in equation 18 and substituting all values into equation 23 gives

$$\theta = \arccos\left(\frac{(x_i - x_{i-1})(x_{i+1} - x_i) + (y_i - y_{i-1})(y_{i+1} - y_i) + (z_i - z_{i-1})(z_{i+1} - z_i)}{\left(\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 + (z_i - z_{i-1})^2}\right)\left(\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2}\right)}\right) \tag{24}$$

This gives the change in angle from one segment to the next, which is one measure of the curvature. To convert this to an estimate of the radius of curvature, the formula for vectors is used:

$$R_c = \frac{|\mathbf{a}| + |\mathbf{b}|}{2\theta} \tag{25}$$

Using the calculated magnitudes of the vectors and using the previously calculated value for Θ, a value for the radius of curvature at point i on Figure 18 can be determined. With these formulas, the radius of curvature could be calculated at each point of our waveguide locus. Calculating the radius of curvature for the primary waveguide (in yellow) in Figure 17 gives the graph below in Figure 19.

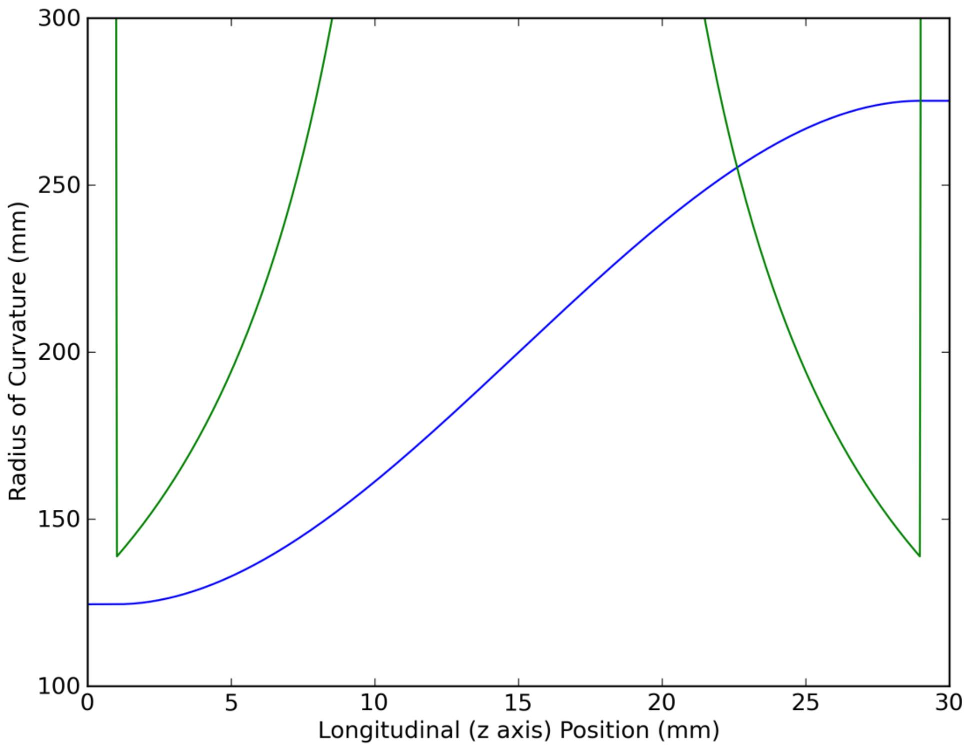


Figure 19 - Radius of curvature for a simple cubic spline in green, with the physical trajectory of the waveguide itself given in blue

The data in green gives the radius of curvature in millimeters at each point on the waveguide, while the physical locus of the waveguide itself is drawn in blue and overlaid for reference purposes (not to scale). Here, the radius of curvature drops to its lowest value of around 138 mm at either end where the straight sections transition directly to the curved portion of the waveguide. As the waveguide straightens out in the middle, the radius of curvature increases and for most of its length, this waveguide has a high radius of curvature. This method of displaying the radius of curvature will assist in the analysis of all further waveguides.

4. Sidestep Design

An initial prototype of the design was fabricated for the set of waveguides shown in Figure 17. The fabricated pupil remapper was then placed in an optical test setup shown below in Figure 20 to measure the waveguide properties.

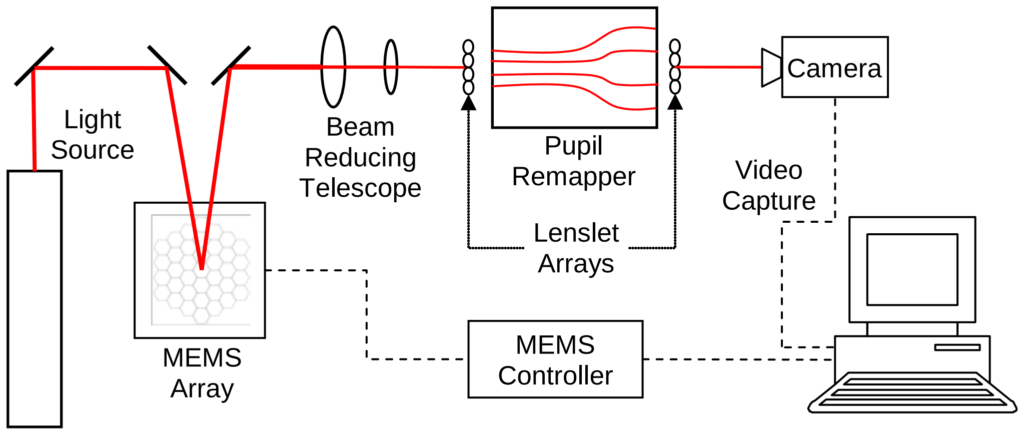


Figure 20 – Optical test setup for measuring interference fringes of pupil remapper chip

This setup consisted of a collimated light source (laser, broadband light source, lamp, etc.) to illuminate the waveguides and simulate the star light. The pupil from this light source is projected on to the MEMS array so that it completely fills it. The light is reflected to and from the MEMS array via a right angle mirror and then passes through a 20x beam reducing telescope that reduces the beam so the segments on the MEMS array match up with the individual elements of the two-dimensional input lenslet array. The lenslet array then matches up with the inputs of the pupil remapper waveguides. The light passes through each waveguide and then is output through another lenslet array on the back side of the remapper. The outputs of the remapper can then be imaged using a video camera. This camera feeds real-time image data to a computer, which is also connected to the MEMS controller. Using custom software, the computer scans the selected individual segments of the MEMS array through tip-tilt and piston adjustments to maximize the power throughput through each waveguide.

Tests of the initial prototype showed that interferometric fringes could be obtained from a pupil remapper, and that the path length matching among the waveguides was within the coherence length. However, after further testing concerning the stability of the closure phases recovered, a problem was found. When observing a spatially coherent source, closure phases should be zero and furthermore are inherently resistant to phases errors introduced in the pupil plane. Unfortunately, testing showed that the performance of our device in this respect was far below expectations, with RMS closure phase errors of several tens of degrees when piston terms were introduced in the pupil. Detailed description of these findings is beyond the scope of the present work, however, this immediately motivated a search for the root cause and a further generation of remapper designs to ameliorate the problem.

After a dedicated program of laboratory testing, the cause for the poor phase stability was tracked down to unguided stray light propagating through the bulk glass and interfering with the light in the waveguides. As illustrated in Figure 21 below, a significant fraction of light which does not couple to the guides spreads outward in a cone depicted as a red triangle. This stray-light interference is unfortunately placed so as to overlap with the desired signal at the output of the remapper waveguides.

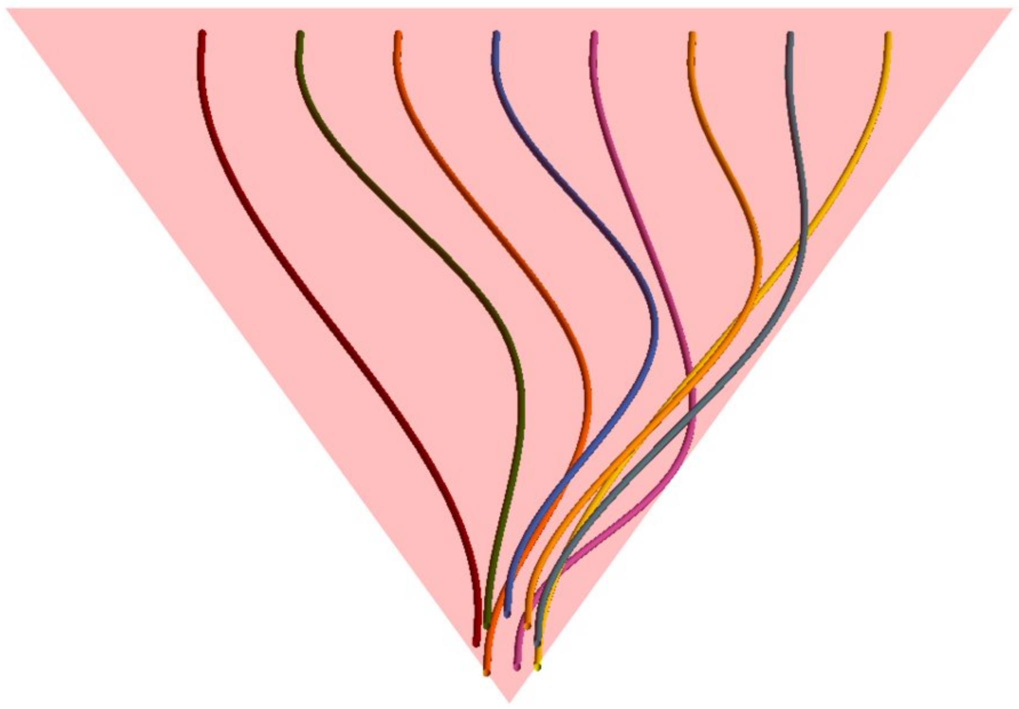


Figure 21 - Image of three-dimensional waveguides with overlaid cone of optical noise interference

Further details of this laboratory testing campaign is described in *Norris et al* [38], however for the purposes here, it is sufficient to state that the performance achieved was not competitive for astronomical application and further re-designs to mitigate the phase stability problems were essential.

4.1 Adding a Lateral Sidestep to Avoid Interference

Figure 21 shows that the bulk of the unguided stray light spreads out as a cone at a specific angle equal to the f-ratio of the final injection lenslet array. Since the remapper architecture boasts the capability to route the waveguide ports to any desired location, the output waveguide array was moved laterally until it lay outside the cone of worst light interference. Figure 22 illustrates this "sidestep" remapper concept.

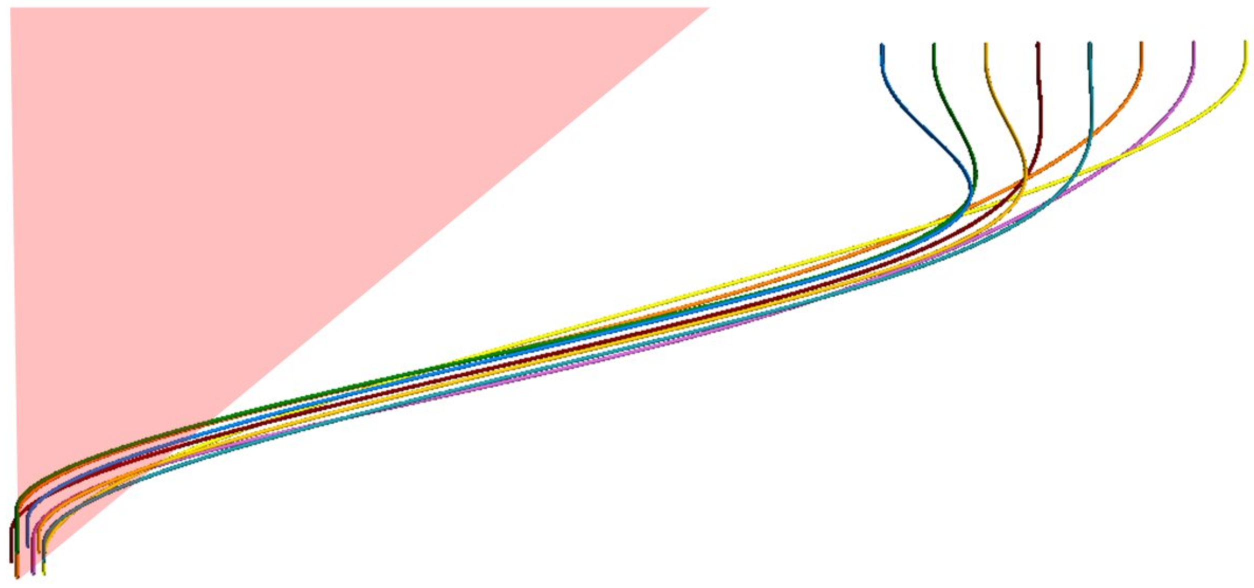


Figure 22 – Utilization of sidestep waveguide design to avoid light interference

The output points for this new design have been shifted to the right in the x axis by 5 mm. At this new position, the outputs of the waveguide lie outside most of the interference cone from the stray input light and should give better results in recovering robust closure phases. The image of the waveguides in Figure 22 came from the completed design. To get to this point, additional design challenges arose with the 5 mm sidestep offset, which will be discussed in the next few sections.

4.2 Proximity Challenges

Adopting the most straightforward approach to the design, new input and output points incorporating the sidestep were fed into the algorithm and the process of adjusting the positions of the waveguides was performed as described earlier using the rotation method. By manually adjusting parameters, it was found that very small rotation adjustments in the secondary waveguides now produce large changes in the y axis. This is intuitive as the value of the offset is so large in the axis that any change in θ in Figure 15 will produce a bigger change in the y axis than the previous design. This causes problems when trying to keep the height of all waveguides within the 350 µm constraint.

4.2.1 An Additional Lead-In Straight Section

In the early stages of manually adjusting the rotation angle of the waveguides, the main difficulty noticed was that the waveguides tended to come into proximity conflict where they all bunch together turning the corner to implement the sidestep. Figure 23 below illustrates one such clash.

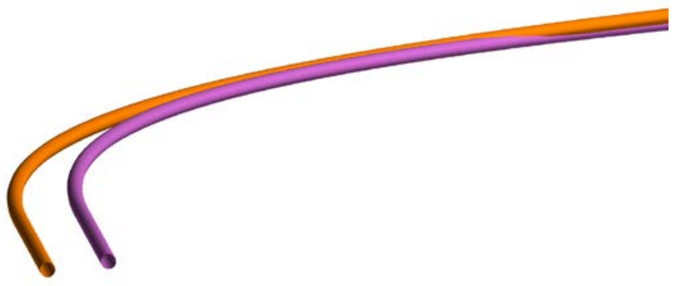


Figure 23 – Diagram showing two waveguides at input point clashing at the onset of the sidestep

It is seen above that the orange waveguide runs into the purple waveguide as both waveguides curve laterally towards their respective outputs. The earlier solution to such clashes, rotation of the waveguides, is now problematic as discussed in the previous paragraph. However, a solution was found by extending the straight section of the outside waveguide. There is already a straight section at the beginning of each waveguide, so if an extra straight section was added to the orange waveguide, the two waveguides stayed separated and kept their separation all the way to the output. The successful outcome of this strategy is shown below in Figure 24.

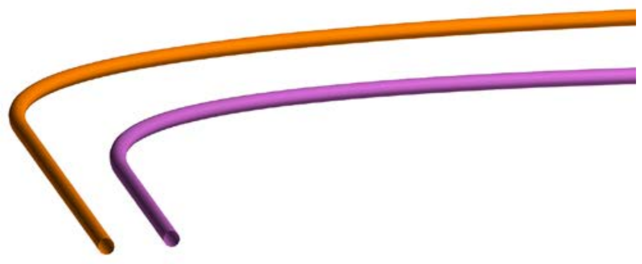


Figure 24 – Same waveguides as Figure 20 now spatially separated with an additional length of straight waveguide section to the orange waveguide

This gives yet another design feature to keep the waveguides spatially separated. However this does come at a cost. With the additional straight section at the beginning, there was less length to implement the spline curve and so the middle knot needed more lateral offset, increasing the overall maximum curvature of the waveguide.

4.3 Effects of Sidestep Design on Radius of Curvature

Examining the sidestep waveguides in Figure 22, the radius of curvature as a function of length was plotted for the primary waveguide (in yellow) and is shown in Figure 25. Comparing the values for radius of curvature of this sidestep waveguide versus the straight through design plot in Figure 19 shows the same shape for both graphs, but the increase in lateral offset has dropped the minimum value of radius of curvature from around 138 mm down to around 23 mm. Thus, the addition of the sidestep has contributed to a considerable drop in the radius of curvature.

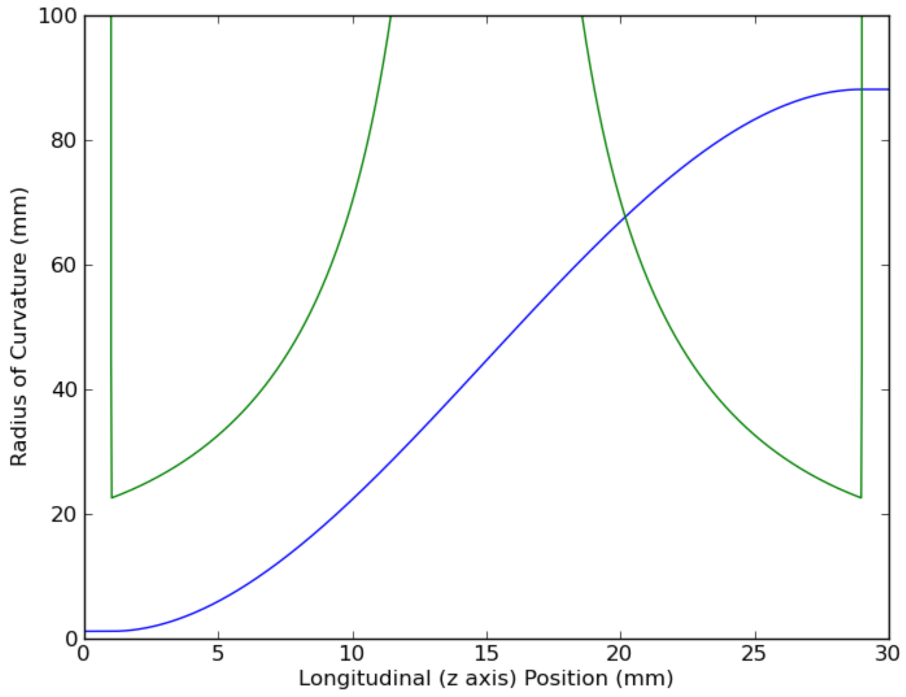


Figure 25 – Radius of curvature for a simple cubic spline in green, with the physical trajectory of the waveguide itself given in blue

As a comparison with the relatively benign gentle curves of the primary waveguide shown in Figure 25, the waveguide with the lowest radius of curvature of the same set will be examined. Figure 26 below shows one of the secondary waveguide locus (blue) along with its radius of curvature (green). In that image, the radius of curvature drops to an even lower value of around 10 mm near the beginning of the waveguide as it transitions from the end straight section to the portion of the curve with the largest slope. As the light travels along this waveguide, it encounters several turning points of large radius of curvature, and several regions of tight curvature dictated by the spline solution. The effects of these changes in radius of curvature will be investigated further in Section 6.

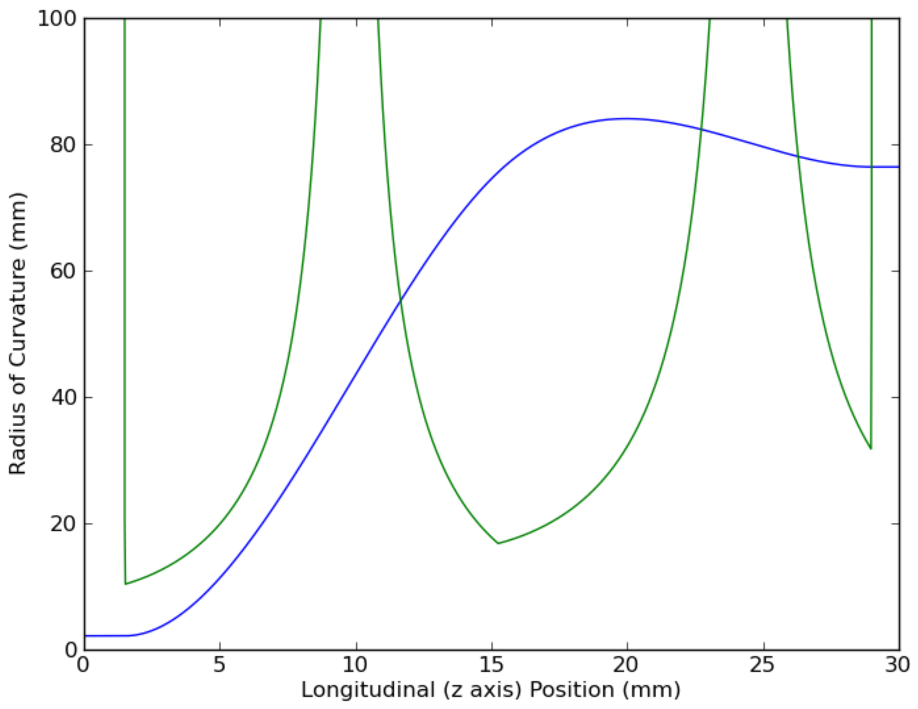


Figure 26 – Radius of curvature (green) for a more complex spline curve design (blue)

5. A Waveguide Based on a Circular Arc

5.1 Design of the Two-Dimensional Arc-based Curve

The radius of curvature measure is based on the radius of the circle that best matches the curvature at any given point. As an alternative waveguide design method to the cubic spline, the use of a circular arc to construct the waveguides was explored. This particular architecture holds great promise for minimizing the bends for it presents the mathematically lowest curvature solutions when threading a trajectory meeting our requirements. To construct a simple two-dimensional waveguide in the cylindrical coordinates z and r with a given lateral offset, similar to the spline based waveguide shown in Figure 8, the design procedure started with an arc from a circle with a tangent parallel to the z axis to connect to the input straight section at the beginning of the waveguide. Likewise, a circular arc, tangential to the z axis, was used at the end. To connect these two arcs together, the radius of both was selected such that the tangents of the two arcs intersect perfectly at the midpoint of the line that connects the start and end points. Figure 27 below illustrates the concept.

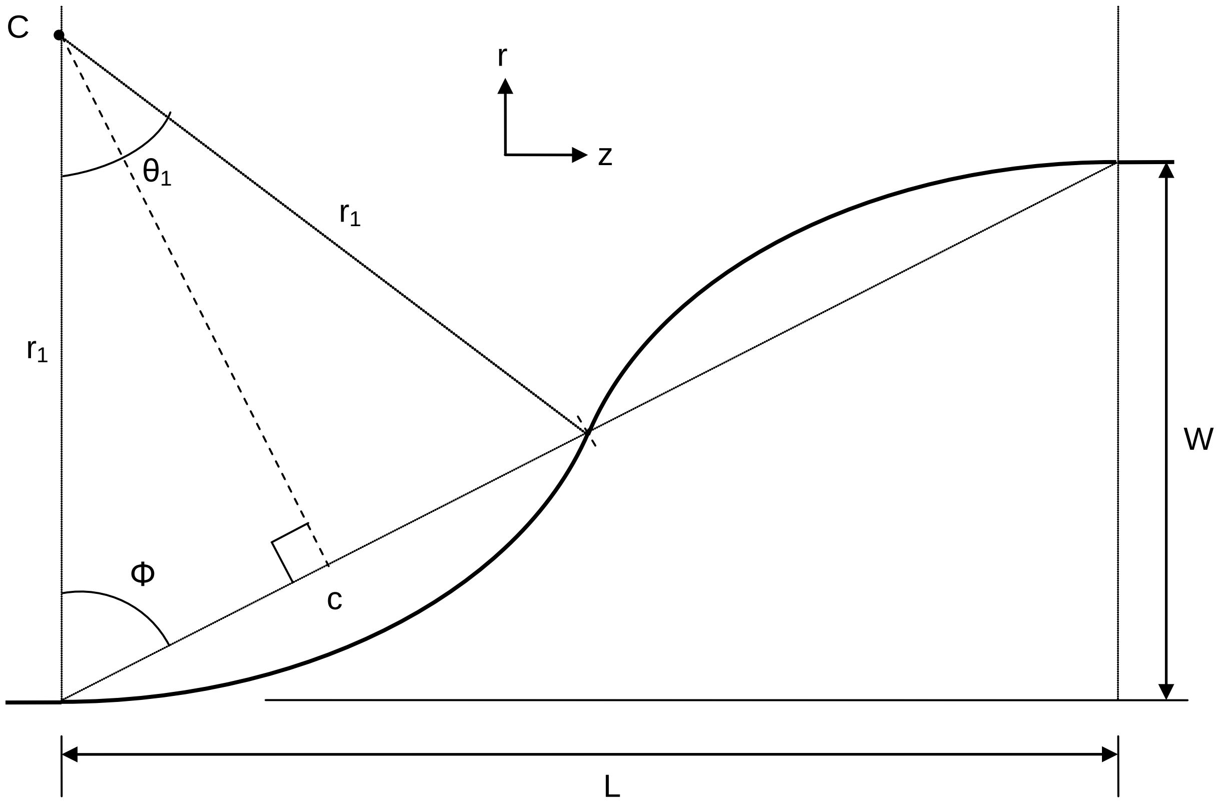


Figure 27 – Creation of a waveguide from two circular arc sections with variables for calculation and design of waveguides are indicated

To construct the left arc, we require the values for r_1 and θ_1 as a function of L and W. With some simple geometrical construction, it can be shown that if we define angle Φ to be:

$$\Phi = \arctan\left(\frac{L}{W}\right) \tag{26}$$

Then θ, can be readily found as:

$$\theta_1 = 2(90° - \Phi) \tag{27}$$

To derive the value of the radius r, the value of the chord c and the cosine function can be used to give:

$$r_1 = \frac{\frac{c}{2}}{\cos(\Phi)} \tag{28}$$

The arc for the left half of the waveguide could now be created with the values of the radius and the arc angle θ_1. To calculate the values of the right arc in Figure 27, the process is similar to the left arc calculation, except the center is reflected to the bottom right corner. The y value is then added to the y value of that center to get the actual y coordinate, with the two curves meeting in the middle.

With the method for calculating a curve made from arcs given, a primary, arc-based waveguide can be created, with the same input and output coordinates as the previous sidestep spline design shown in Figure 8.

For an arc based waveguide, the two arcs for the primary waveguide were created with the midpoint at the middle, and for any pair of input and output points, the entire path is uniquely defined. However this causes a problem: the flexibility to adjust the path length of the guides is needed to meet the matching criterion. At first glance it might appear that a ready solution might be to move the junction of the two arcs away from the middle toward one of the endpoints, thus creating one arc of larger radius and one with a smaller radius. For the waveguide in Figure 27, an equation for the total path length of the curve is:

$$\text{Length}_{\text{total}} = \text{Length}_{\text{ends}} + 2\pi * r_{\text{left}} * \frac{\theta_{\text{left}}}{360°} + 2\pi * r_{\text{right}} * \frac{\theta_{\text{right}}}{360°} \tag{29}$$

with r being the radius of each arc, and θ the angle that each arc swept out. If the midpoint is shifted along the chord line, the radius of the left arc increases, and the radius of the right arc decreases. However, the angle θ must remain the same for both arcs so that they intersect together at the same angle. Subtracting the length of the ends and dividing both sides of the equation by 2π and $\frac{\theta}{360°}$ gives the result that the sum of r_{left} and r_{right} is a constant value. Thus, shifting the point of intersection of the two arcs does nothing to alter the path length.

5.1.1 Addition of a Bridging Straight Section

Another possibility explored for adding path length to a curve made from two arcs was joining them with a bridging straight section inserted at the tangential meeting point. Thus, the arcs could be made at any angle and the path length increased. Equation 29 was modified to show what would happen with the addition of such a straight section.

$$\mathrm{Length_{total}} = \mathrm{Length_{ends}} + \mathrm{Length_{straight\ section}} + 2\pi * r_{left} * \frac{\theta_{left}}{360°} + 2\pi * r_{right} * \frac{\theta_{right}}{360°} \qquad (30)$$

Equation 30 shows that for a fixed total length, the additional straight section leaves less length for both the left and right arc sections. This means that either the radius or angle of the arc will need to be decreased. Looking again at Figure 27, if the angle that the left arc sweeps out only is decreased, the entire geometry fails to deliver a smooth curve which meets at the other end. So, the addition of a straight section requires a reduction in the radius of both arcs. This, in turn, implies a reduction in the radius of curvature of at least one arc. Such a strategy runs counter to the original motivation: keeping minimal bend loss with as high of overall curvature as possible

5.1.2 Addition of a Third Circle

From the preceding sections, the conclusion was that design of a waveguide curve based solely on two arcs could not increase path length without significantly decreasing the radius of curvature and increasing the bend loss. However, the addition of a third circular arc to connect the other two arcs was found to present a possible solution.

A third circle can be added as depicted in Figure 28 below, such that there are now two smooth tangential transitions between arcs and the input and output directions are preserved. However, the third circle is always “wedged” in between the two circles and gives a parameter to adjust the geometry. The way to increase the path length for the arc was to decrease the radii of all three circles simultaneously by the same amount. This way the radius of curvature is the same for all three circles, and is a global maximum (desirable for minimizing bend losses). While decreasing the radius of each circle, the original input and output arcs keep their edges tangential to the straight section endpoints. The third circle is constructed at a position that touches both of the other circles tangentially.

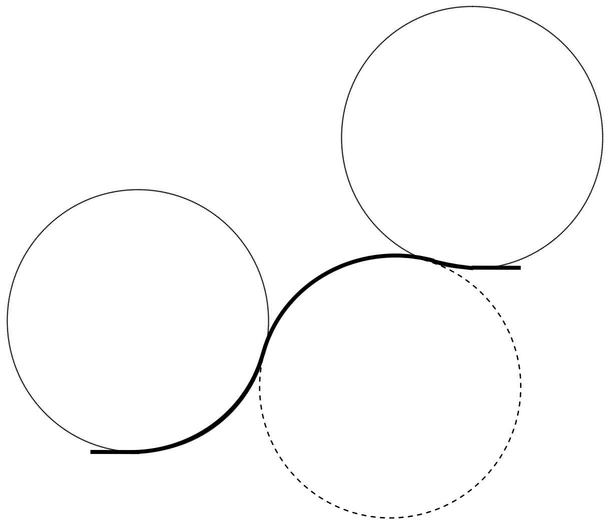


Figure 28 – Design of waveguide with three arcs from tangential circles with reduced radius to give additional path length

With the reduction of the radii of all three circles, the waveguide curve now shows an increased path length over the two circle based waveguide. Because the radius of each circle was reduced, the radius of curvature for the whole curve (constant over the whole waveguide) was reduced as well. For the process of ensuring path length matching, the circles were assembled as in Figure 28, with the radius of each circle set to the same value as the calculated radius of the primary waveguide. Because the secondary waveguide had a smaller lateral offset, its length was less than the length of the primary waveguide. The radius of each circle was then decreased by a small amount until the path length of the entire curve was within tolerance of the path length of the primary waveguide curve. The process was repeated until each waveguide's path length was matched.

5.1.3 Constructing a Three Arc Curve

Construction of a three arc waveguide required new sets of equations to construct the sections. Modeling the geometry of the three arc design gave the following equation

$$\mathrm{Length_{total}} = \mathrm{Length_{ends}} + 2\pi \mathrm{r_{left}}\frac{\theta_{\mathrm{left}}}{360°} + 2\pi \mathrm{r_{center}}\frac{\theta_{\mathrm{center}}}{360°} + 2\pi \mathrm{r_{right}}\frac{\theta_{\mathrm{right}}}{360°} \tag{31}$$

Ignoring the straight end sections for now and focusing on the curved portion only, the curve is made from three circular arcs, traced from 3 identical circles in different positions. To calculate the correct angle that each arc sweeps out, with the radius chosen to meet the path matching criterion as discussed above, the center positions of all three circles were needed. In addition, the point where arc 1 transitions to arc 2, as well as the point where arc 2 transitions to arc 3, was also needed. Once all of these points were calculated, the curve could be constructed.

Looking at Figure 29 below, the center of circle 1 had an x value the same as the waveguide start point on the left, and a y value equal to the start point y value plus the radius. The same held for the center of circle 3 and the waveguide end point on the right. With these two center points found, a line could be constructed between these two points, the length of which was found by:

$$L_{C1,C3} = \sqrt{(x_{C3} - x_{C1})^2 + (y_{C3} - y_{C1})^2} \tag{32}$$

This line was used to triangulate the position of the center of circle 2. If two circles are tangential to each other, a line can be drawn from the center of one circle to the other that passes through that tangential point. Because circle 2 is tangential to both circle 1 and circle 3, there will be two lines drawn from the center of circle 2 to the other two circles, creating a triangle with segment L connecting the centers of circle 1 and 3. Because the radii of all three circles are equal, the lengths of these two lines are also equal, creating an isosceles triangle, with the angles opposite to the two identical sides being equal. These angles were denoted as Φ given by the equation, using the bisect of θ_2:

$$\Phi = \arccos\left(\frac{\frac{L_{1,3}}{2}}{2r}\right) \tag{33}$$

With the value of Φ, the three different angles needed to construct the three separate arcs could be calculated and are also shown below in Figure 29.

To first calculate angle θ_1, an imaginary horizontal line was drawn from C1 forming the base of a right triangle with a hypotenuse of $L_{1,3}$. With a little further simple geometry, the following angle can be found

$$\theta_1 = 90° + \arcsin\left(\frac{y_{C3} - y_{C1}}{L_{1,3}}\right) - \Phi \tag{34}$$

With the value of θ_1, the x and y coordinates of C2 were

$$x_{C2} = 2r * \sin(\theta_1) + x_{C1} \quad , \quad y_{C2} = -2r * \cos(\theta_1) + y_{C1} \tag{35}$$

Using the same imaginary horizontal line as the calculation for C1 and considering the angles at vertex C3 gives the equation

$$\theta_3 = \arccos\left(\frac{y_{C3} - y_{C1}}{L_{1,3}}\right) - \Phi \tag{36}$$

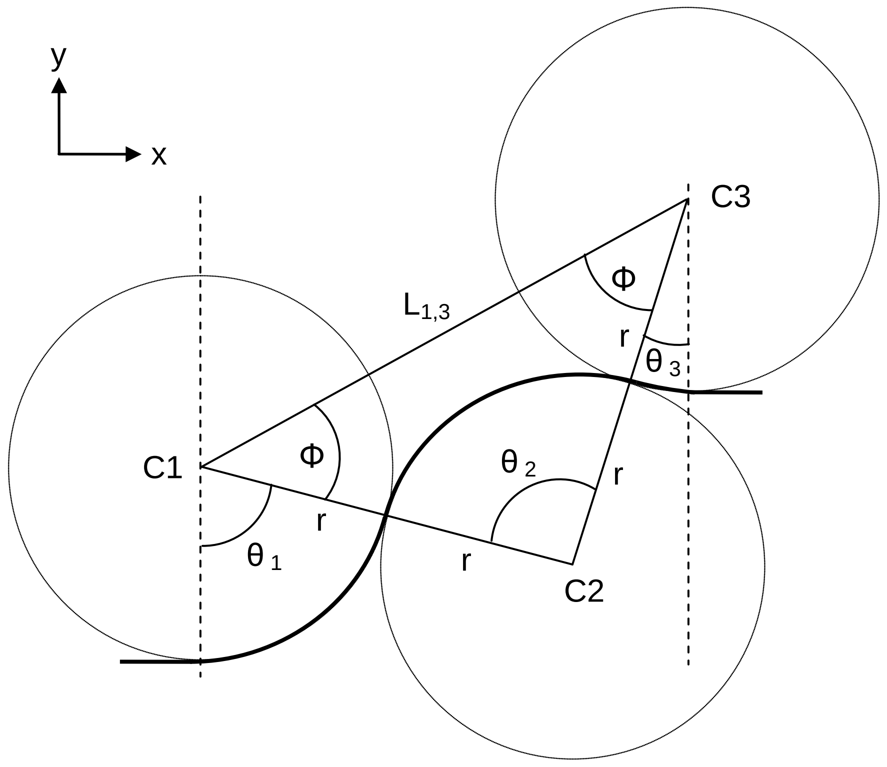


Figure 29 – Diagram for calculating portions of a three arc waveguide along with variables needed for calculation

The value of θ_2 is now trivially given by summing the angles in the triangle. The procedure for creating the curve was similar to the design based on two circular arcs. The value of x was based on the increment value given previously, and the value y was calculated. Depending on where the value of x lay, one of three equations was used. If x lay within the arc traced by θ_1, y will be calculated by the radius and the value of the local variable, angle α. The x value of the transition point from arc 1 to arc 2 was determined by

$$x_{T12} = r * \sin(\theta_1) \tag{37}$$

If the value of x is between the start point and x_{T12} the value of α was

$$\alpha = \arcsin\left(\frac{x}{r}\right) \tag{38}$$

And using that angle value, the value of y was calculated as

$$y = y_{C1} - r * \cos(\alpha) \tag{39}$$

To determine the next transition point between arc 2 and arc 3, the equation to use was

$$x_{T23} = x_{C3} - r * \sin(\theta_3) \tag{40}$$

If the value of x lay between x_{T12} and x_{T23}, the value of α was

$$\alpha = \arcsin\left(\frac{|x_{C2} - x|}{r}\right) \tag{41}$$

Here the absolute value was needed as the value of the inverse sine could change depending on if the value of x lay to the left or the right of the x value of C2. To get the value for y, the equation below was used

$$y = y_{C2} + r * \cos(\alpha) \tag{42}$$

Finally, if the value of x lay in arc 3, the value of α was

$$\alpha = \arcsin\left(\frac{x_{C3} - x}{r}\right) \tag{43}$$

With the value of y found by:

$$y = y_{C3} - r * \cos(\alpha) \tag{44}$$

This now fully specifies the x and y coordinates for all parts of the curve, and the equations implemented into the algorithm to create secondary waveguide curves based on circular arcs.

5.2 Construction of Three-Dimensional Arc Based Waveguides and Proximity Adjustments

The construction of a three-dimensional arc based waveguide used a similar procedure as the spline based waveguide detailed in Section 3.3. The angle θ of the start point to endpoint projection was used to interpolate a single two-dimensional curve into three dimensions. This was also how the primary arc based waveguide was created. The secondary waveguides were initially created this way, but again the proximity issues forced a change in the way that the waveguides were developed.

Adjusting the waveguide curve to meet the waveguide proximity requirement was a more difficult task for the secondary arc based curves than for the spline based curve. The spline based curve allowed for a rotational movement of the middle knot in space to move the waveguide, and the spline creation algorithm created the curve based on that point. The arc curve algorithm was created in either the x or y dimension (three-dimensional Cartesian coordinates) and was based on a fixed single radius for the waveguide. As discussed previously, the waveguides were highly constrained in the y dimension due to the physical chip height requirement from fabrication, meaning that any curvature in the y dimension must be very small. As before, two waveguides were created in both the x and y dimension against the z axis as length. The x axis curve contained most of the offset and it was to this dimension that the primary and secondary guides were formed as described above. The y axis could then be constructed by using the same two arc method as the primary waveguide. Since, the x axis curve accounts for the majority of the curvature, this design would still be optimized for the highest radius of curvature.

However because the two-arc method gives a fully prescribed curve for a given input and output, this gave no intrinsic ability to adjust the position of the waveguide in space to meet proximity requirements. If instead a three arc curve is employed with a very large radius then the ability to tune the shape of the guide to avoid clashes is recovered. Figure 30 below shows how the y axis curve changed for radius values of 50 mm, 100 mm, and 200 mm.

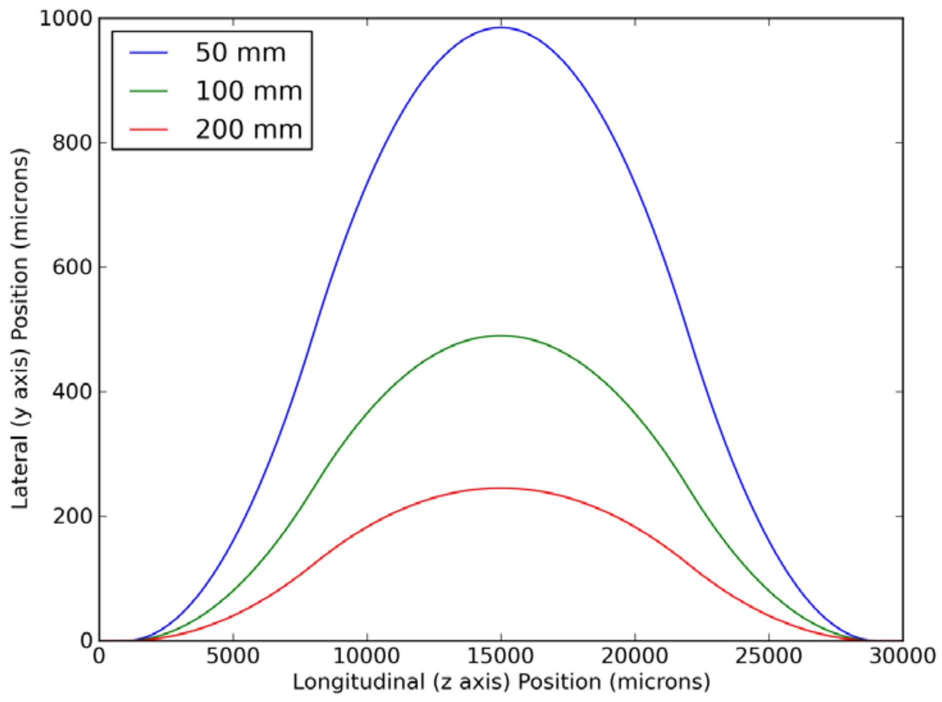


Figure 30 – Change of y-axis arc curve at different radius values showing a decreasing radius of curvature contributing to increased offset in the y axis

Figure 30 shows that using a specific radius for the y curve, independent of the x curve, provided a way to adjust the position of the three-dimensional waveguide in space. If more elevation of the waveguide was needed in the y dimension to avoid a clash, a lower radius of curvature was passed into the y curve algorithm. In order to simultaneously match the path lengths for waveguides now curving in both x and y, the concept of a *scaling factor*, which adjusts the simultaneous value of the y radius based on the current value of the x radius, was introduced. As the x radius decreased from the value set by the primary waveguide, the y radius decreased in proportion set by the scaling factor. This permitted a single-parameter adjustment which enabled path length matching and vertical-height clashes to be resolved simultaneously.

For a waveguide with an x offset of 5 mm, the scaling factors that give the y curves above in Figure 30 are 1.69, 3.76, and 7.68. The x curves for those same waveguides are shown below in Figure 31.

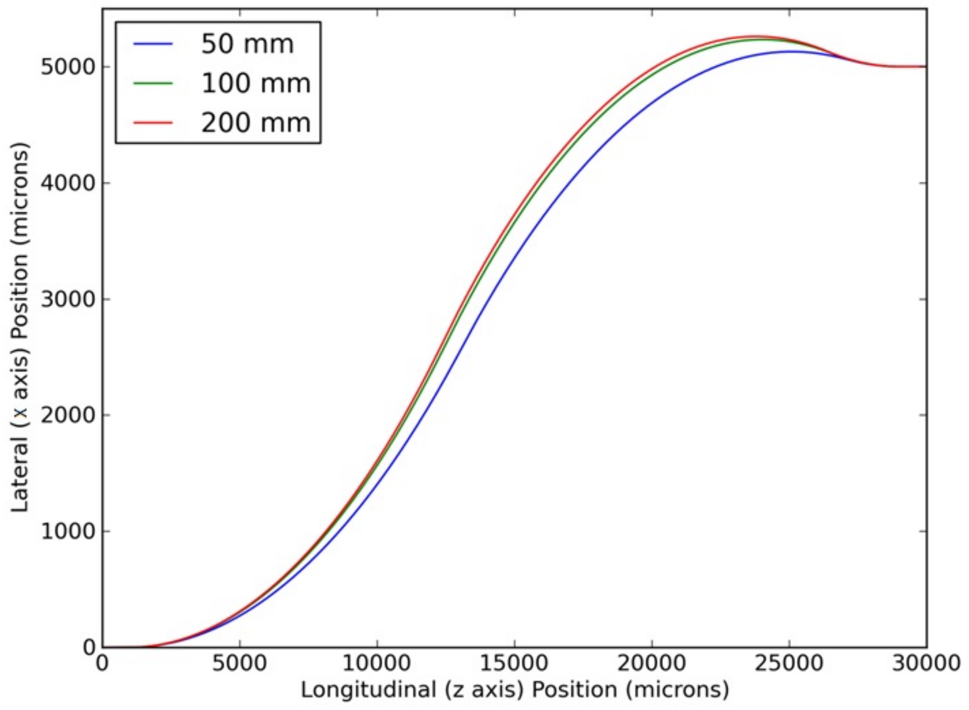


Figure 31 - Change of x-axis arc curve at the same different radius values

This graph shows a decrease in the scaling of the curve in the x axis. This was because as the curve in the y axis increases, it contributed more towards the overall three-dimensional path length, relieving some of the demands in the x dimension. When combined, the three-dimensional waveguide had a lower global radius of curvature, as shown in the table below:

Scaling Factor	Radius of Curvature(mm)
1.69	25.5
3.76	25.7
7.68	25.8
None	25.8

Table 1 – Radius of Curvature Values after Waveguide Scaling at Same Total Curve Length

As the scaling factor decreases, a larger offset in the y axis occurs and when the waveguide is length matched, the overall radius of curvature decreases. However, the lower scaling factor and higher curve in the y dimension did not lower the radius of curvature by a significant amount. Also, the first two scaling factors gave curves that had a difference of more than 350 µm, so in practice, neither of these could be used due to the physical chip height requirement. However, the straight forward ability to yield large vertical height offsets meant that this proximity scaling mechanism could be useful in creation of the waveguides.

Ensuring that the three-dimensional waveguides algorithm worked properly when the difference between the end and start points, in x and/or y, was negative required a slightly different algorithm than the spline based curves. The arc based curve creation algorithm discussed in Section 5.1 required a fixed calculation with the endpoint greater than the start point. So, any negative values in x and/or y required the absolute value of the offset between start and endpoints to be passed in. To then orient the three-dimensional waveguide correctly in space, an algorithm was created based on the difference of the endpoints in both the x and y dimensions and also whether the scaling factor for the y radius is positive or negative. An in-depth discussion of this algorithm is given in the Appendix. In addition to positioning of the x and y coordinates in space, the algorithm also has the ability to rotate a waveguide design, like the one in Figure 29, 180 degrees, giving another option for waveguide placement.

Finally, use of the scaling factor to place the waveguides in space was done using visual alignment from three-dimensional images. In addition, the lead in straight sections discussed in Section 4.2.1 also worked the same way for the arc based waveguides as the splines. These lead in sections also reduced the radius of curvature of the arc based waveguides, so were used carefully.

6. Characterization of Optical Power Loss in Waveguides

Chapter 23 of Snyder and Love's book, *Optical Waveguide Theory* [31], is devoted to bends in optical fibers and waveguides. In that chapter, the relationship between the radius of curvature of a waveguide and optical power loss due to that radius is established. This formalism has been applied here to determine the power loss in both the spline and arc based waveguides based on their radius of curvature values.

6.1 Bend Loss due to Radius of Curvature

In this section, the concept of *macrobending* loss is investigated, in which the radius of curvature of the bend in the waveguide is much higher than the cross-sectional radius of the waveguide itself. *Microbending* loss, where the bends are on the order of the waveguide radius, won't be addressed here, as any microbends in these waveguides will be errors related to the fabrication process. Any microbending in the design here would be related to the continuity of the waveguide design, and a continuous waveguide has already been specified as a basic design requirement.

The physical mechanism of macrobending loss is due to radiation of the modal field traveling in a bent waveguide. As a light wave travels in a straight waveguide, the entire field travels at a specific phase velocity, depending on the effective index of refraction for that mode. When that modal field travels in a bent waveguide with a constant radius of curvature, the phase velocity now is an angular velocity rotating about the center of curvature. Because it is an angular velocity, the linear velocity needs to be higher at the outside edge of the curve than at the center.

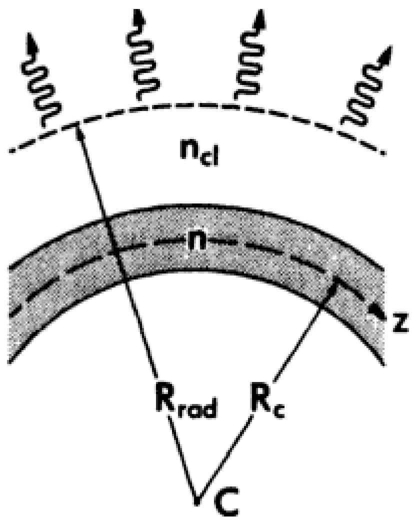


Figure 32 – Illustration of bend loss radiation in a waveguide with n as the index of the core section (shaded), n_{cl} as the cladding section, R_c as the radius of curvature of the waveguide and the outward lines illustrating the direction of radiating bend loss (courtesy: *Snyder and Love*)

As the phase velocity is governed by the local refractive index, which is symmetrical across the guide between the inner and outer edges of the curve, the outside of the field will start to lag behind the rest of the field. This distorts the wavefront, causing a component of the wavefront vector to point outward radially. Hence, some of the energy radiates outward, dissipating into the cladding. Figure 32 illustrates the concept, with n, the index of refraction of the core, n_{cl} the index of refraction of the cladding, and R_c the radius of curvature of the waveguide.

6.1.1 Bend Loss Calculations

Calculation of the curvature of the waveguide based on discrete sections was performed in Section 3.5. To find the bend loss due to the radius of curvature of an individual waveguide section, the discrete bend loss equation from Snyder & Love [31] is adopted:

$$\mathrm{P(z) = P(0)e^{-\gamma z}} \tag{45}$$

This says that if light travels in a waveguide over a distance of z, and this waveguide is curved, the power at the output will equal the power at the input multiplied by a negative exponential factor dependent on the distance z and a coefficient gamma. The difference between input and output power gives the bend loss. Snyder and Love also give two bend loss equations, depending on the type of waveguide. For a step profile, where the transition across the waveguide is nearly instantaneous from core to cladding and is flat across the core (like most optical fibers), the equation is

$$\gamma = \frac{\sqrt{\pi}}{2\rho}\sqrt{\frac{\rho}{R_c}}\ \frac{U^2}{V^2W^{1.5}}\ \frac{1}{K_1^2(W)} \cdot e^{\left(\frac{-4R_cW^3\Delta}{3\ \rho\ V^2}\right)} \tag{46}$$

For a graded profile fiber, which has a parabolic shape across the core, the bend loss coefficient is

$$\gamma = \frac{\sqrt{\pi}}{2\rho}\sqrt{\frac{\rho}{R_c}}\ \frac{V^4}{(V+1)^2\sqrt{V-1^2}} \cdot e^{\left(\frac{(V-1)^2}{V+1} - \frac{-4R_c(V-1)^3\Delta}{3\ \rho\ V^2}\right)} \tag{47}$$

These two equations contain several variables that represent certain physical parameters of an optical waveguide. The first equation has the term $K_1^2(W)$ and refers to the modified Bessel function of the second kind. The variable ρ is the cross-sectional radius of the waveguide itself, while R_c is the radius of curvature for this particular section of the waveguide. V (or V number) is a common optical waveguide parameter known as the *normalized frequency* and is calculated by the equation

$$\mathrm{V = k\rho\sqrt{n_{core} - \ n_{cladding}}} \tag{48}$$

where k is the wave number. The next two variables, U and W, are defined in Snyder and Love as the core and cladding parameters respectively. These two parameters rely on another common waveguide parameter, β, known as the propagation constant. The propagation constant determines how an optical wave travels in a waveguide and can be calculated by multiplying k with the effective index of refraction. The effective index of refraction is not a straightforward calculation, and is the overlap integral of a propagation mode with the physical structure of the core and cladding of a waveguide in which it travels. Both the propagation constant and effective index of refraction can be calculated by the software program RSoft or obtained using a Gaussian estimation from Snyder and Love. These methods will be discussed later.

With the propagation constant calculated, the equations for the core and cladding parameters are as follows

$$U = \rho\sqrt{k^2 n_{core}^2 - \beta^2} \tag{49}$$

$$W = \rho\sqrt{\beta^2 - k^2 n_{cladding}^2} \tag{50}$$

The last undefined parameter is Δ, which is referred to in Snyder and Love as the profile height parameter and calculated by

$$\Delta = \frac{n_{core}^2 - n_{cladding}^2}{2n_{core}^2} \tag{51}$$

The critical piece information that the combination of equations 45 and either equation 46 or 47 reveal is that for a given waveguide, output power is related to the radius of curvature by a double exponential function. Thus, small changes in radius of curvature can cause *very large* changes in bend loss, assuming the rest of the waveguide parameters remain the same. This will be very important to remember for the rest of the waveguide curvature analysis.

6.1.2 Algorithm Implementation and Demonstration

With the equations to determine bend loss over a waveguide section determined, a demonstration that the bend loss of a full waveguide can be obtained from the sum of the discrete sections is required. This is straightforward to show mathematically. Equation 45 shows that for a section of length z, the output power will equal the input power times the exponential bend loss factor. So, for a total section length z_{total}, equal to the sum of the length of n identical sections, the equation will be

$$P_{out} = P_{in} e^{-\gamma_{total}\, z_{total}} \tag{52}$$

To construct the equation for the combined waveguide of n discrete sections, the same equation is used, with the power out of the first section becoming the power in of the second section, the power out of the second becoming the power in of the third and so on, making the equation a product of all exponential terms like so

$$P_{out} = P_{in} \cdot e^{-\gamma_1 z_1} \cdot e^{-\gamma_2 z_2} \cdot \ldots \cdot e^{-\gamma_n z_n} \tag{53}$$

For a circular arc of n sections, the radius of curvature for all sections will be the same as the total arc. Since the value of γ only relies on the radius of curvature, assuming that all other waveguide parameters remain the same, then $\gamma = \gamma_1 = \gamma_2 = \gamma_n = \gamma_{total}$. Bringing the gamma value and the negative sign outside of the exponential product gives

$$P_{out} = P_{in} \cdot e^{-\gamma} \cdot e^{z_1} \cdot e^{z_2} \cdot \ldots \cdot e^{z_n} \tag{54}$$

The product of exponentials can be rewritten as a sum of the exponents, giving

$$P_{out} = P_{in} \cdot e^{-\gamma} \cdot e^{(z_1 + z_2 + \ldots + z_n)} \tag{55}$$

Since the sum of all individual sections is equal to z_{total}, it can be substituted into equation 55 to get equation 45. Thus, calculating bend loss as a product of discrete sections equals the bend loss that would be obtained for an entire section of similar length. This is trivial for a waveguide made up of circular arcs, as the radius of curvature does not change. For a waveguide based on a cubic spline or similar design, where the radius of curvature of each section *does* change, this is important, as it says the value of γ can be adjusted based on the radius of curvature of that particular section. Hence, the discrete radius of curvature values calculated for each waveguide segment can be used to calculate the bend loss for that particular waveguide section. The product of all bend losses for each section should give an accurate estimation of the bend loss for the entire waveguide.

Although these are three-dimensional waveguides being constructed, it's worthy to note that the curvature and bend loss calculations only involve a two-dimensional model, as two intersecting line segments will always be in their own plane. Therefore, bend loss computed in two dimensions can provide an accurate representation of the problem.

6.2 Comparison of Curvature for Spline and Arc Based Waveguides

Now that the importance of a waveguide's radius of curvature and its effect on power throughput has been discussed, the radius of curvature for both the spline and arc methods will be compared. A waveguide using the arc based design method in Section 5 was created with the same input and output positions as the sidestep spline waveguide shown in Figure 25. Figure 33 below shows the new arc based primary waveguide curve in blue along with the spline based sidestep in green for comparison.

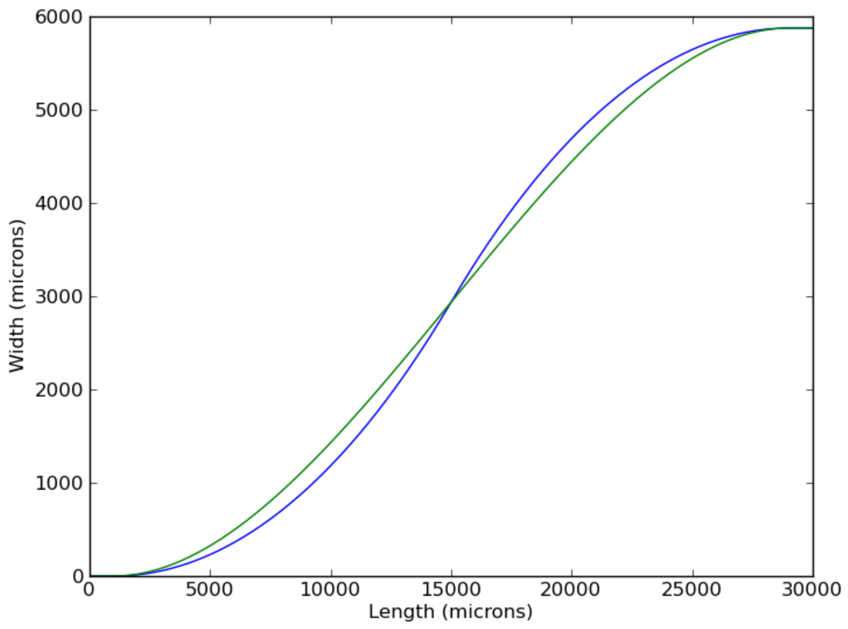


Figure 33 – Comparison of circular arc (blue) and cubic spline (green) based waveguides

Plotting the values of the radius of curvature for the two waveguide designs gives the graph below in Figure 34. This shows the blue arc-based waveguide has a larger minimum radius of curvature than the green spline-based waveguide.

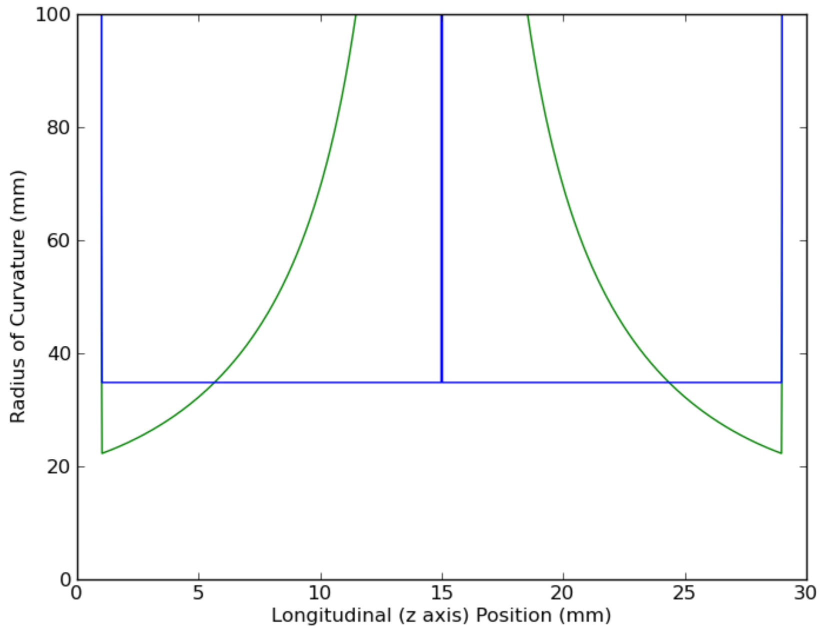


Figure 34 - Radius of curvature profile for arc waveguide (blue) vs. spline waveguide (green)

The spline exhibits dramatically greater curvature near the ends of the trajectory with little to no curvature in the middle. For the arc based curve (in blue), the values for the radius of curvature are the same for all points on the curve. This was what one would expect as the curves were made from a circular arc with a constant radius. It does have one point in the center section, where the curves join at a tangent, which is straight ("infinite" radius of curvature). Comparing the arc curve versus the spline curve shows that for most of the waveguide, the arc actually had a lower radius of curvature over the waveguide. However, the spline based curvature had a much lower curvature near the endpoints, with the values at the endpoints being around 23 mm for the spline curve versus around 35 mm for the arc based curve.

A secondary arc was created using the same endpoints as the spline based curve in Figure 26 and the radius of curvature also examined. First, the radius from the creation of the primary curve in Figure 33 was found to be 34.8 mm, and the total path length from this curve was found to be 30,814 µm. This value was passed into the secondary, three arc creation algorithm, and the path length of the resulting curve calculated. The initial value of the path length for the secondary guide was 30,620 µm. To match the path length, the radius for the secondary curve was decreased by 35 µm increments until the path length was matched to the primary waveguide. The final radius for the secondary waveguide, with path length matched, was 25.9 mm. With the path length matched, the result is the blue waveguide curve shown below in Figure 35. The spline based waveguide from Figure 26 is shown in green for a comparison.

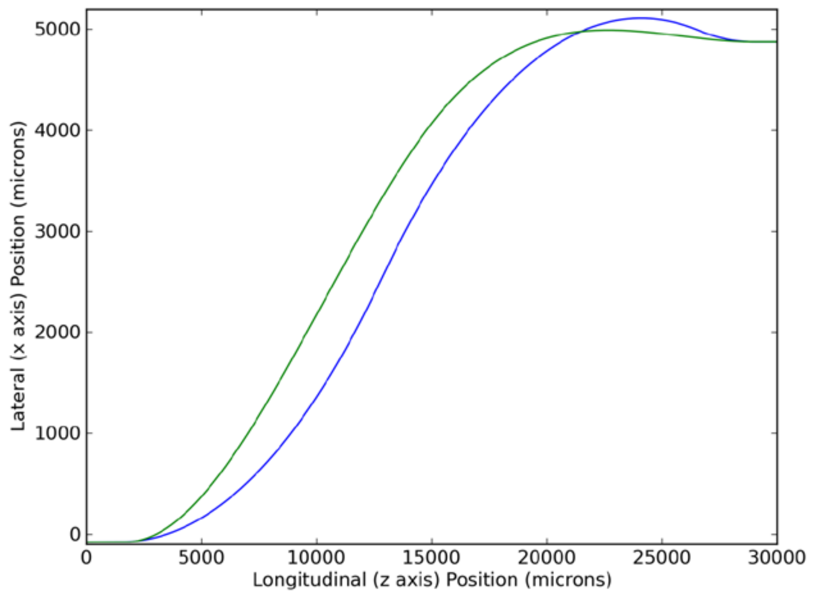


Figure 35 – Comparison of spline (green) and arc (blue) based waveguides with same endpoints

The radius of curvature values for the spline (green) and arc (blue) based waveguides are shown below in Figure 36 for comparison.

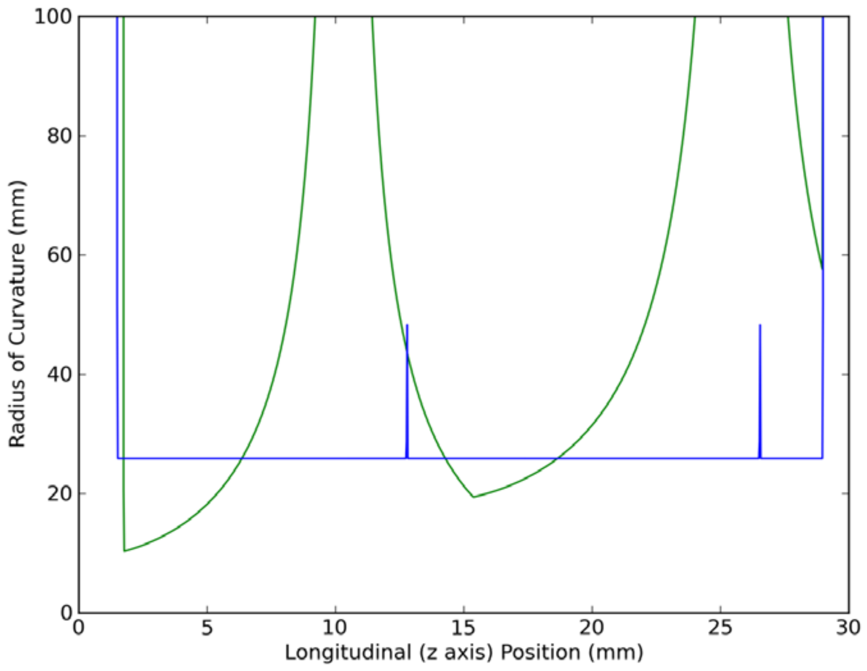


Figure 36 - Comparison of radius of curvature for spline (green) and arc (blue) waveguides

The radius of curvature graph shows similar results to the primary waveguide comparison. The arc based waveguide shows a flat radius of curvature at 25.9 mm over the length of the curved portion of the waveguide. There are two spots where the waveguide radius of curvature increased, coinciding with the transition points between the three arcs as discussed previously. The spline curve had a higher overall radius of curvature than the arc based curve but has two areas where the radius dips below the arc radius of curvature. The lowest point reaches a value of the radius of 10.3 mm. This now raises the key issue that must be further investigated - whether the sections with lower radius of curvature contribute to a higher overall bend loss for the waveguides designed with a cubic spline than the circular arcs.

6.3 Bend Loss Comparison

Using the formulas discussed previously, a few tests were performed to see what happens to the bend loss while varying several parameters. Since the waveguides thus far have been constructed on a 30 mm long glass block, a sample waveguide length was chosen of 30 mm. The radius of curvature was varied from 1 mm to 40 mm in 1 mm increments for a 30 mm long waveguide and the bend loss over the range of radii plotted. To see the effects of changing the waveguide parameters themselves, the difference between the core index of refraction and a constant cladding index of 1.4877 was set to three values of 0.005, 0.004 and 0.003. For each of these index values, Beta coefficient values of 6.041393049, 6.03812908, and 6.035121234 were used, respectively. These values were determined using a step index waveguide in RSoft with a waveguide radius of 4.85 µm. Three curves will then be created, so the effects of changing the core index of refraction can be determined on the bend loss.

6.3.1 Step Index Waveguides

The first test was performed using the step index bend loss Equation 46 with the resulting graph shown below:

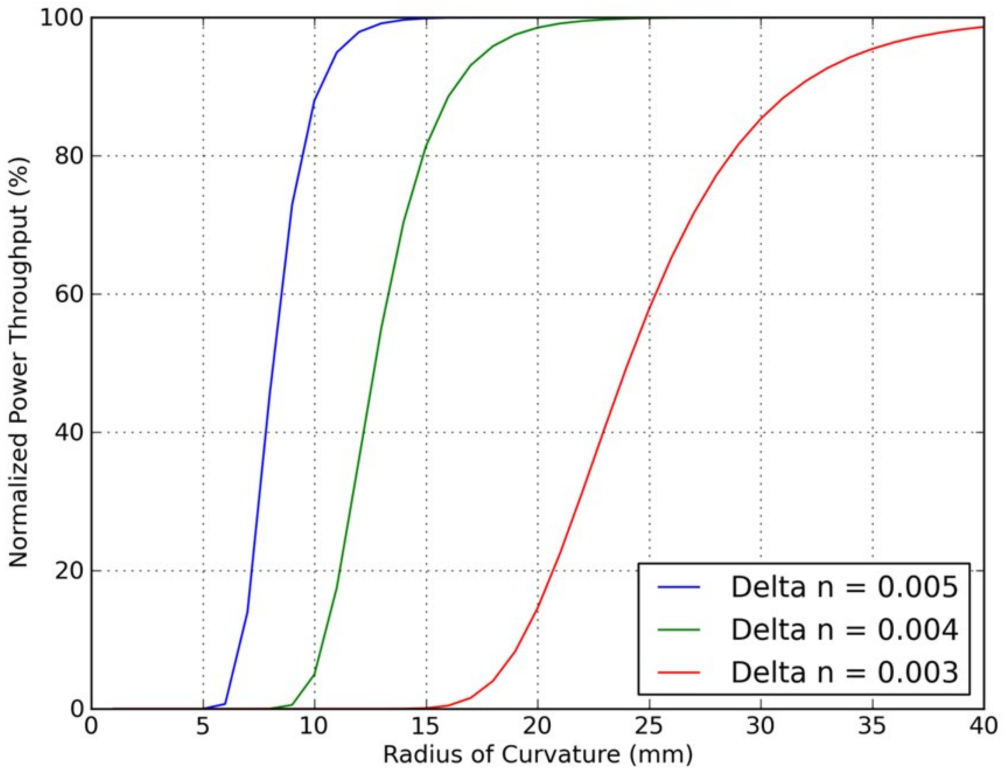


Figure 37 - Power throughput versus radius of curvature for a 30 mm step index waveguide at three different values of core index of refraction

As expected, the power throughput at all values of the core index of refraction show that as the radius of curvature of the waveguide is increased, the power throughput is increased. Also, as the difference in index of refraction between the core and the cladding decreases, the range of radii of curvature over which power transmission rises from 0 – 100% increases. This is intuitive as the index of the core decreases, more light will be lost in waveguide bends, so to contain the same amount of light, the waveguide must have less curvature. These curves show the existence of a steep slope where the power throughput of the waveguide can go from 100% to 0% with only a decrease in the radius of curvature of 10 mm over the length of the waveguide.

Using this same data, a plot of bend loss per mm of waveguide length was created, as shown in Figure 38 This shows the bend loss in decibels over a range of radii of curvature from 5 – 40 mm.

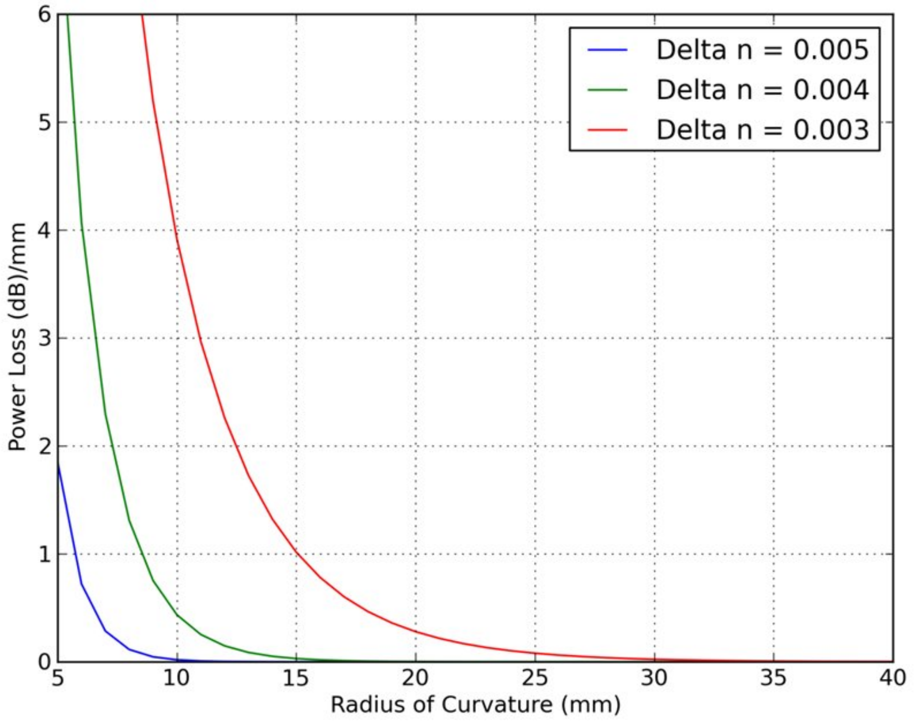


Figure 38 – Power loss vs. radius of curvature in decibels for a 1 mm step index waveguide at various values of core index of refraction

Because of the double exponential nature of the bend loss equation, even when plotted in decibels, a logarithmic measure, the bend loss per mm of the three different delta n curves rises in an exponential curve. As a reference example, with a radius of curvature of about 8 mm, a 1 mm step index waveguide, with a delta n equal to 0.003, will lose 75% of its optical power. This reinforces the importance of avoiding low radius of curvature when designing optical waveguides.

6.3.2   Graded Index Waveguides

The same set of graphs, as in Figure 37 and Figure 38, was produced for a graded index profile, calculated by equation 47

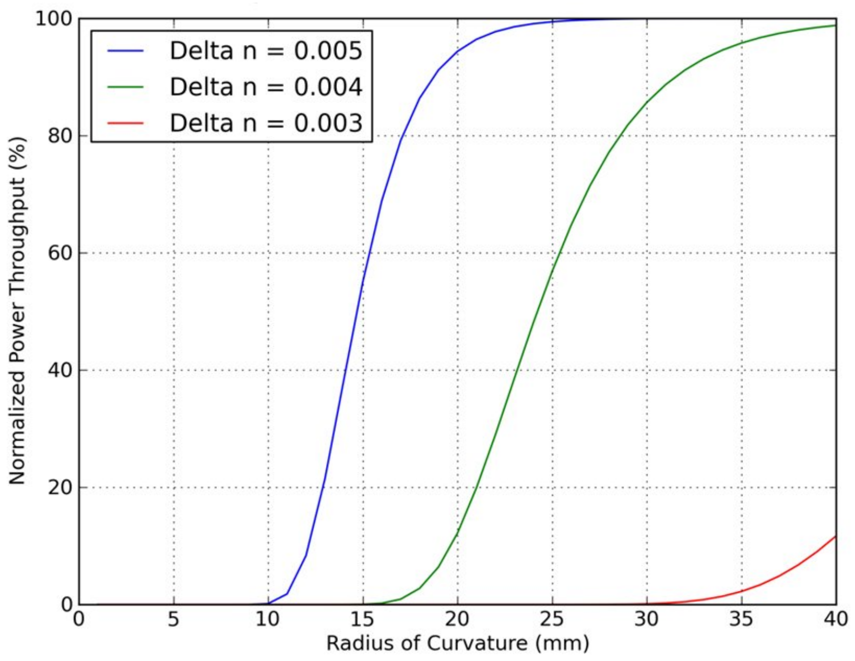


Figure 39 – Power throughput vs. radius of curvature for a 30 mm graded index waveguide at various values of core index of refraction

The graded index profile has increased bend loss for the waveguides at all values of delta n, shifting the curves to the right, with an extreme shift for a delta n of 0.003. The slope of these curves isn't as steep, meaning that a shift in the radius of curvature will have less effect on the power throughput than for a step index profile.

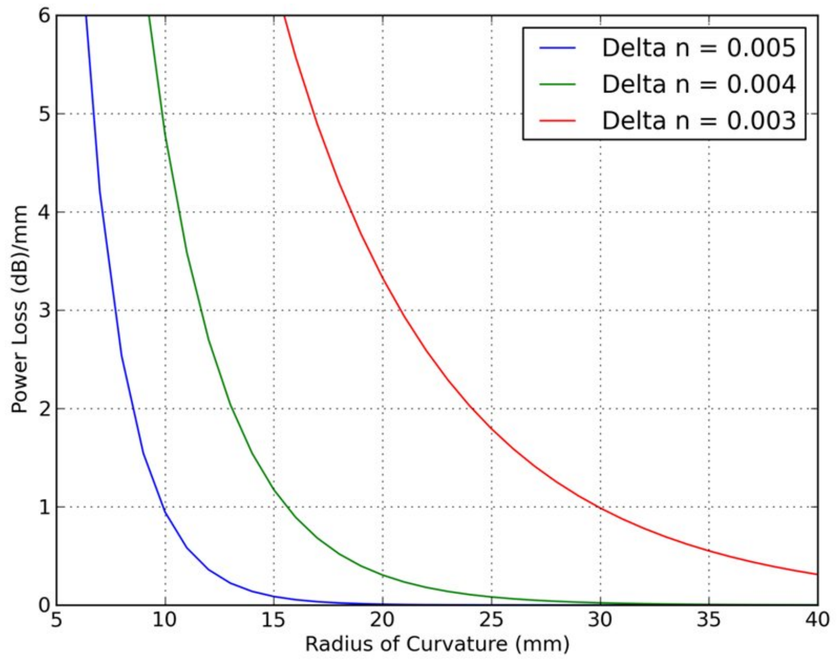


Figure 40 - Power loss vs. radius of curvature in decibels for a 1 mm graded index waveguide at various values of core index of refraction

Plotting the bend loss per mm for the graded profile gives the graph in Figure 40. The bend loss per mm curves here show that the loss is much greater for a graded index waveguide than the step index waveguide. The curves here are shifted to the right over the curves in Figure 38. So, a graded profile waveguide is definitely more prone to bend losses in a waveguide than a step index profile. From a discussion point, this makes sense as step index profile has a higher overall average of refractive index modulation as opposed to the graded profile, which tapers off at the edges.

These graphs further illustrate the importance of radius of curvature and how it affects power throughput due to bending in the waveguide. A comparison of the power throughput for both the spline and arc waveguides will be made in the next section, after investigating all waveguide power loss mechanisms.

7. Estimated Power Loss Calculations

The two different design methods have now been analyzed in terms of loss due to bends in the waveguides. However, to get a complete theoretical description of estimated power throughput, and additional loss process must be characterized and quantified: *transition loss.*

7.1 Transition Loss

Transition loss results from an abrupt difference in curvature in an optical waveguide. When a mode is traversing a section of a particular curvature and encounters a section of different curvature, there is a mismatch between the optical fields, leading to a radiation loss. There are three scenarios in which this can occur. First, a waveguide can go from a straight section to a curved section and vice-versa. Second, a waveguide can go from a section of a particular curvature in one direction and then transition to a section of the same or different curvature, but in the opposite direction. Third, a waveguide can go from a section of a particular curvature to another section of a different curvature in the same direction. These three scenarios are visualized below, along with the modal fields for the differing section, in Figure 41.

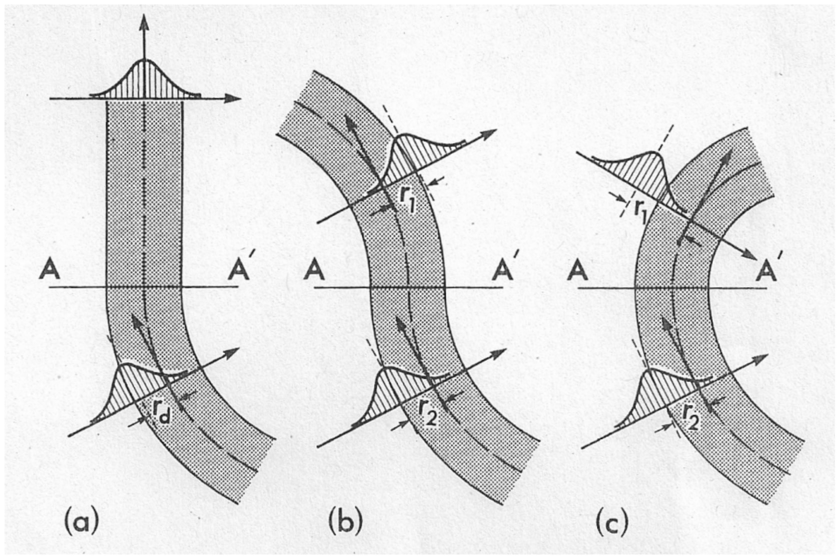


Figure 41 - Transition loss illustrations (a) straight to curved, (b) curved to opposite curvature, (c) curved to different curvature value in same direction (courtesy: *Snyder and Love*)

7.1.1 Loss Equation

Snyder and Love [31] give two equations for calculating transition losses. The first is for a transition between a straight section and a curved.

$$P_{out} = P_{in}\left(1 - \frac{1}{R_c^2}\frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right) \tag{56}$$

The second equation is for a transition between two sections of different curvature.

$$P_{out} = P_{in}\left(1 - \left(\frac{R_1 \pm R_2}{R_1 R_2}\right)^2 \frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right) \tag{57}$$

For the second equation, if the two curved sections are in the same direction, then the two radii should be subtracted. If the bends are in the opposite direction (180° change of direction), then the two radii should be added. The only variable here that wasn't previously defined in section 6.1.1 is the parameter r_0 which is the spot size of the waveguide mode. The determination of the spot size is not straight forward, but in this case, can be determined by a Gaussian approximation given again in Snyder and Love.

$$r_0 = \frac{\rho}{\sqrt{V-1}} \tag{58}$$

7.1.2 Algorithm Implementation

As has been discussed previously, the waveguides here are based on discrete sections. Transition loss is determined by comparing the radii of curvature between two adjacent sections, and then using either equation 56 or 57, appropriately, to determine the transition loss. If the radii of curvature between two sections are the same, including a straight section with an infinite radius of curvature, then obviously, the transition loss are zero and can be ignored. For the arc based waveguides designed previously, there is actually no transition loss across most of the waveguide, since the radius of curvature is identical at all points along a circular arc. However, there are a few key sections where the transition loss will occur in a basic arc based waveguide.

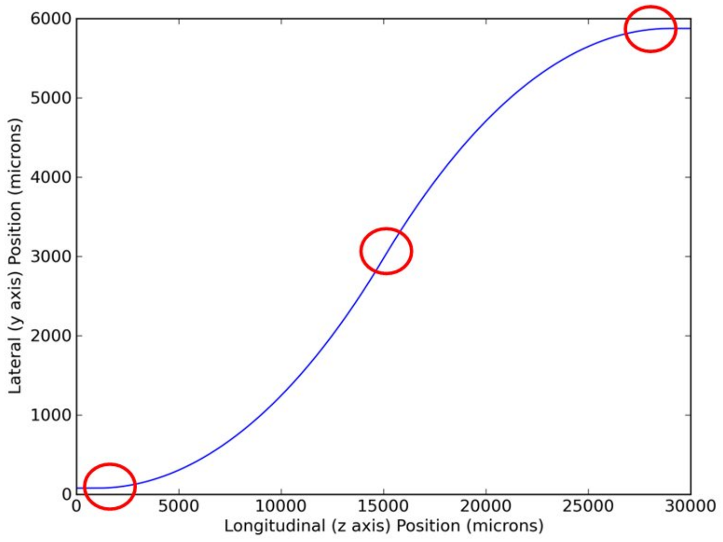


Figure 42 - Circular arc waveguide with transitions highlighted

There are two points where the straight end sections transition into the curved section, (equation 56 calculates this loss), and there is one section in the center of the primary waveguide where two arcs of the same curvature but opposite direction intersect with each other (equation 57). These regions are highlighted in Figure 42.

Comparing equations 56 and 57 shows that the only two differing terms between the two are

$$\frac{1}{R_c^2} \quad \text{vs.} \quad \left(\frac{R_1 \pm R_2}{R_1 R_2}\right)^2 \tag{59}$$

Since the waveguide above has the same radius of curvature throughout, but in opposite direction, the right term reduces to:

$$\left(\frac{R_c + R_c}{R_c R_c}\right)^2 = \left(\frac{2R_c}{R_c^2}\right)^2 = \left(\frac{2}{R_c}\right)^2 = \frac{4}{R_c^2} \tag{60}$$

So, for these circular arc waveguides, the center transition between two opposite arcs has four times the transition loss than the straight to curved transition loss given by equation 56. Like the bend loss, the total transition loss will be the product of each individual transition loss. If the individual transition loss is taken from equation 56, the output power after all transition losses in the waveguide will be

$$P_{out} = P_{in}\left(1 - \frac{1}{R_c^2}\frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right)^2\left(1 - \frac{4}{R_c^2}\frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right) \tag{61}$$

This equation is for a waveguide based on two circular arcs. For a path length matched waveguide based on three circular arcs, as in Figure 29, there will be an additional transition from opposite arcs, giving an additional second term as

$$P_{out} = P_{in}\left(1 - \frac{1}{R_c^2}\frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right)^2\left(1 - \frac{4}{R_c^2}\frac{\rho^2 V^4}{8\Delta^2}\left(\frac{r_0}{\rho}\right)^6\right)^2 \tag{62}$$

These equations allow the entire transition loss for arc based waveguides to be calculated from one equation, making things simple and saving computation time. For a waveguide with constantly varying radius of curvature, such as one constructed from a cubic spline, the computation must be performed for each discrete section. Revisiting Figure 25 to illustrate transition losses, the image below shows that the cubic spline waveguide has a transition from straight section to curved waveguide at the points on the waveguide that have the lowest radius of curvature. This will contribute a significant transition loss. However, for the rest of the waveguide, the transition loss will be low.

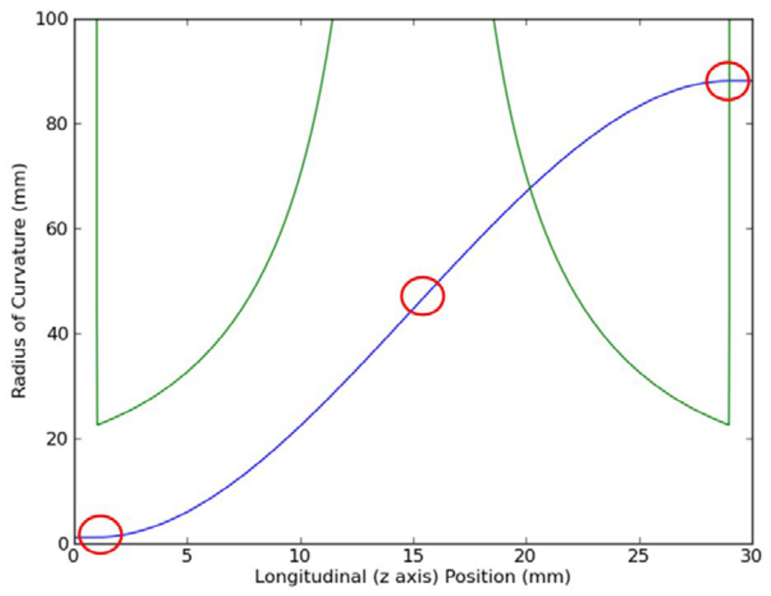


Figure 43 - Cubic spline based waveguide transition illustration

For the circled center transition, where the waveguides transition in different directions, the actual radius of curvature is very high, meaning the transition loss is very low. For the rest of the curved waveguide, the $\left(\frac{R_1 \pm R_2}{R_1 R_2}\right)^2$ term will be a subtraction of the two radii, as the curve is in the same direction. Therefore, for cubic spline based curves the transition loss will be very small along most of the waveguide.

7.2 Transition Loss Comparison

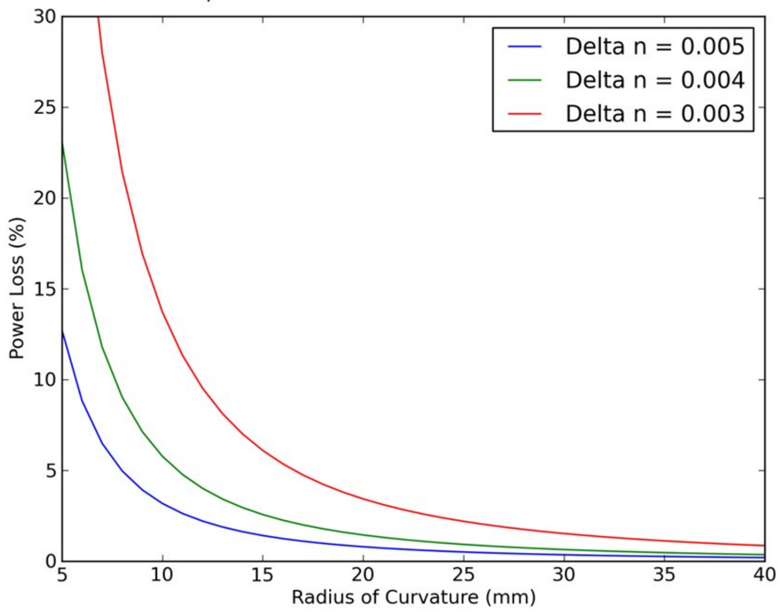


Figure 44 - Transition loss from straight section to curved section of specified radius at various values of delta n

As the equations in the previous section show, transition loss does have a dependence on the physical parameters of the waveguide itself. To show the effects, the loss for a transition from a straight waveguide to a curved waveguide was tested at three different values of the difference in the index of refraction between the core and the cladding and the radius of curvature of the curved waveguide varied between 5 and 40 mm. The results are shown in Figure 44. The curves created also show the inverse square relationship of transition loss to the radius of curvature. The graph also shows the trend that the transition loss increases as the change in index of refraction decreases. The transition loss equations 56 and 57 hold regardless of the waveguide index profile or shape, so the results would be the same for step, graded, or any other index profile. The next graph in Figure 45 is for a transition from a curved waveguide section of radii from 5 – 40 mm to a curved waveguide section of the same curvature in the opposite direction, e.g. 5 mm to -5 mm, 6 mm to -6 mm, etc.

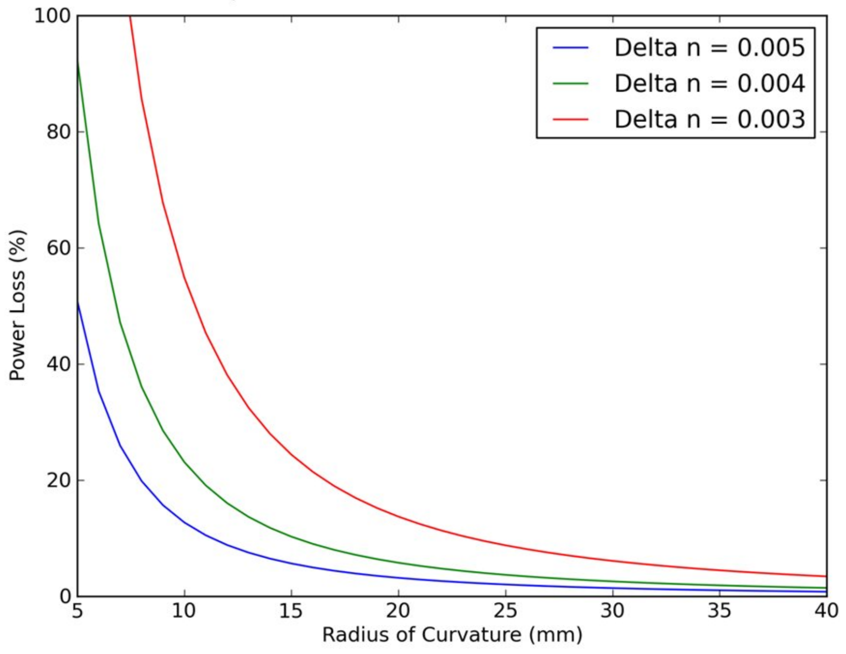


Figure 45 - Transition power loss from curved to opposite curved section of specified radius at various values of delta n

The curves in this graph have the same shape, except the vertical scale is different and is exactly four times the scale of the graph in Figure 44. This matches the calculations made in Section 7.1.2. Both the arc and spline waveguide sets have two straight to curved transitions, with the spline waveguides having a higher transition loss due to their higher curvature at that transition. However, the arc based waveguides have either one or two curve-to-opposite-curve transitions which have four times the transition loss of the straight-to-curved transition.

This graph also illustrates an important point in that for lower values of delta n, the transition loss reaches 100%, and if extrapolated further, would go over that amount, which is impossible as you can't have more than 100% power loss. Reviewing equations 56 and 57 also shows that for sufficiently small radii of curvature, these equations actually predict a *negative* power. This means that these equations should only be used for sufficiently high values of radii of curvature as they breakdown below a certain value, depending on the index of refraction of the waveguide.

This section showed that the loss from a transition is based on a second power term to radius of curvature. In contrast, the bend loss was showed to have dependence on a double exponential relation to the radius of curvature, and thus it is the more critical point of focus. The next section will show how the power throughputs for both the spline and arc based waveguides are affected by both bend and transition losses.

7.3 Total Power Loss for Waveguide Sets

The two power loss mechanisms above can be combined to give a total power loss value. For comparison, the set of eight sidestep waveguides previously created using the cubic spline method in Figure 22 will be analyzed and compared to a set, with identical endpoints, created with the arc method (shown later in Figure 58). These will be tested at the three different values of core index of refraction as in the previous section, giving a total power throughput at each index value. The effects of the step and graded index profile will both be examined. To calculate bend loss for each waveguide, the algorithm calculates the three-dimensional coordinates of the waveguide curve in 25 µm along the z axis, as mentioned previously. Over 30 mm, this gives 1200 points. At each point, the local radius of curvature and the bend loss coefficient γ are calculated and multiplied by the segment length z between points, with the throughput power calculated per equation 45. This calculation is then repeated over all 1200 points to give the total waveguide power throughput. Likewise, the transition loss is calculated incrementally for the cubic spline waveguide and repeated over the entire waveguide.

The first graph in Figure 46 shows the total combined power throughput of the two sets of eight waveguides with a step index profile. The arc and spline waveguides are both displayed at the three different values of core index of refraction. For each set, the waveguides are ordered from lowest throughput to highest throughput. Here, the power throughputs for the delta n values of 0.005 and 0.004 give a very slight power throughput advantage to the cubic spline based waveguides. However, for a delta n value of 0.003, the arc based waveguides have a clear power throughput advantage over the cubic spline designs.

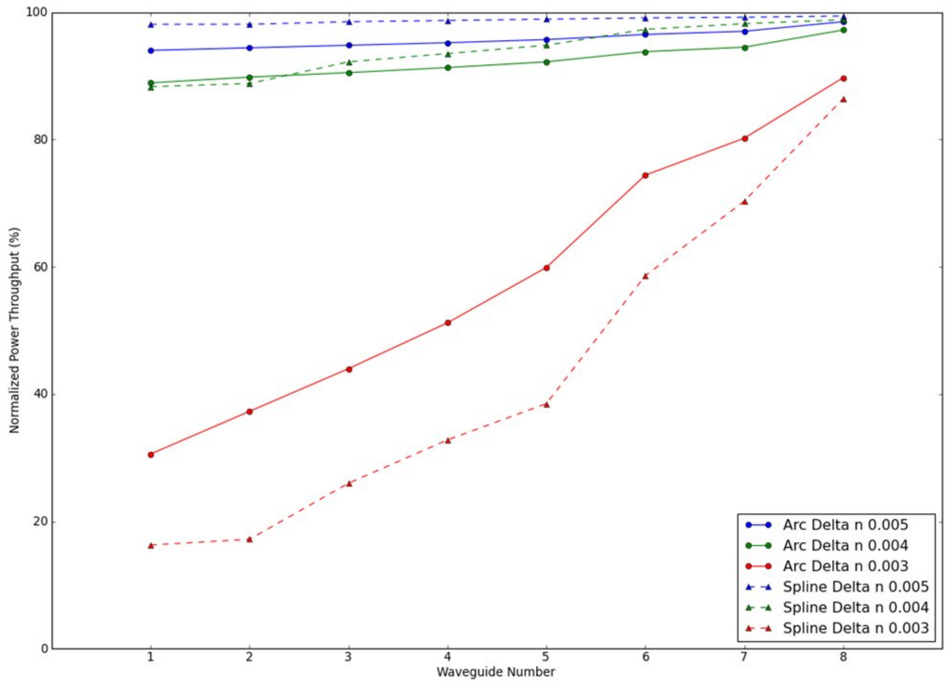


Figure 46 – Power throughput for 8 step index profile waveguides using both arc and spline technique at various values of core index of refraction

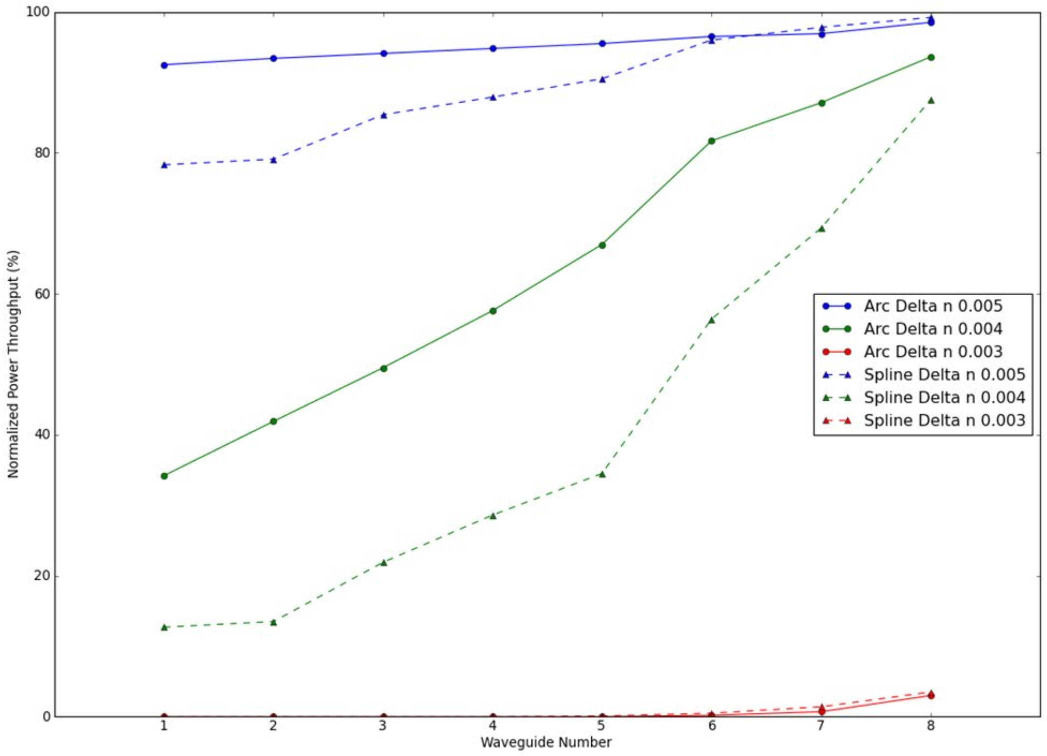


Figure 47 - Power throughput for 8 graded index profile waveguides using both arc and spline technique at various values of core index of refraction

For the graded index profile waveguides in Figure 47, the arc based waveguides have a clear advantage in power throughput values except for the delta n value of 0.003, where the power throughput is near zero for all waveguides. To determine why the total power throughput values differ between the two designs, the power throughput for all waveguide designs was measured for bend and transition losses, as well as the total power throughput after the contribution of both loss factors. The average across all eight waveguides of each waveguide set was taken with the values given in Table 2 and Table 3. These tables show that the total throughputs for the arc waveguides are lower for the high values of delta n, in a step index profile, due to the transition loss. At these values, the bend losses for both arc and spline waveguides are near zero and the spline waveguides have much less transition loss. For the lowest step index delta n value, the increased bend loss of the spline based design becomes the much larger loss factor, giving a greater total power loss for these designs compared to the arc waveguides.

Waveguide Set	Bend	Transition	Total
Arc - 0.005	100.0	95.7	95.7
Cubic Spline - 0.005	99.9	98.8	98.8
Arc - 0.004	99.9	92.4	92.3
Cubic Spline - 0.004	96.0	97.9	94.0
Arc - 0.003	69.5	82.6	58.4
Cubic Spline - 0.003	45.1	95.0	43.3

Table 2 - Average power throughputs for step index profile waveguides, broken into bend and transition loss components

Waveguide Set	Bend	Transition	Total
Arc - 0.005	99.5	95.7	95.3
Cubic Spline - 0.005	90.3	98.8	89.3
Arc - 0.004	68.8	92.4	64.1
Cubic Spline - 0.004	41.3	97.9	40.6
Arc - 0.003	1.0	82.6	1.3
Cubic Spline - 0.003	0.9	95.0	1.1

Table 3 - Average power throughputs for graded index profile waveguides, broken into bend and transition loss components

In the graded waveguide sets, the bend loss factor is much higher for the spline based designs at the higher delta n values. The bend loss for a delta n value of 0.003 here is so large that the bend and total losses are near 100%. Note that for both the step and graded index profile waveguides, the transition losses are the same since the shape of the waveguide profile does not play a part in these loss mechanisms.

These tables show that as the difference in core index of refraction becomes less, bend loss plays a more important role. Hence, the use of the arc based waveguide technique is better suited, as waveguides with the same input and output coordinates can have much less bend loss. However, these simulated estimates are for a step or graded index profile and not the actual direct write profile that will be used for fabrication. To simulate the direct write profile requires a more advanced numerical simulation tool, known as RSoft, which is the subject of the next chapter.

8. Waveguide Simulation using RSoft

As discussed briefly in the introduction, RSoft is a software simulation tool that allows for testing and measurement of optical waveguides. RSoft is based around the Beam Propagation Method and gives an estimate of the power throughput for a given set of waveguide conditions. The software has the capability to test step index profiles and also has a Gaussian index profile that closely replicates a graded index profile, allowing for a comparison between methods. The first test was to compare the bend losses from RSoft to the values obtained earlier with the theoretical formulas from Snyder and Love.

8.1 Bend Loss

Testing the step and graded index waveguides was performed by coding a straight 30 mm waveguide in RSoft's BeamPROP program and then utilizing a "simulated bend" facility. As described in Section 2.5, bend loss can be calculated by simulating the optical effects of a bend in a waveguide, rather than tackling the more computationally difficult modeling job of carrying the field propagation through a rotating coordinate frame which follows the guide. To match the Snyder and Love calculations of bend loss only for the waveguide, each time the simulated bend radius is changed, the propagating mode needs to be recalculated for that radius and injected into the waveguide at the beginning. This will eliminate the effects of a straight mode injection into a bent waveguide, adding an extra, unwanted transition loss on top.

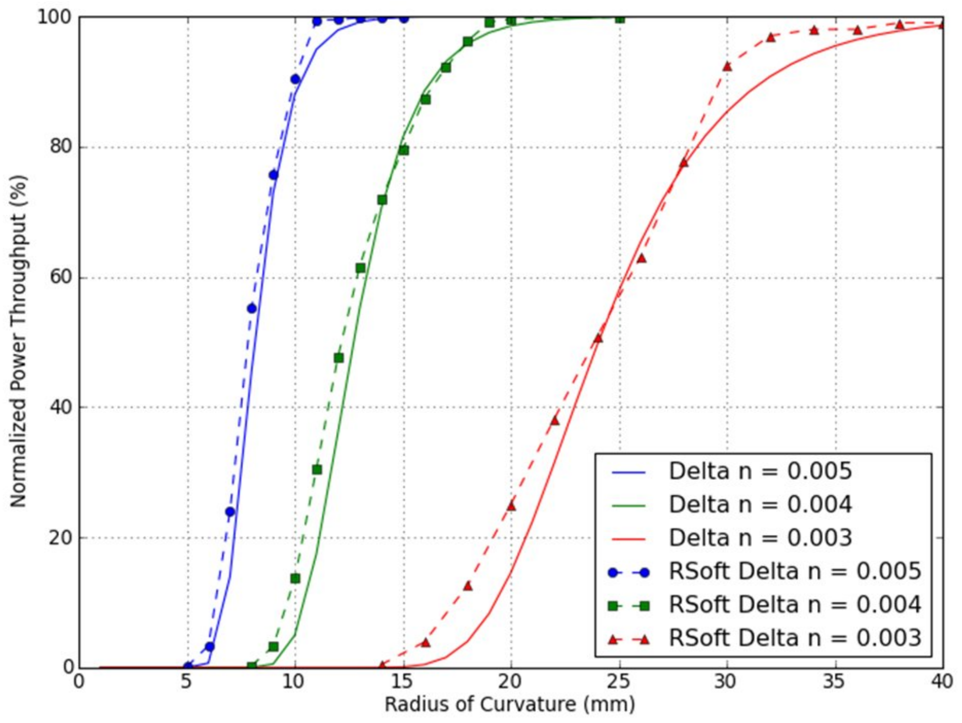


Figure 48 – Power throughput vs. radius of curvature for a 30 mm step index waveguide at various values of core index of refraction with RSoft results added

The bend loss was tested over the same range of radii of curvature from 1 – 40 mm at three values of the core index of refraction, giving a delta n of 0.005, 0.004 and 0.003. This was done for both a step index profile and a Gaussian index profile. These values were then plotted over the earlier bend loss curves shown in Figure 37 and Figure 39. The new graphs are shown in Figure 48 and Figure 49. Figure 48 shows that there is a very good correlation between the values for step index waveguide bend loss calculated from the Snyder and Love algorithms with the results from simulation in RSoft. There are small offsets between the two methods, especially at a delta n value of 0.003, but either method will give comparable results in the prediction of bend losses in these waveguides.

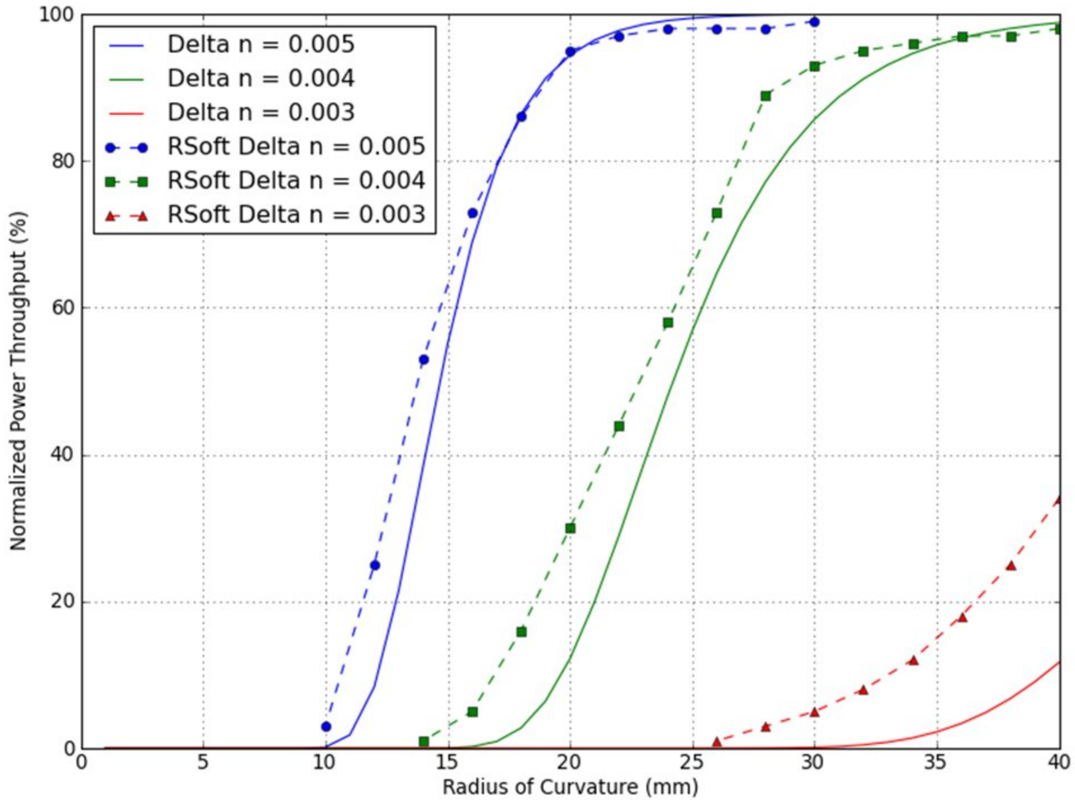


Figure 49 – Power throughput vs. radius of curvature for a 30 mm Gaussian index waveguide at various values of core index of refraction with RSoft results added

Figure 49 also shows a good correlation between Snyder and Love based algorithms and RSoft simulations results for a delta n value of 0.005, slightly degraded results for 0.004, but a larger variance for a delta n value of 0.003. So, either method should give a reasonable estimate of bend losses for Gaussian profile waveguides at delta n values of 0.005 and 0.004, however this degrades rapidly for a delta n value of 0.003 and becomes nearly unusable. The reason for this discrepancy was attributed to the difference in the determination of index profile width for a Gaussian profile in RSoft vs. the theoretical calculations. As the delta n becomes lower, the index profile widens and the larger change in width causes a discrepancy in the volume of the index profile. The RSoft profile was slightly larger and therefore had a higher power throughput.

8.2 Bend Loss for Waveguide Sets

The arc based waveguide set created in Section 5, was programmed with all bends and transitions into RSoft. The average calculated power throughputs for this set were given previously (in Table 2) for step index profile waveguides. To compare the simulated values from RSoft for a step index profile, the table was updated with the average total power throughput from the eight waveguides calculated by RSoft. The Gaussian index was not compared, as the power loss was too great for some values of core index of refraction, as previously discussed.

Waveguide Set (Style – delta n)	**Avg. Analytical Calculated Power Throughput (%)**	**Avg. RSoft Simulation Power Throughput (%)**
Arc - 0.005	95.7	97.3
Cubic Spline - 0.005	98.8	97.9
Arc - 0.004	92.3	93.8
Cubic Spline - 0.004	94.0	94.3
Arc - 0.003	58.4	59.4
Cubic Spline - 0.003	43.3	56.0

Table 4 - Comparison of calculated and simulated power throughput for waveguide sets

This table shows that for most of the waveguide configurations, the average calculated power and the average simulated power are within 2% of each other. The exception to this is the power values for the cubic spline at delta n of 0.003, where average simulated power throughput was 29% higher than the calculated power throughput. To model waveguides in RSoft using a simulated bend requires sections of a certain length to have a specific radius of curvature. A cubic spline waveguide has a constantly changing radius of curvature. To attempt to quantify this in RSoft, the analytical calculation algorithms were modified to produce a waveguide section at each calculated point, along with the radius of curvature, in an RSoft readable form. Where the arc based waveguides were constructed with four or five sections, the spline based waveguides were created using over 1200 sections. Thus, modeling of cubic spline waveguides in RSoft is difficult and the results do not compare to the calculated results at low values of delta n. For the arc based waveguides, however, either analytical calculations or RSoft simulations can be used effectively to simulate expected power throughputs for a given step index waveguide.

8.3 Modeling HPO Waveguides in RSoft

So far, it has been shown that total waveguide power losses can be calculated using analytical equations as well as simulated using RSoft, and that these two methods correlate well for the simplest case of a step index profile for the arc based design method. However, what is needed is a prediction method for waveguides fabricated using the direct write waveguide technique that has been discussed previously. The waveguides fabricated for the research in this thesis used a High Power Objective (HPO) laser, which has a unique index profile shape fabricated in the glass. This complex profile is due to the specific physics responsible for the index change in the glass substrate, as discussed further in *Jovanovic et al* [39]. An effective bend loss calculation for the HPO profile was not found in the literature, nor can it be effectively estimated using bend loss equations for simple geometries available. However, the index of an HPO waveguide can be coded into RSoft and the waveguides simulated using that index profile. An HPO index profile was measured by the fabrication laboratory at Macquarie University, creating a numerical profile to input into RSoft. An image of the index profile entered in RSoft is shown below in Figure 50, along with an image of the mode as it travels in the waveguide.

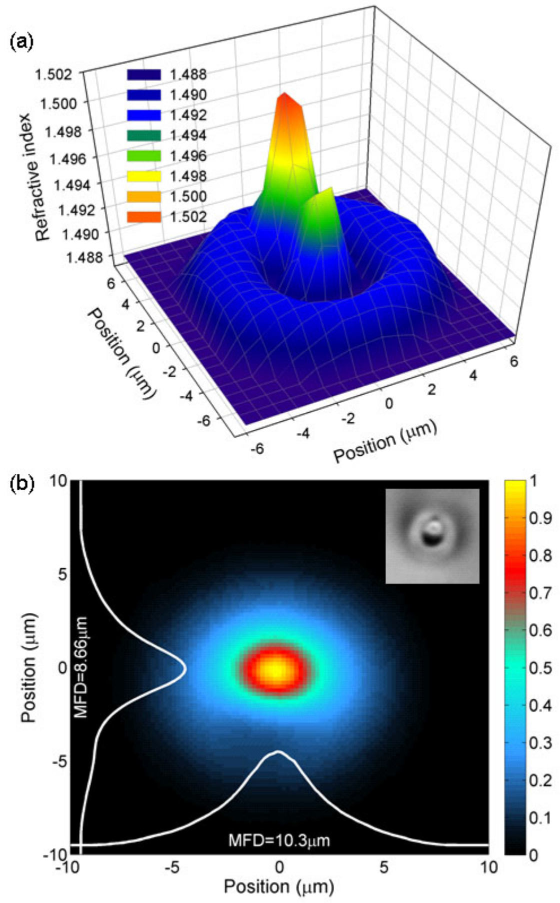


Figure 50 - (a) Refractive index profile of an ultrafast laser inscribed waveguide at 1550 nm. (b) Intensity distribution determined by RSoft of the guided waveguide mode in two dimensions, overlaid by a vertical and horizontal cut profile shown as the two white curves. Inset shows a cross-sectional micrograph of the waveguide which has been scaled down to fit in Fig. (b)

This shows a three-dimensional cross section of an HPO profile in the X and Y dimensions in image (a), with the y position on the left axis (the direction the laser is coming from) and the x position on the bottom axis. This profile reaches a peak delta n of 0.0143, but this peak is only about 2 microns in diameter, where the step and Gaussian profiles used thus far for estimates have a diameter of 10 microns. The image also shows that the HPO profile actually has a second, smaller peak, also, illustrating that the HPO profile is complex and is difficult to approximate using any standard shape. Image (b) shows a snapshot of the mode traveling in the waveguide indicating that despite its complex form, beams of light are observed to propagate effectively through this waveguide in a single-moded fashion.

8.3.1 HPO Bend Loss Simulation

With a mode obtained for the HPO waveguide in RSoft, the same bend loss testing performed with the step and Gaussian index waveguides can be performed using the HPO waveguides. The graphs created in Section 8.1 were updated with the results of bend loss testing for a 30 mm long HPO waveguide and given below.

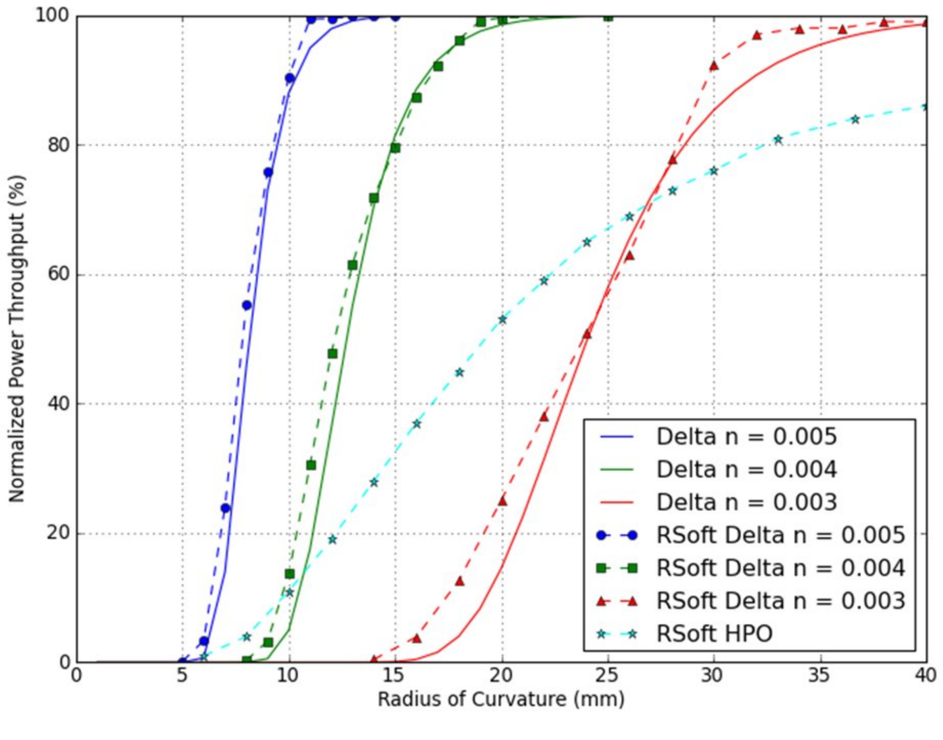


Figure 51 - Power throughput vs. radius of curvature for a 30 mm step index waveguide at various values of core index of refraction with RSoft results (including HPO) added

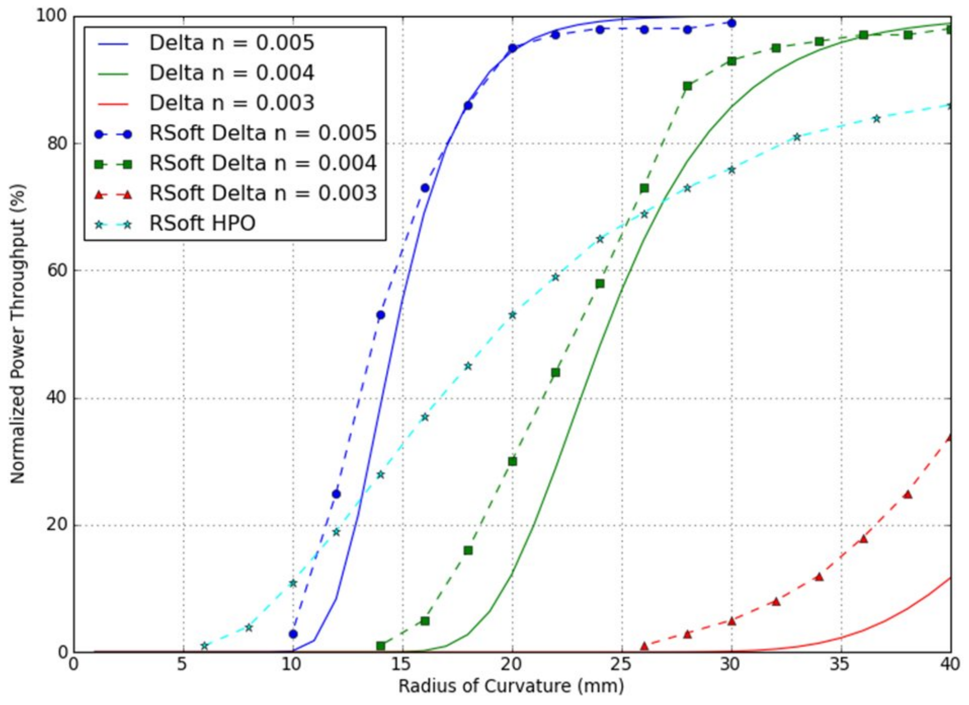


Figure 52 - Power throughput vs. radius of curvature for a 30 mm Gaussian index waveguide at various values of core index of refraction with RSoft results (including HPO) added

The updated graphs show that the bend loss curve for the HPO waveguide does not rise as steeply as the step or Gaussian index curves. Thus, neither index curve would make a good model to simulate the bend loss values for an HPO waveguide. To get an experimental estimate of bend loss for HPO waveguides requires fabricated waveguides. It also shows that the HPO profile does have fairly high bend losses, even at reasonably large radii of curvature. The follow-on effects of this bend loss profile will be investigated further in the next section.

9. Fabrication and Physical Results

To create the waveguide designs for fabrication, the positional data from all eight waveguides are translated into a format compatible with the fabrication laboratory computer. For more detail on the process of fabricating direct write waveguides, refer again to *Jovanovic et al* [39].

9.1 Physical Bend Loss Radius Scan Chip

To explore the bend loss for an HPO fabricated waveguide, a chip was created with right angle waveguides of increasing radius of curvature. Each track on this chip had two straight to bend transitions, and a series of 90° arcs all in a single plane (no change in height). The chip was designed and fabricated by Nemanja Jovanovic at Macquarie University and is illustrated below in Figure 53.

Radii (mm)	Lead in (mm)	Distance from the edge of the chip (mm)
13.3	15.7	28.0
16.6	13.9	29.5
20.0	12.0	31.0
23.3	10.2	32.5
26.6	8.4	34.0
30.0	6.5	35.5
33.3	4.7	37.0
36.6	2.9	39.5
40.0	1.0	41.0
Straight guides		41.15
		41.30

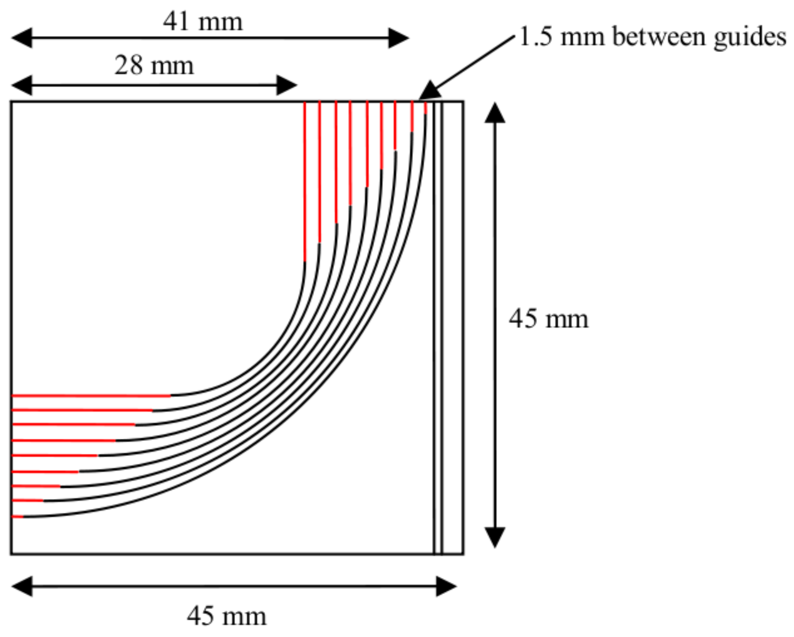


Figure 53 - Design of bend loss radius scan chip (courtesy: Nemanja Jovanovic)

This radius parameter scan chip allows a bend loss per unit length measurement to be derived for an HPO waveguide under conditions of varying curvature. This can then be used in a lookup table in the optimization algorithms to give a fairly accurate estimate for the bend loss of a given waveguide.

This chip was probed by injecting laser light at 1550 nm into each waveguide. Fibers with index-matched oil were coupled to both the waveguide input for injection and to the output where a probe fed an optical power meter to measure the transmitted signal. Figure 54 shows an image of the setup to measure optical power throughput, with red light at 630 nm used for the photograph to better illustrate a particular waveguide and together the injection and probe fibers. The waveguide shown being tested is a pupil remapper waveguide and not from the radius scan chip.

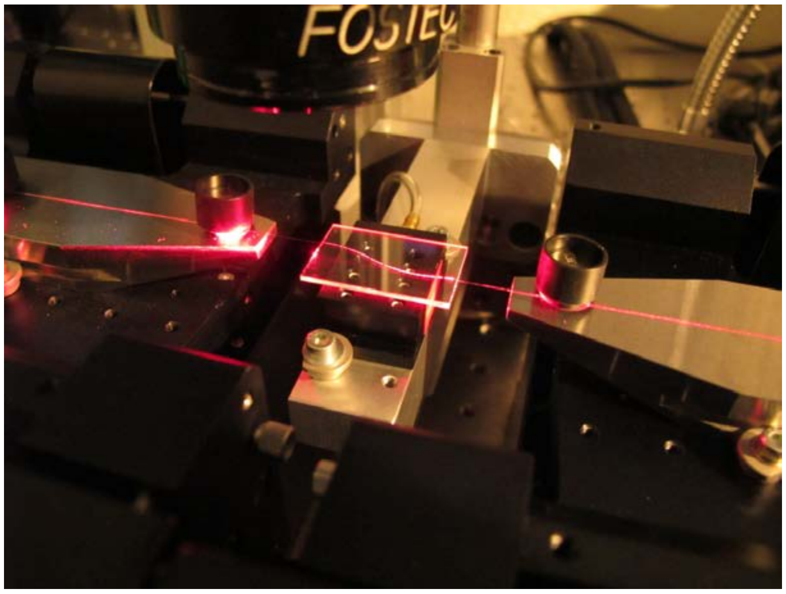


Figure 54 - Optical power measurement setup for measuring power throughput of waveguides

An optical power measurement was taken for each waveguide and then normalized to the power measured when coupling the two fiber optic probes directly together and setting that value to be a power throughput of 100%. These values are given in the third column in Table 5. Each waveguide has an absorption loss associated with it due to the intrinsic properties of the glass. The last remaining loss mechanism is coupling loss and typical coupling losses for all waveguides and measurement setup is around 9% for both coupled points. By measuring the power through a set of the straight waveguides fabricated on the right side of the chip, a value for the absorption loss without bends can be determined.

Removing the coupling loss from the straight waveguides gives the relationship for absorption loss of $(1 - e^{(-0.0075z)}) * 100\%$, where z is the length of a waveguide section in millimeters. Thus the glass absorption loss can be isolated by dividing the normalized throughput value by the above factor and the total length of the waveguide in column 2. The coupling loss can then also be removed and the remaining power loss should be solely due to bends in the waveguide, which is given in column 3. Subtracting the lead-in length in Figure 53 from the total length in column 2 gives the bend length for each waveguide. This length value can be divided by the power, and then the natural log applied to that value to give the bend loss per mm of length value in column 5. To get an estimated power throughput, this bend loss value can be used to give the power throughput for a hypothetical 30.7 mm waveguide at that bend radius, given in column

6. This length was chosen as this was the path length for the sidestep arc design waveguides. Finally, absorption and coupling could be added back in for the 30.7 mm waveguide to give the throughput in column 7.

Arc Radius (mm)	Total Waveguide Length (mm)	Normalized Throughput (%)	Remove Absorption And Coupling (%)	Bend Loss/mm	Throughput for 30.7 mm Waveguide (%)	With Absorption and Coupling (%)
40	62.83	42.9	75.4	0.00448	87.1	63.0
36.6	57.49	41.9	73.0	0.00549	84.5	61.1
33.3	52.31	40.9	69.3	0.00700	80.7	58.3
30	47.12	39.6	66.2	0.00875	76.5	55.3
26.6	41.78	34.9	57.7	0.01315	66.8	48.3
23.3	36.60	31.6	51.8	0.01800	57.6	41.6
20	31.42	17.6	28.4	0.04005	29.2	21.1
16.6	26.08	8.0	12.8	0.07880	8.9	6.4
13.3	20.89	1.4	2.3	0.18151	0.4	0.3

Table 5 - Results from power probing of arc radius scan waveguides

To compare these physical results, the waveguides in Figure 53 were replicated in RSoft. The power throughput from RSoft simulations for each waveguide is shown in column 3 of Table 6.

Arc Radius (mm)	Arc Length (mm)	Simulated Power Throughput	Bend Loss/mm	Throughput for 30.7 mm Waveguide w/ A & C (%)
40	62.83	0.77	0.00416	63.6
36.6	57.49	0.74	0.00524	61.5
33.3	52.31	0.71	0.00655	59.1
30	47.12	0.67	0.00850	55.7
26.6	41.78	0.62	0.01144	50.9
23.3	36.60	0.56	0.01584	44.4
20	31.42	0.5	0.02206	36.7
16.6	26.08	0.43	0.03237	26.8
13.3	20.89	0.37	0.04759	16.8

Table 6 – Results from RSoft simulations for arc radius scan waveguides

As RSoft does not incorporate the absorption and coupling losses, the loss in the waveguide will be due to bend losses alone (ignoring transitions losses for the moment). Thus, the natural log can be taken of the power throughput and divided by the arc length to get the bend loss per mm of length values in column 4. This bend loss value can be used to estimate the power throughput for a 30.7 mm long with waveguide and adding in absorption and coupling to get the values in column 5. If the RSoft simulated and physically measured results for bend loss per mm of length from both Table 5 and Table 6 are plotted together on a log/linear graph, the result is shown below in Figure 56.

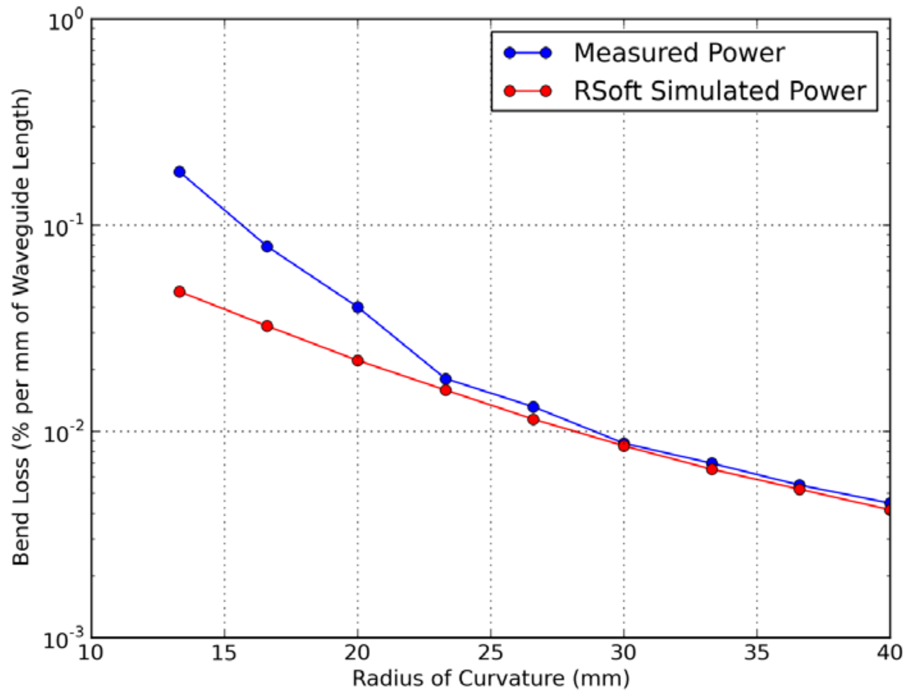


Figure 55 - Bend loss per mm of length vs. radius of curvature for physically measured (blue) and RSoft simulated (red) waveguides

This graph shows that the bend loss per mm of both the simulated and measured waveguides have a very good correlation for high values of radius of curvature and this curve is nearly linear when plotted on a logarithmic scale. However, for radius of curvature values below about 23 mm, the bend losses for the physically measured waveguides increase sharply over the simulated results. Thus, the fabrication process may be showing limitations and imperfections when waveguide radius of curvature values fall below 23 mm.

These bend loss values can be used compare the simulated and physical extrapolated results for a theoretical 30.7 mm waveguide. These values are taken from the above table with coupling and absorption loss added and the resulting graph is shown in Figure 56.

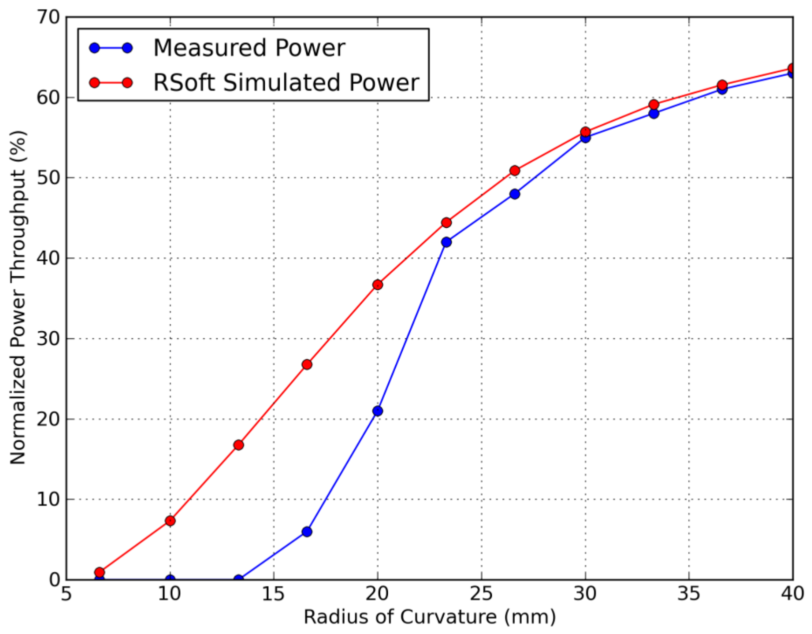


Figure 56 - Simulated vs. measured power throughputs for 30 mm HPO waveguide

This graph also shows that the measured and simulated measurements for bend loss correlate very well for radius of curvature values 23 mm and above. For radius values below that figure, the physical bend loss diverges somewhat from the simulated result and the power throughput decreases significantly.

The results from the fabrication of the arc radius scan chip waveguides, and the comparison with simulations of those waveguides, show two important things. First, that for radius of curvature values down to 23 mm, bend losses in waveguides can be accurately predicted by RSoft. Second, as the physical waveguide bend losses increase dramatically for radius of curvature values less than 23 mm, waveguides using these low radii of curvature values should be avoided. Furthermore the fabrication results could be difficult to predict using RSoft.

A consequence of this requirement for low curvature is illustrated in Figure 36, which compares two waveguides created from the arc and spline based sidestep design. It can be seen that the radius of curvature for the arc waveguide sits just below 23 mm across the entire waveguide, while the spline based waveguide has a tightest curve with down to nearly 11 mm. This region, circled in red, illustrated that the arc based design should be used for the high curvature values in preference to the spline for sidestep chips. Thus, the circular arc waveguides were the basis for all further designs.

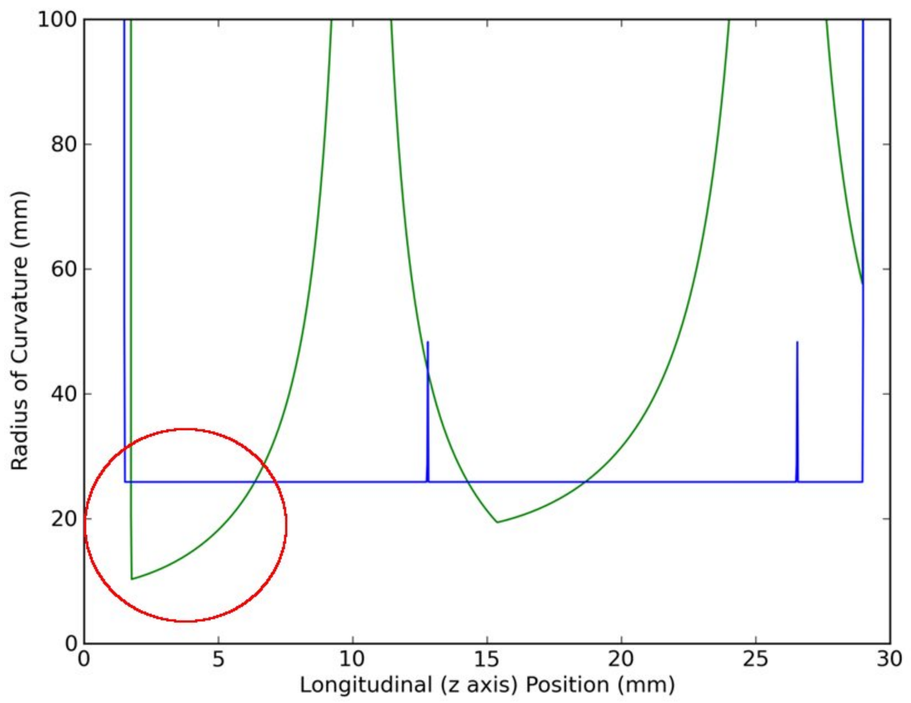


Figure 57 - Comparison of radius of curvature for spline (green) and arc (blue) waveguides with low radius of curvature region circled in red

The physical mechanism for the excess bend loss observed at the lower radius of curvature values was determined to be due to mechanical difficulties with the laser fabrication translation stages when moving at the high velocities. As the radius of curvature decreases, the translation stage also accelerates around the curve faster, thereby causing a higher susceptibility for fabrication defects. The data from this measured radius parameter scan curve was used to implement a lookup table in the Python application to give a calculated estimate for the bend loss. The physically calculated values for bend loss per mm in Table 5 were then implemented into the calculation algorithm as a lookup table. Now, the calculation algorithms finally had a robust, experimentally derived bend loss estimate for the HPO waveguides. This lookup table has been used to predict throughputs in the next section.

9.2 Predicted Power and Radius of Curvature Throughputs

Based on the test results in the previous section, the arc design was used to fabricate the latest sidestep remapper chip. An image of this arc sidestep is shown in Figure 58.

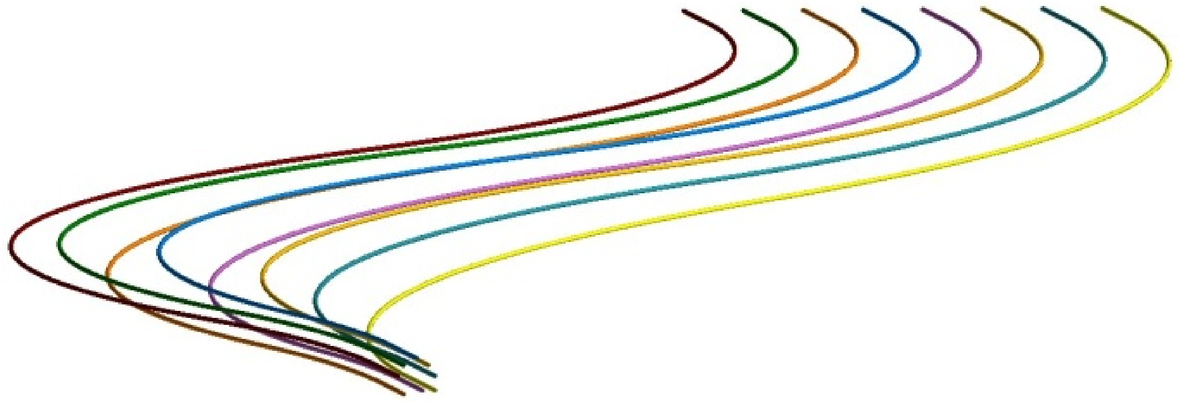


Figure 58 - 8 waveguide sidestep arc based design

In the process of designing these waveguides, the new HPO bend loss lookup table was implemented and a full power estimate including bend, transition, absorption, and coupling losses was run on all eight waveguides. In addition, these waveguides were simulated in RSoft, the power throughputs determined and absorption and coupling loss added to those values. The physical power throughputs for each waveguide were also measured using the optical probe setup from Section 9.1. These three power values were plotted on together and the values of these are shown in Figure 59 with the power scale on the left hand side. The radius of curvature of each arc based waveguide has also been plotted on the same graph using the scale on the right hand side.

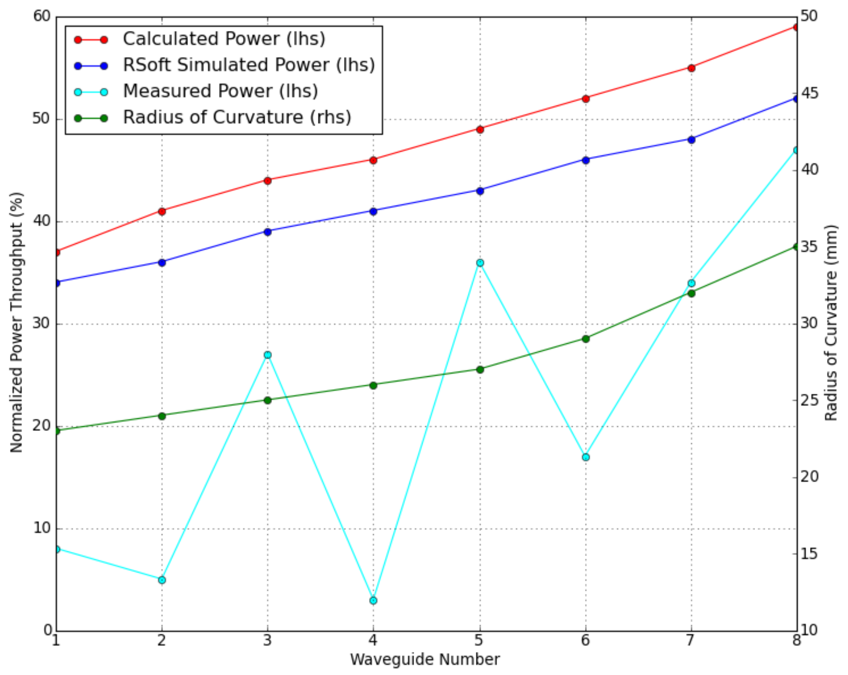


Figure 59 – Graph of calculated (red), simulated (blue) and measured (aqua) power throughput results with overlaid radius of curvature (green) values

The graph in Figure 59 shows that the calculated power throughputs of these waveguides should range between 36 – 58%. These predictions match the RSoft values with an error range between 5 and 10%. In addition, the radius of curvature values show all waveguides were fabricated with radius values above 23 mm, which was found in the last section to be the minimal radius to be safely used. Despite all of these separate predictions agreeing that waveguides should have good power throughputs, the measured throughputs are worse and in some cases dramatically so, ranging from 5 – 47%. This implies that there are still unknown fabrication imperfections in the HPO fabrication process that cause lower throughputs in the waveguides for complex, three-dimensional structures with low radius of curvature values. Note that the pattern of low throughputs does not show strong correspondence with tighter radius of curvature. Furthermore, the simple planar arc chip discussed in section 9.1 did exhibit throughputs which were monotonic with radius and conformed well with expectations, implying that the difficulties may stem from the challenges of writing curves into the y dimension. The throughputs for some of these waveguides will make them unsuitable for stellar measurements, however lab measurements of the coherence lengths can still be made, which are discussed in the next section.

9.3 Physical Waveguide Path Length Measurements

To measure interferometric fringes in the laboratory, the setup in Figure 20 was used with the addition of a near-infrared transmission grating between the output lenslet array and the camera. This cross disperses the output light from all eight waveguides so that the interferometric fringes can be imaged directly on the camera. “Turning off” six of the eight waveguides, by mispointing the micromirror for each of those waveguides off-axis, gives a pair of interference fringes from two waveguides. Selecting different combinations of waveguide pairs will give different spacing of the fringes. Figure 60 shows interference fringes between waveguide pairs with output spacing of 750 µm (left) and 1750 µm (right). The fringes run horizontally, following the orientation of the output waveguides.. The two sets of fringes depicted have different spatial frequencies as they arise from output apertures separated by different distances, with the image on the right having a longer baseline and hence higher spatial frequency. The fringes are cross dispersed by a transmission grating, with wavelength running in the vertical from shorter wavelengths (top) to longer wavelengths (bottom) over the range 1500 – 1600 nm. The effect of the changing wavelength can be seen in the fringe spatial frequency which becomes bigger, causing the pattern to appear to expand towards the bottom. If the wavefronts were in phase across all output apertures, the fringes would run exactly vertically. However the two images in Figure 60 are seen to exhibit inclined due to the manifestation of a *phase delay* between the signals from the two contributing waveguides.

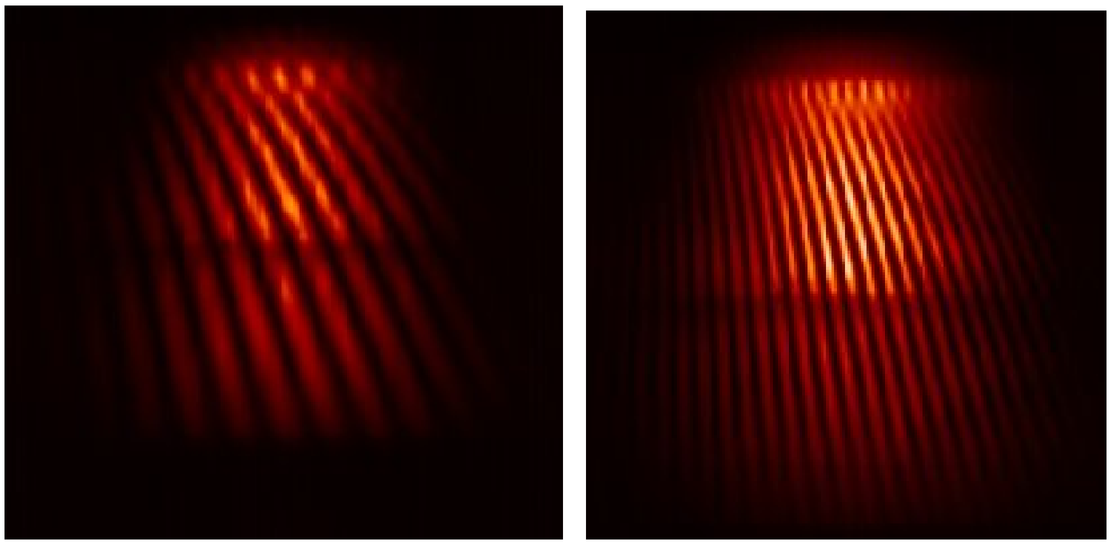


Figure 60 – Two images of wavelength cross-dispersed interference fringes between waveguide pairs. Image on left shows interference fringes between two waveguides spaced 750 µm apart, image on right shows interference fringes between two waveguides spaced 1750 µm apart. Wavelength runs in the vertical direction while spatial frequency runs horizontally.

Using fringe data like that in Figure 60, it is possible to measure the phase delay between the wavefronts coming from the two separate waveguides. Using the known wavelength scale and geometry of the optics, it is then a trivial matter to convert this phase delay (in radians) into a physical delay length that one optical path has with respect to the other. This value of the optical path difference can be recorded for all pair combinations of waveguides in a particular device.

Measuring the phase slopes was essentially achieved with a channel spectrum analysis. Correct sampling of the Fourier components corresponding to the fringes was ensured by plotting the power (intensity) spectrum of the fringes between a particular waveguide pair as a function of wavelength. This is depicted in Figure 61 (where the data arise from the fringe pattern seen in the left image in Figure 60. This graph is overlaid by a sampling template (white line) giving the expected locus of fringe power given the known wavelength scale and interferometer baseline.

Fringe interferograms were then taken while adjusting the path length difference by pistoning the corresponding segment of the MEMS array. This piston was moved from a nominal location of -1.5 µm to +1.5 µm in 0.1 µm steps (when used in reflection, this results in 6 microns of actual optical path difference). As described above, changing the optical path has the effect of tilting the slope of the fringe phase as a function of wavelength. If the plots for 31 separate piston locations are all overlaid for the same waveguide pair, the result is the graph shown in Figure 62.

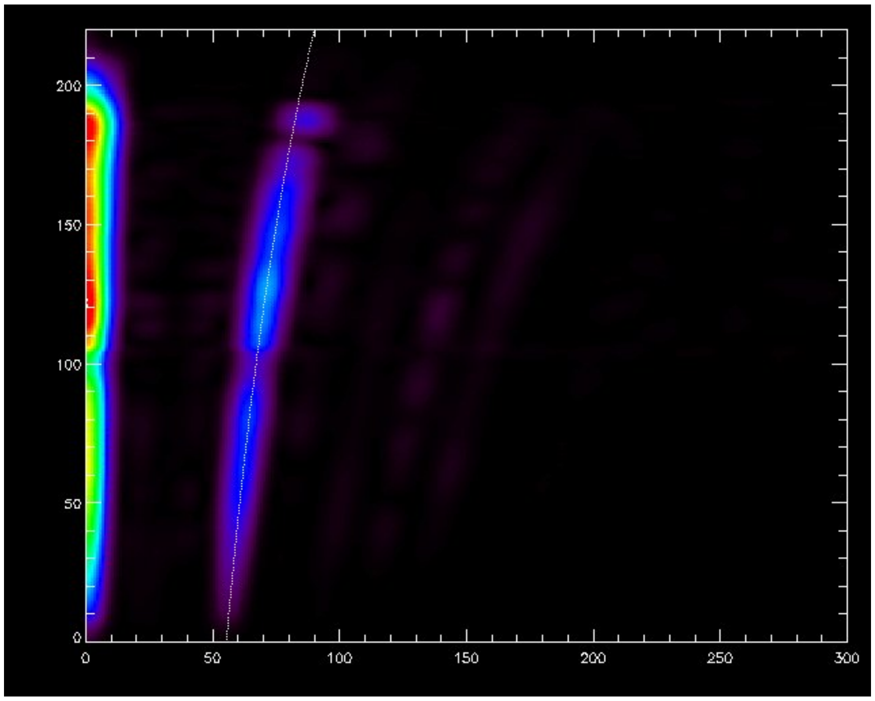


Figure 61 – Power spectrum for two waveguides 750 µm apart. Wavelength channel is on the vertical axis, while fringe spatial frequency in inverse pixels is on the horizontal axis. The line of predicted fringe power is overlaid on fringe spectrum (white overplotted line).

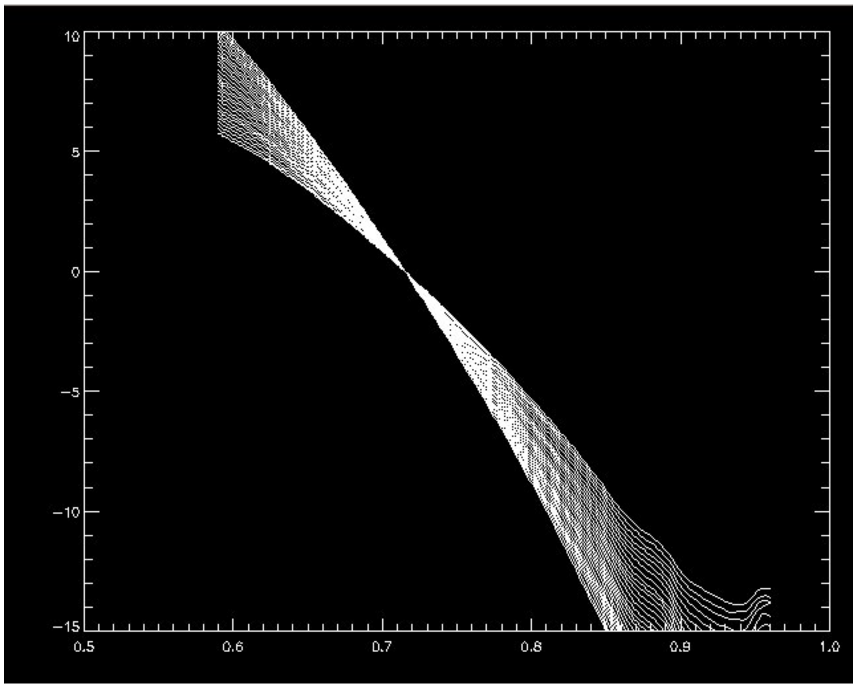


Figure 62 – Difference in fringe phase (radians; vertical axis) plotted as a function of inverse wavelength (1/microns, horizontal axis). The 31 different plots shown illustrate the change in fringe phase slope as one contributing waveguide is pistoned from -1.5 µm to 1.5 µm.

It can be seen that by changing the phase offset for this particular baseline, the slope of the fringe phase as a function of wavelength can be changed. However, over the entire range available from the MEMS array (6 microns), it can be seen that although a sizeable difference in slope can be obtained, the phase slope cannot be eliminated. This tells us immediately that the waveguides are not matched to within 6 microns, and indeed exhibit a mismatch at least several times worse than this value. By using the information in these plots, the precise phase difference between that particular pair of waveguides can be determined, and the process repeated for all waveguide pairs. Using the line of best fit will give the physical path difference between a waveguide pair.

Using one waveguide as a reference and comparing path length differences to the other seven waveguides will yield a differential path length measurement for each individual waveguide. A plot of the path length differences of the fabricated waveguide set given in Figure 58 is shown in Figure 63.

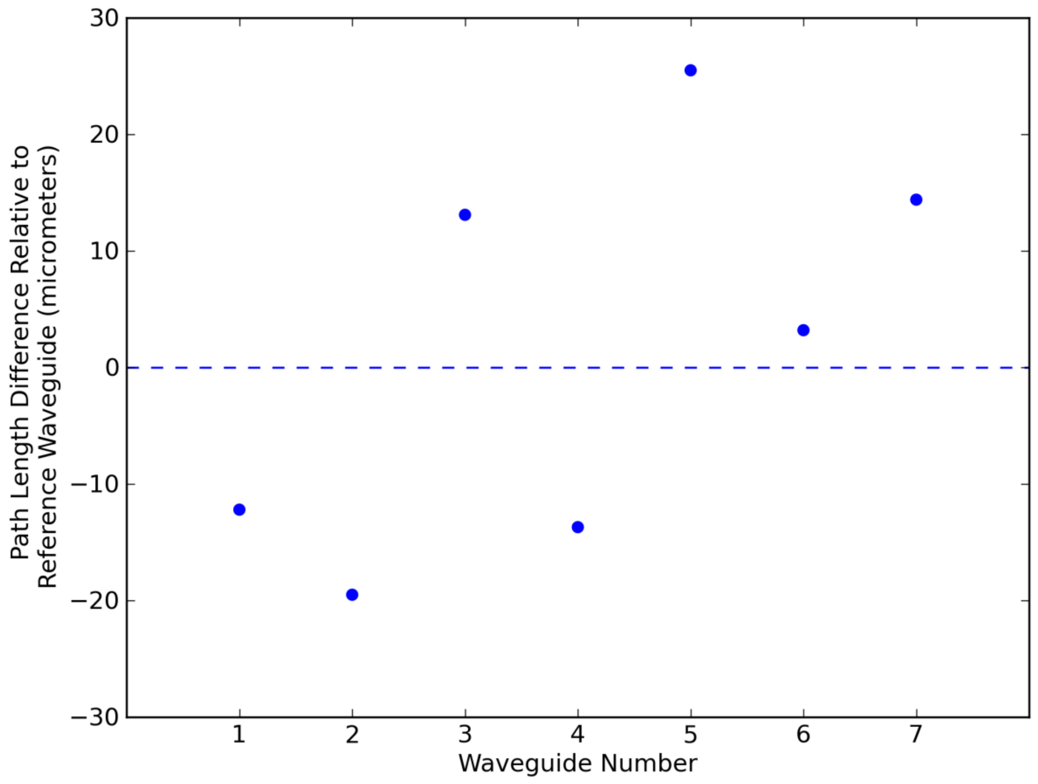


Figure 63 - Plot of path length differences of remaining waveguides in set compared to reference waveguide.

By comparing the path length differences in this graph, the relative path length differences between other waveguides can be determined. Thus, the longest path length difference in this set of eight waveguides is between waveguide 2 (-19.5 µm) to waveguide 5 (25.5 µm) for a total path length difference of 45 µm. This means that this set of waveguides could be used for scientific measurements in optical setups in which the tolerance on path length matching to a coherence length was greater than that value.

Although the performance reported here is within our specification range for various optical setups, there is a significant degradation in the performance between the mathematical trajectories (which are path length matched much better than one micron) and the measured performance of the fabricated component. This could be due to several factors, such as variations in the refractive index between different guides (written with different depths, curvatures or times), non-ideal performance of the stages causing errors in the guides, or possibly other sources of error due to misalignments in the optical apparatus making the measurements.

10. Conclusions

10.1 Summary

This thesis has covered the process of design and fabrication of path length matched waveguides in a single block of glass. In the process of investigating the methods to meet the design challenges, a full analysis of mechanisms of power loss in optical waveguides was performed. This analysis showed that using a computer implementation of theoretical equations for curvature, bend loss and transition loss gave an estimate of the power throughput of a given optical waveguide. This was confirmed by simulated measurements using Beam Propagation Method (BPM) software. Furthermore, our custom computer implementation allowed these measurements to be performed in a fraction of the time it would take to import the waveguides into the BPM software and have the power measurement calculated.

As part of developing the methods to design the waveguides, waveguides design on two different geometrical principles were explored. The first method used cubic splines to route the guides, providing a straight forward method of design. However, if the calculated power losses were unacceptable, a second method based on joined segments of circular arc could be used to design the waveguide. This circular arc design allowed for waveguide design with a higher minimum radius of curvature: a key feature as losses can be catastrophic for even short sections of high curvature. Either of these methods could then be used to lay out a waveguide chip, and the power throughputs calculated. Despite these advances, power throughputs of waveguides which have curvature in three dimensions created with the HPO process cannot be well predicted nor guaranteed to meet some user-defined specification. Further investigation into the reasons for the degraded performance compared to expectations is still needed.

Laboratory analysis performed on the first set of designed waveguides showed interferometric fringes obtained using these waveguides. To the best of our knowledge these were the first arbitrarily routed, path length matched waveguides, fabricated for interferometric purposes. A more detailed analysis was performed on the waveguides themselves, and the path length of all waveguides were found to be within 45 µm of each other.

Despite the power throughput problems, a set of waveguides created by the design process in this thesis was taken to the Anglo Australian Telescope in May of 2011. The waveguide chip was installed within the optical setup of the Dragonfly instrument on the telescope and the first stellar interference fringes were obtained by the system [40]. Visibilities and closure phases were recorded across the astronomical H band (and also part of the J band) for all baselines present in the input pupil and for all sets of closing triangles.

10.2 Future Work

In order to build further upon the designs explored here, a few items of possible improvements have been added.

Two design methods were created here, the cubic spline and circular arc, but no side by side direct experimental comparison was performed on the data. A sidestep cubic spline design was fabricated for interferometric testing, but exhibited the same erratic power measurements. Likewise, the arc based design method came later in the process, so no straight through designs were created on that. When the fabrication process is improved to get more consistent performance, a direct experimental comparison should be made to give more complete results.

The key advantage of this direct write pupil remapper technology is the ability to use more of the pupil area than aperture masking. This allows for more light from a stellar target to be measured. As the MEMS array used for this project has 37 segments, it is conceivable that a 37 waveguide remapper could one day be created. The next likely prototypes to be fabricated will have 12 – 16 waveguides fabricated.

If more waveguides are to be added to the design, the process of keeping the required proximity between the waveguides will increasingly become more complex. This will require more sophisticated coding algorithms to determine the position of the waveguides in space and shaping them to fit while keeping the radius of curvature high.

Because of the difficulty with the current HPO laser fabrication technique in fabricating waveguides with the current design, alternative direct write lasers will be investigated to fabricate the waveguides. Some of these rely on very different physical processes to enact a change in the refractive index of the glass, and so the final waveguide produced may not suffer from the same issues which have hampered progress using the HPO fabrication. This will also hopefully improve the path length matching distance of all waveguides.

To expand the capabilities of the direct write pupil remappers, expansion into longer wavelengths and the mid-infrared is a big science goal. This will require alternate types of glass or substrate transparent in these regions, which will in turn demand and different laser parameters for successful fabrication.

Bibliography

1. J. Bland-Hawthorn, P. Kern, “Astrophotonics: a new era for astronomical instruments”, OPTICS EXPRESS 1880 (2009).

2. G. Perrin et al, “Interferometric coupling of the Keck telescopes with single-mode fibers,” Science 311, 194 (2006).

3. J. Bland-Hawthorn, M. A. Englund and G. Edvell, “New approach to atmospheric OH suppression using an aperiodic FBG,” Opt. Express 12, 5902-5909 (2004).

4. R. R. Thomson et al, “Ultrafast laser inscription of an integrated photonic lantern”, OPTICS EXPRESS 5698 (2011).

5. J. Bland-Hawthorn and A. J. Horton, “Instruments without optics: an integrated photonic spectrograph,” SPIE 6269, 21-34 (2006).

6. B. Martin, “Optics for the Giant Magellan Telescope”, OPN July/August (2009)

7. J.W. Hardy, *Adaptive Optics for Astronomical Telescopes*, Oxford University Press (1998)

8. P.G. Tuthill, S. Lacour, P. Amico, M. Ireland, B. Norris, P. Stewart, T. Evans, A. Kraus, C. Lidman, E. Pompei, and N. Kornweibel, “Sparse Aperture Masking (SAM) at NAOS/CONICA on the VLT”, Proc. SPIE 7735, 77351O (2010).

9. Monnier, J. D., Tuthill, P. G., Lopez, B., Cruzalebes, P., Danchi, W. C., Haniff, C. A., "The Last Gasps of VY Canis Majoris: Aperture Synthesis and Adaptive Optics Imagery", ApJ, vol. 512, p. 351-361 (1999)

10. T. J. Cornwell, “The Applications of Closure Phase to Astronomical Imaging”, Science Magazine, Vol. 245 no. 4915 pp. 263-269 (1989)

11. M.J. Ireland et al, “Dynamical Mass of GJ 802B: A Brown Dwarf in a Triple System”, Astrophysical Journal, 678:463Y471 (2008)

12. T. Bedding, "The Orbit of the Binary Star Delta Scorpii", Astron.J., 106, pp. 768-772 (1993)

13. H. C. Woodruff, P. G. Tuthill, J. D. Monnier, M. J. Ireland, T. R. Bedding, S. Lacour, W. C. Danchi, M. Scholz, "The Keck Aperture Masking Experiment: Multi-Wavelength Observations of 6 Mira Variables", Astrophys.J, 673:418Y433 (2008)

14. P. G. Tuthill, J. D. Monnier, and W. C. Danchi, "A dusty pinwheel nebula around the massive star WR104", Nature 398, pp. 487-489 (1999)

15. V. Coudé du Foresto, “Fringe benefits: the spatial filtering advantages of single-mode fibers”, Integrated Optics for Astronomical Interferometry (1997)

16. S. Nolteu et al, “Femtosecond waveguide writing: a new avenue to three-dimensional integrated optics”, Appl. Phys. A 77, 109–111 (2003)

17. A. Ródenas et al, “Three-dimensional mid-infrared photonic circuits in chalcogenide glass”, Optics Letters / Vol. 37, No. 3 (2012)

18. P.G. Tuthill et al, “Photonic technologies for a pupil remapping interferometer”, Proc. of SPIE-Astronomical Instrumentation, paper 7734-59, (2010).

19. Saleh and Teich, *Fundamentals of Photonics 2nd Edition*, Wiley (2007)

20. J.D. Monnier, *“*Optical Interferometry in Astronomy”, Reports on Progress in Physics (2003)

21. J.D. Monnier, “Phases in Interferometry”, New Astronomy Reviews Vol. 5 (2007)

22. E. Huby, G. Perrin, F. Marchis, S. Lacour, T. Kotani, G. Duchêne, E. Choquet, E. L. Gates, J. M. Woillez, O. Lai, P. Fédou, C. Collin, F. Chapron, V. Arslanyan, K. J. Burns, "FIRST, a fibered aperture masking instrument. I. First on-sky test results", A&A, **541**, A55 (2012)

23. K.M. Davis et al, “Writing waveguides in glass with a femtosecond laser”, Optics Letters Vol. 21, No. 21 (1996)

24. C. B. Schaffer et al, “Laser-induced breakdown and damage in bulk transparent materials induced by tightly focused femtosecond laser pulses”, Meas. Sci. Technol., **12**, pp. 1784–1794 (2001)

25. S. Nolte et al, “Femtosecond waveguide writing: a new avenue to three-dimensional integrated optics,” Appl. Phys. A – Mater. **77**, 109-111 (2003)

26. R.R. Thomson, “Ultrafast-laser inscription of a three dimensional fan-out device for multicore fiber coupling applications”, Optics Express, Vol. 15, Issue 18, pp. 11691-11697 (2007)

27. A. Ródenas, G. Martin, B. Arezki, N. Psaila, G. Jose, A. Jha, L. Labadie, P. Kern, A. Kar, and R. Thomson, "Three-dimensional mid-infrared photonic circuits in chalcogenide glass," Opt. Lett. 37, 392-394 (2012)

28. J. P. Berger, Haguenauer, P. Kern, K. Perraut, F. Malbet, I. Schanen, M. Severi, R. Millan-Gabet and W. Traub, "Integrated optics for astronomical interferometry IV: First measurements of stars," A & A 376, L31 (2001)

29. H. Gu and J. Xu. “Design of 3D Optical Network on Chip”, Proceedings of International Symposium on Photonics and Optoelectronics (SOPO) (2009)

30. K. Okamoto, *Fundamentals of Optical Waveguides* (2006)

31. A.W. Snyder and J.D. Love, *Optical Waveguide Theory*, Chapman and Hall (1983)

32. A. Labeyrie, S.G. Lipson, P. Nisenson, *An Introduction to Optical Stellar Interferometry*, Cambridge University Press (2006)

33. S.M. Eaton et al, “Transition from thermal diffusion to heat accumulation in high repetition rate femtosecond laser writing of buried optical waveguides” Optics Express 9443 (2008)

34. S. Lacour et al, “Characterization of integrated optics components for the second generation of VLTI instruments”, Proc. of SPIE Vol. 7013, 701316, (2008)

35. J.H. Mathews and K.K. Fink, *Numerical Methods Using MATLAB 4th Edition*, Prentice Hall (2003)

36. J. Kiusalaas, *Numerical Methods in Engineering with Python,* Cambridge University Press (2005)

37. Ramachandran and Varoquaux, “Mayavi: 3D Visualization of Scientific Data”¸ IEEE Computing in Science & Engineering (2011)

38. B. Norris et al, “Challenges in Photonic Pupil Remapping for Optical Stellar Interferometry”, *IQEC/CLEO Pac-Rim Proceedings* (2011)

39. N. Jovanovic et al, "Direct Laser Written Multimode Waveguides for Astronomical Applications," in *Bragg Gratings, Photosensitivity, and Poling in Glass Waveguides*, OSA Technical Digest, paper JThA28 (2010)

40. N. Jovanovic et al, "First stellar photons through an integrated photonic pupil remapping interferometer", *Proceedings of the International Quantum Electronics Conference and Conference on Lasers and Electro-Optics Pacific Rim 2011*,paper C1212 (2011)

Appendix – Python Application code

This appendix contains the Python code that implements the functionality to create the waveguides, as well as performing analysis on them. The first portion contains the various variables and class objects needed for the various functions. The main logic portion calls out the various functions to perform specific tasks.

- CalculateInputOutputDistance function calculates the linear distance between x and y end points. This data is used by the DeterminePrimaryWaveguides function to determine the primary waveguide in each set. DetermineSecondaryWaveguides populates a list with the rest of the waveguides
- If fabrication of arcs is desired, then the lines under ARC BASED CODE should be uncommented with the spline code commented.
    - CreatePrimaryArcWaveguide creates the primary waveguide. It calls the function CreateDoubleArcSegment to perform the construction.
    - CreateSecondaryArcWaveguides contains the code to fabricate the secondary arc based waveguides. The positioning code and code to keep track of radius reduction is here. In this code, it calls the function CreateTripleArcSegment to fabricate a waveguide from three circular arcs
- If fabrication of splines is desired, then the lines under SPLINE BASED CODE should be uncommented with the arc code commented.
    - CreatePrimarySplineWaveguide creates the primary waveguide. It calls the function CreateSingleSpline to perform the construction.
    - CreateSecondaryArcWaveguides contains the code to fabricate the secondary splines based waveguides. The positioning code and code to keep track of radius reduction is here. In this code, it calls the function CreateDoubleSpline to fabricate a waveguide with an offset middle knot.
- With the waveguides successfully designed and path length matched, the function SegmentDistanceCurvatureCheck is called. This calculates the curvature, the closest distance between any two waveguides and the depth at which the waveguides are created.
    - This function also calls CalculatePowerLoss. This function goes through a specific waveguide and calculates the bend, transition, coupling and absorption power losses for each waveguide to give an estimated power throughput. Choosing a step index profile, Gaussian index profile, or an HPO profile from a lookup table is set here.
- PlotAllSegments creates the three-dimensional plot using the Mayavi library
- CreateLaserWriteFiles takes the three-dimensional coordinates of each waveguide and writes them to a file in a format to be read by the laser fabrication computer.

```
# -*- coding: utf-8 -*-
# Python file to design best path(s) through an optical chip based on input and output positions
#
# Copyright Ned Charles 2011
# The references here are all used for education purposes only
#

############################################################################################################
# ** CLASS DECLARATION **
############################################################################################################
#
# PositionDimensionDataObject
#
# Contains all start and end positions, physical requirements and dimensions,

class PositionDimensionDataObject(object):

    def __init__(self, NumberOfSegments):
        self.NumberOfSegments = NumberOfSegments
        self.PathXStartPosArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.PathYStartPosArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.PathXEndPosArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.PathYEndPosArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.XDistanceArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.YDistanceArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.DirectDistanceArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.SecondaryIndexList = np.zeros(1, dtype=np.int)
        self.SegmentCreatedArray = np.zeros(NumberOfSegments, dtype=np.float64)
        self.ArcCutoffPoint = np.zeros(NumberOfSegments, dtype=np.float64)
        self.RadiusOffsetArray = np.zeros(NumberOfSegments-1, dtype=np.float64)
        self.LeadInSegmentArray = np.zeros(NumberOfSegments-1, dtype=np.int)

    PrimarySegmentNumber = 0
    PrimarySegmentRadius = 0.0
    BlockSectionWidth = 0
    BlockSectionHeight = 0
    BlockSectionLength = 0
    EdgeStraightSectionLength = 0.0
    CenterBridgeLength = 0.0
    BlockSectionLength = 0.0
    ZSpatialPrecision = 0.0
    CalculatedPathDistance = 0.0
    SegmentLengthPrecision = 0.0
    NumberZSlices = 0

############################################################################################################
#
# WaveguidePropertiesDataObject
#
# Contains all information about the physical properties of the optical waveguide.  This information
# will be used to estimate light loss in the waveguide.  The parameters are based on Snyder and Love
# textbook and the calculations to determine these parameters are performed later on.

class WaveguidePropertiesDataObject(object):

##    def __init__(self, NumberOfSegments):
##        self.NumberOfSegments = NumberOfSegments

    # These parameters are set once and good for the physical properties of all waveguides
    IndexCore = 0.0
    IndexCladding = 0.0
    WaveguideRadius = 0.0        # The rho parameter
    Wavelength = 0.0            # microns
    NumericalAperture = 0.0
    NormalizedFrequency = 0.0   # The V parameter
    DeltaParameter = 0.0
    UParameter = 0.0
    WParameter = 0.0
    RadiusZero = 0.0            # The spot size - calculate on pg 341 using relationships there
    BetaCoefficient = 0.0


############################################################################################################
# ** AUXILIARY FUNCTIONS **
############################################################################################################
# Function - CalculateWaveguideProperties
#
# The parameters in the WaveguidePropertiesDataObject are calculated here.  The calculations are dervied
# from the textbook Optical Waveguide Theory by Snyder and Love, 1983.

def CalculateWaveguideProperties(WPData):

    WavePropData.IndexCore = 1.4927
    WavePropData.IndexCladding = 1.4877
    WavePropData.WaveguideRadius = 4.85
    WavePropData.Wavelength = 1.55  # microns

    # Numerical Aperture is based on the core and cladding indices
    WPData.NumericalAperture = np.sqrt(WPData.IndexCore**2 - WPData.IndexCladding**2)

    # Parameters from back of Snyder and Love
    WPData.NormalizedFrequency = ((2.*np.pi)/WPData.Wavelength) * WPData.WaveguideRadius * \
                                 WPData.NumericalAperture
    WPData.DeltaParameter = (WPData.IndexCore**2 - WPData.IndexCladding**2)/(2.*WPData.IndexCore**2)
    # Beta Coefficient is an arbitrary parameter based on the waveguide characteristics
##    BetaCoefficientArray = [6.041393049,6.03812908,6.035121234] from RSoft -> delta n = 0.005,0.004,0.003
    WPData.BetaCoefficient = 6.041393049

    # Given beta value, calculate W and U parameters
    kParam = (2.*np.pi)/WPData.Wavelength
    WPData.WParameter = WPData.WaveguideRadius * np.sqrt(WPData.BetaCoefficient**2 - \
                                                    (WPData.IndexCladding**2 * kParam**2))
    WPData.UParameter = WPData.WaveguideRadius * np.sqrt((WPData.IndexCore**2 * kParam**2) - \
                                                    WPData.BetaCoefficient**2)

    # Step Profile
    WPData.RadiusZero = WPData.WaveguideRadius/(np.sqrt(2. * np.log(WPData.NormalizedFrequency)))
    # Gaussian Profile
    WPData.RadiusZero = WPData.WaveguideRadius/np.sqrt(WPData.NormalizedFrequency - 1)

    print "******************************************************************"
    print "WAVEGUIDE PROPERTIES"
    print "IndexCore: " + str(WPData.IndexCore)
    print "IndexCladding: " + str(WPData.IndexCladding)
    print "WaveguideRadius: " + str(WPData.WaveguideRadius)
    print "Wavelength: " + str(WPData.Wavelength)
    print "NumericalAperture: " + str(WPData.NumericalAperture)
    print "NormalizedFrequency: " + str(WPData.NormalizedFrequency)
    print "DeltaParameter: " + str(WPData.DeltaParameter)
    print "UParameter: " + str(WPData.UParameter)
    print "WParameter: " + str(WPData.WParameter)
    print "RadiusZero: " + str(WPData.RadiusZero)
    print "******************************************************************"


############################################################################################################
# DESIGN FOR DETERMINING DISTANCES
############################################################################################################
# Function - CalculateInputOutputDistance
# This code calculates the differences in x and y between the input and output points of each spline
def CalculateInputOutputDistance(PDData):
    for i in range(0, len(PDData.XDistanceArray)):
        PDData.XDistanceArray[i] = PDData.PathXEndPosArray[i] - PDData.PathXStartPosArray[i]
        PDData.YDistanceArray[i] = PDData.PathYEndPosArray[i] - PDData.PathYStartPosArray[i]
        #Direct distance formed by taking the hypotenuse of the X and Y difference above
        PDData.DirectDistanceArray[i] = np.sqrt((PDData.XDistanceArray[i])**2 + (PDData.YDistanceArray[i])**2)
    return

############################################################################################################
# Function - DeterminePrimarySegments
# Find which spline has the largest distance.  Returns the index of the largest segment

def DeterminePrimarySegments(DirectDistanceArray):
    LargestDistanceIndex = np.argmax(abs(DirectDistanceArray))
    print "Largest ordinal segment number: " + str(LargestDistanceIndex)
    return LargestDistanceIndex

############################################################################################################
# Function - DetermineSecondarySegments
# Find a list with the secondary segments

def DetermineSecondarySegments(PDData):
    PDData.SecondaryIndexList[0] = -1

    for i in range(0, PDData.NumberOfSegments):
        # If this segment doesn't exist in the largest list, add it to the secondary list
        if(i != PDData.PrimarySegmentNumber):
            #If the first item is -1 (first element in the list), replace, otherwise add
            if(PDData.SecondaryIndexList[0] == -1):
                PDData.SecondaryIndexList[0] = i
            else:
                PDData.SecondaryIndexList = np.append(PDData.SecondaryIndexList, i)

############################################################################################################
# AUX FUNCTIONS FOR BELOW
# Functions - For tridiagonal matrix deconvolving
# Adapted from Numerical Engineering Methods in Python (2005), Section 2.4

def LUdecomp3(c,d,e):
    n = len(d)
    for k in range(1,n):
        lam = c[k-1]/d[k-1]
        d[k] = d[k] - lam*e[k-1]
        c[k-1] = lam
    return c,d,e

def LUsolve3(c,d,e,b):
    n = len(d)
    for k in range(1,n):
        b[k] = b[k] - c[k-1]*b[k-1]
    b[n-1] = b[n-1]/d[n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - e[k]*b[k+1])/d[k]
    return b

############################################################################################################
# Function - CalculateSegmentLength
# Calculates the segment length based on the data points
```

```
def CalculateSegmentLength(PDData, XYZSegmentDataArray, SegNum):
    # Calculate spline length
    totalPathDistance = 0.0
    previousSliceXValue = PDData.PathXStartPosArray[SegNum]
    previousSliceYValue = PDData.PathYStartPosArray[SegNum]
    previousSliceZValue = 0.0

    for i in range(1, len(XYZSegmentDataArray[SegNum,:,0])):
        #Add the path amount to the total distance
        totalSliceDistance = np.sqrt((XYZSegmentDataArray[SegNum,i,0]-previousSliceXValue)**2 + \
                                     (XYZSegmentDataArray[SegNum,i,1]-previousSliceYValue)**2 + \
                                     (XYZSegmentDataArray[SegNum,i,2]-previousSliceZValue)**2)
        totalPathDistance += totalSliceDistance
        previousSliceXValue = XYZSegmentDataArray[SegNum,i,0]
        previousSliceYValue = XYZSegmentDataArray[SegNum,i,1]
        previousSliceZValue = XYZSegmentDataArray[SegNum,i,2]
    #end slice for loop

    return totalPathDistance

###############################################################################################################
# DESIGN FOR CREATING SPLINE FOR EACH PATH
###############################################################################################################
# Function - CreateSpline
# To calculate the path, the program uses the spline function.  This takes the starting and end
# points of the path, then returns the centerpoints of the function for each z-slice.
# http://www.physics.utah.edu/~detar/phys6720/handouts/cubic_spline/cubic_spline/node1.html
#
# This spline creation program uses the XY Distance plus the Z Distance to create the spline.  The
# XY difference will be the difference in YLineValues below on the line to pass in that will create
# it.  The Z difference will be used to create the XLineValues to pass in below.  The slopes of the
# line created will then be passed in to the code below.  This will return the 2nd derivative
# k values to pass into the Evaluate Spline function, which will return the distance values at each
# slice.  Using the X and Y beginning and end values of the segment will allow the spline to be
# translated into an array of X, Y values.

def CreateSpline(XYDistance, ZDistance, Invert = 0, DoubleSpline = 0, DoubleSplineOffset = 0.0):
    XLineValues = np.array([0.0,ZDistance/2.0,ZDistance]) # minimum of three points
    LineSlope = XYDistance/ZDistance
    YLineValues = LineSlope*XLineValues

    if (DoubleSpline == 1):
        YLineValues[1] += DoubleSplineOffset

    if (DoubleSpline == 2):
        XLineValues[1] = ZDistance/3.
        TwoThirdDataPointXVal = (ZDistance*2.)/3.
        YLineValues[1] = LineSlope * XLineValues[1]
        TwoThirdDataPointYVal = LineSlope * TwoThirdDataPointXVal

        YLineValues[1] += DoubleSplineOffset
        TwoThirdDataPointYVal -= DoubleSplineOffset

        XLineValues = np.insert(XLineValues, 2, TwoThirdDataPointXVal)
        YLineValues = np.insert(YLineValues, 2, TwoThirdDataPointYVal)

    if (DoubleSpline == 3):
        # For secondary spline, the line passed in needs to be adjusted by picking two more points at 1/4
        # and 3/4 of the total length and then incrementing these values vertically in the opposite
        # direction to create a 3 segment model.  This will increase the total value of the spline.
        # Increment the data points at 1/4 and 3/4 of the lengths to create the 3 segments
        OneQuarterDataPointXVal = ZDistance*.25
        ThreeQuarterDataPointXVal = ZDistance*.75
        OneQuarterDataPointYVal = LineSlope * OneQuarterDataPointXVal
        ThreeQuarterDataPointYVal = LineSlope * ThreeQuarterDataPointXVal

        # Increase these values by the offset
        OneQuarterDataPointYVal += DoubleSplineOffset
        ThreeQuarterDataPointYVal -= DoubleSplineOffset

        # Add these into the X and Y line value arrays
        XLineValues = np.insert(XLineValues, 1, OneQuarterDataPointXVal)
        XLineValues = np.insert(XLineValues, 3, ThreeQuarterDataPointXVal)
        YLineValues = np.insert(YLineValues, 1, OneQuarterDataPointYVal)
        YLineValues = np.insert(YLineValues, 3, ThreeQuarterDataPointYVal)

    x = XLineValues
    y = YLineValues

    # Slope specified at the left and right sides of the splines
    LeftTangent = 0.
    RightTangent = 0.

    # Mathematics from Numerical Methods using MATLAB 4th Edition, Section 5.3
    # with adapted code from Numerical Engineering Methods in Python (2005), Section 3.3
    # This code calculates the second derivative (k) at each knot specified
    n = len(x) - 1

    # The values for the clamped spline can be solved using equation 12 of the MATLAB
    # book.  This equation is only valid for the values from 1...n-1, with modifications for
    # 1 & n-1 made below.  To solve the equation, the values of k for 0 and n must be substituted

    # h is an intermediate variable that calculates the distance between x values
    h = np.zeros((n), dtype=np.float64)
    h[0:n] = x[1:n+1] - x[0:n]
    a = np.zeros((n), dtype=np.float64)
    b = np.ones((n+1), dtype=np.float64)
    c = np.zeros((n), dtype=np.float64)
    u = np.zeros((n+1), dtype=np.float64)
    k = np.zeros((n+1), dtype=np.float64)

    # Calculate matrix parameters: a + b + c = u
    a[0:n-1] = h[0:n-1]
    b[1:n] = 2.0*(h[0:n-1] + h[1:n])
    c[1:n] = h[1:n]
    u[1:n] = 6.0*(((y[2:n+1]-y[1:n])/h[1:n])-((y[1:n]-y[0:n-1])/h[0:n-1]))

    # The loop is only valid for parameters from 1 to n-1.  For the first value, the value for k[0]
    # is unknown, so the eqn. after the loop to calculate k[0] is substituted into the main loop eqn.,
    # solving for k[0]
    b[1] = 1.5*h[0] + 2*h[1]
    u[1] = u[1] - 3*(((y[1]-y[0])/h[0])-LeftTangent)

    # For the value of c, it depends on how many points the spline has.  If it is only three, there
    # is no need to proceed to the matrix solving loop, as there is only one parameter to solve for

    if(n == 2):
        # This means that the value k[2] is also unknown and calculated later.  There is also only
        # one unknown, k[1], so that can be solved now instead of needing matrix algebra
        k[1] = (u[1] - 3*(RightTangent - ((y[2]-y[1])/h[1])))/(1.5*(h[0]+h[1]))
    else:
        # There is more than one equation with more than one variable to be solved.
        # There is a special case if the loop variable is equal to n-1.  This means that the
        # value for k[n] is not known at this time, so a substitution will need to be performed
        # to be able to solve the matrix.  Otherwise the values are as above

        # There is no value for c (zero), b and u are modified
        b[n-1] = 2*h[n-2] + 1.5*h[n-1]
        u[n-1] = u[n-1] - 3*(RightTangent - ((y[n]-y[n-1])/h[n-1]))

        a,b,c = LUdecomp3(a,b,c)
        k = LUsolve3(a,b,c,u)

    # Plug in after solving for all center values
    k[0] = (3./h[0])*(((y[1]-y[0])/h[0]) - LeftTangent) - k[1]/2.
    k[n] = (3./h[n-1])*(RightTangent - ((y[n]-y[n-1])/h[n-1])) - k[n-1]/2.

    return x,y,k

############################################################################################################
# Function - EvaluateSpline
# Adapted from Numerical Engineering Methods in Python (2005)

def EvaluateSpline(XYKDataArray,x):

    xData = XYKDataArray[0]
    yData = XYKDataArray[1]
    kData = XYKDataArray[2]

    def findSegment(xData,x):
        iLeft = 0
        iRight = len(xData)- 1
        while 1:
            if (iRight-iLeft) <= 1: return iLeft
            i =(iLeft + iRight)/2
            if x < xData[i]: iRight = i
            else: iLeft = i

    i = findSegment(xData,x)
    h = xData[i] - xData[i+1]
    y = ((x - xData[i+1])**3/h - (x - xData[i+1])*h)*kData[i]/6.0 \
      - ((x - xData[i])**3/h - (x - xData[i])*h)*kData[i+1]/6.0   \
      + (yData[i]*(x - xData[i+1])                              \
       - yData[i+1]*(x - xData[i]))/h

    return y

############################################################################################################
# SPLINES
############################################################################################################
# Function DesignPrimarySpline

def DesignSplineSection(DirectDistance, ZDistance, ZPrecision, XStartPos, YStartPos, XDistance, YDistance, \
                        XYZSegDataArray, DoubleSpline = 0, DoubleSplineOffset = 0.0, AngleOffset = 0.0):

    # The code will create two splines, one for the x value in space and the other for the y value.
    # The code can be passed in an angle offset that will specify a direction to stretch the segment in.

    # The spline was created using the DirectDistanceArray.  This array falls on a line that lies
    # in the x-y plane and was created earlier using the starting and ending x-y points of the
    # segment.  We can use the slope of the line to determine the x and y coordinates of a point
    # that lies on this line.  The tangent of the corresponding angle equals the line slope

    XYLineAngle = np.arctan2(YDistance,XDistance)
    XDoubleSplineOffset = 0.
    YDoubleSplineOffest = 0.

    # All lateral stretching of the algorithm happens in the plane equal to the slope of the line
    # of the x-y projection of the start to end points of the line in the block.  To stretch the knot(s)
    # at another angle in space, an angle offset is specified here.  The angle is specified with
    # an angle of zero/2 pi (radians) in the x positive direction, although an angle offset of zero
    # means no offset.  Positive angle rotates in a clockwise direction looking from the start end toward
    # the end to match with the right hand curl rule.  The middle knot point is taken from the projection
    # line and then moved from the middle point of the segment in the angle specified.
    if (AngleOffset != 0.0):
        XDoubleSplineOffset = np.cos(AngleOffset)*abs(DoubleSplineOffset)
        YDoubleSplineOffset = np.sin(AngleOffset)*abs(DoubleSplineOffset)

    else:
        # Project along the original angle
        # X value = cos(angle)*hypotenuse, Y value = sin(angle)*hypotenuse
        XDoubleSplineOffset = np.cos(XYLineAngle)*DoubleSplineOffset
        # Need to correct for negative angles because the sine value is negative
        YDoubleSplineOffset = np.sin(XYLineAngle)*DoubleSplineOffset

    # Call the create spline routine to get the k (2nd derivative) values based on the lateral distance
    # and the length.
    XKSplineArray = np.array(CreateSpline(XDistance, ZDistance, DoubleSpline = DoubleSpline, \
                                       DoubleSplineOffset = XDoubleSplineOffset))
    YKSplineArray = np.array(CreateSpline(YDistance, ZDistance, DoubleSpline = DoubleSpline, \
                                       DoubleSplineOffset = YDoubleSplineOffset))

    # The k values are obtained for the x and y splines.  These values will be passed
    # into the Evaluate Spline function.  Now, the spline will be passed in x values to give the
    # interpolated y values at each slice.

    # Set the index of the current slice to zero
    sliceIndex = 0

    # Use to calculate each section and sum of the curve
    totalPathDistance = 0.0
    previousSliceXValue = 0.0
    previousSliceYValue = 0.0
    previousSliceZValue = 0.0

    for ZInterpolatedValue in range(0, int(ZDistance+ZPrecision), int(ZPrecision)):
        XInterpolatedValue = EvaluateSpline(XKSplineArray, ZInterpolatedValue)
        YInterpolatedValue = EvaluateSpline(YKSplineArray, ZInterpolatedValue)
        XYZSegDataArray[sliceIndex,2] = ZInterpolatedValue # Z value

        # Add the Interpolated Value to the start position
        NewXPosition = XStartPos + XInterpolatedValue
        NewYPosition = YStartPos + YInterpolatedValue
        XYZSegDataArray[sliceIndex,0] = NewXPosition
        XYZSegDataArray[sliceIndex,1] = NewYPosition

        #Add the path amount to the total distance
        totalSliceDistance = np.sqrt((XInterpolatedValue-previousSliceXValue)**2 + \
                                 (YInterpolatedValue-previousSliceYValue)**2 + \
                                 (ZInterpolatedValue-previousSliceZValue)**2)
        totalPathDistance += totalSliceDistance
        previousSliceXValue = XInterpolatedValue
        previousSliceYValue = YInterpolatedValue
        previousSliceZValue = ZInterpolatedValue
        sliceIndex += 1
    #end slice for loop

    return totalPathDistance

################################################################################################################
# Function - CreateSingleSpline
# This function goes through and creates the primary spline(s)
def CreateSingleSpline(PDData, segmentIndex, XYZSplineDataArray):

    XStartPos = PDData.PathXStartPosArray[segmentIndex]
    YStartPos = PDData.PathYStartPosArray[segmentIndex]
    XEndPos = PDData.PathXEndPosArray[segmentIndex]
    YEndPos = PDData.PathYEndPosArray[segmentIndex]
    XDistance = PDData.XDistanceArray[segmentIndex]
    YDistance = PDData.YDistanceArray[segmentIndex]
    DirectDistance = PDData.DirectDistanceArray[segmentIndex]

    print "DirectDistance: " + str(DirectDistance)
    print "XDistance: " + str(XDistance)
    print "XEndPos: " + str(XEndPos)

    NetSplineSectionLength = PDData.BlockSectionLength - (PDData.EdgeStraightSectionLength*2.0)

    # Create an XYZ Array for this segment index to pass into the design spline function.
    # It will be returned with data.
    SplineDataArray = np.zeros(((NetSplineSectionLength/PDData.ZSpatialPrecision)+1, 3), dtype=np.float64)

    # No primary conflict.  Pass in the distances to the spline creation algorithm
    SplineDistance = DesignSplineSection(DirectDistance, NetSplineSectionLength, PDData.ZSpatialPrecision, \
                         XStartPos, YStartPos, XDistance, YDistance, SplineDataArray)
    print "SingleSplineDistance: " + str(SplineDistance)

    # Now configure the data array by extrapolating out the end straight sections
    # Add the left section. X and Y values equal to the starting position.  Fill in Z values.
    FirstSectionJoinIndex = int(PDData.EdgeStraightSectionLength/PDData.ZSpatialPrecision)
    XYZSplineDataArray[0:FirstSectionJoinIndex,0] = XStartPos
    XYZSplineDataArray[0:FirstSectionJoinIndex,1] = YStartPos
    ZIncrementArray = np.arange(0.0,PDData.EdgeStraightSectionLength, PDData.ZSpatialPrecision, dtype=np.float64)
    XYZSplineDataArray[0:FirstSectionJoinIndex,2] = ZIncrementArray
    # Copy the center section.  Increment the z-values to match up with beginning straight section
    SplineDataArray[:,2] += PDData.EdgeStraightSectionLength
    SecondSectionJoinIndex = int((PDData.EdgeStraightSectionLength + NetSplineSectionLength) \
                                    /PDData.ZSpatialPrecision)+1
    XYZSplineDataArray[FirstSectionJoinIndex:SecondSectionJoinIndex,:] = SplineDataArray

    #Add in the end section
    XYZSplineDataArray[SecondSectionJoinIndex:,2] = ZIncrementArray + (PDData.EdgeStraightSectionLength + \
                                                        NetSplineSectionLength + PDData.ZSpatialPrecision)
    XYZSplineDataArray[SecondSectionJoinIndex:,0] = XEndPos
    XYZSplineDataArray[SecondSectionJoinIndex:,1] = YEndPos

    # Return the total distance
    return SplineDistance

############################################################################################################
# Function - CreatePrimarySegments
# This function goes through and creates the primary spline(s)

def CreatePrimarySegments(PDData, XYZSegmentDataArray):

    NoPrimarySegmentError = 1
    # First calculate the primary ones

    SegmentIndex = PDData.PrimarySegmentNumber
    XYZSplineDataArray = np.zeros(((PDData.BlockSectionLength/PDData.ZSpatialPrecision)+1, 3), \
                                         dtype=np.float64)

    SplineDistance = CreateSingleSpline(PDData, SegmentIndex, XYZSplineDataArray)

    if (SplineDistance > 0):
        XYZSegmentDataArray[SegmentIndex,:,:] = XYZSplineDataArray
        PDData.SegmentCreatedArray[SegmentIndex] = 1

    #CalculatedPathDistance is an array
    PDData.CalculatedPathDistance = SplineDistance + (PDData.EdgeStraightSectionLength*2)
    print "Segment distance: " + str(PDData.CalculatedPathDistance)

    return NoPrimarySegmentError

############################################################################################################
# Function - CreateDoubleSpline
# This function goes through and creates the primary spline(s)

def CreateDoubleSpline(PDData, segmentIndex, XYZSplineDataArray, DoubleSpline = 1, AngleOffset = 0.0, \
                           SegmentLead = 0):

    NetSplineSectionLength = PDData.BlockSectionLength - (PDData.EdgeStraightSectionLength*2.0) - \
                                 (SegmentLead*PDData.ZSpatialPrecision)

    # Create an XYZ Array for this segment index to pass into the design spline function.
    # It will be returned with data.
    SplineDataArray = np.zeros(((NetSplineSectionLength/PDData.ZSpatialPrecision)+1, 3), dtype=np.float64)

    XStartPos = PDData.PathXStartPosArray[segmentIndex]
    YStartPos = PDData.PathYStartPosArray[segmentIndex]
    XEndPos = PDData.PathXEndPosArray[segmentIndex]
    YEndPos = PDData.PathYEndPosArray[segmentIndex]
    XDistance = PDData.XDistanceArray[segmentIndex]
    YDistance = PDData.YDistanceArray[segmentIndex]
    DirectDistance = PDData.DirectDistanceArray[segmentIndex]
    ZSpatialPrecision = PDData.ZSpatialPrecision
    EdgeStraightSectionLength = PDData.EdgeStraightSectionLength

    print "#################"
    XMidPos = 0.
    YMidPos = 0.
    XDifference = XEndPos - XStartPos
    YDifference = YEndPos - YStartPos

    print "Start/End Positions"
    print XStartPos, YStartPos
    print XEndPos, YEndPos

    XYLineAngle = np.arctan2(YDistance, XDistance)

    print "LineAngle: " + str(XYLineAngle)

    # This while loop will repeat the code below until the spline length comes within the tolerance specified
    IncrementValue = 5.0
    SegmentDistanceDifference = 0.0
    LastSegmentDistanceDifference = 0.0
    ToleranceAchieved = 0
    RunNumber = 1
    SwitchToFlipMethod = 0
    OffsetValue = 5.0

    print "Angle offset: " + str(AngleOffset)

    while(ToleranceAchieved == 0):

        SplineDistance = DesignSplineSection(DirectDistance, NetSplineSectionLength, \
                                    PDData.ZSpatialPrecision, \
                                XStartPos, YStartPos, XDistance, YDistance, SplineDataArray, \
                                DoubleSpline = DoubleSpline, DoubleSplineOffset = OffsetValue, \
                                AngleOffset = AngleOffset)

        # Now configure the data array by extrapolating out the end straight sections
        # Add the left section. X and Y values equal to the starting position.  Fill in Z values.
        FirstSectionJoinIndex = int(EdgeStraightSectionLength/ZSpatialPrecision + SegmentLead)
        XYZSplineDataArray[0:FirstSectionJoinIndex,0] = XStartPos
        XYZSplineDataArray[0:FirstSectionJoinIndex,1] = YStartPos
        ZIncrementArray = np.arange(0.0,EdgeStraightSectionLength+(SegmentLead*ZSpatialPrecision), \
                                        ZSpatialPrecision, dtype=np.float64)
        XYZSplineDataArray[0:FirstSectionJoinIndex,2] = ZIncrementArray

        # Copy the center section.  Increment the z-values to match up with beginning straight section
        SplineDataArray[:,2] += EdgeStraightSectionLength + SegmentLead*ZSpatialPrecision
        SecondSectionJoinIndex = int((EdgeStraightSectionLength + NetSplineSectionLength)/ZSpatialPrecision) + \
                                         SegmentLead + 1
        XYZSplineDataArray[FirstSectionJoinIndex:SecondSectionJoinIndex,:] = SplineDataArray

        #Add in the end section
        ZIncrementArray2 = np.arange(0.0,EdgeStraightSectionLength, ZSpatialPrecision, dtype=np.float64)
        XYZSplineDataArray[SecondSectionJoinIndex:,2] = ZIncrementArray2 + (EdgeStraightSectionLength + \
                                                                NetSplineSectionLength + \
                                                                (1+SegmentLead)*ZSpatialPrecision)
        XYZSplineDataArray[SecondSectionJoinIndex:,0] = XEndPos
        XYZSplineDataArray[SecondSectionJoinIndex:,1] = YEndPos


        # Now see how the returned value compares to the needed spline distance
        NewSegmentDistance = 2*EdgeStraightSectionLength + SegmentLead*ZSpatialPrecision + SplineDistance
        SegmentDistanceDifference = PDData.CalculatedPathDistance - NewSegmentDistance
        if(abs(SegmentDistanceDifference) <= PDData.SegmentLengthPrecision):
            # The segment is within tolerance.  Use these values and exit the function
            ToleranceAchieved = 1
        else:
            ThisRun = np.sign(SegmentDistanceDifference)
            LastRun = np.sign(LastSegmentDistanceDifference)

            if ((ThisRun != LastRun and LastSegmentDistanceDifference != 0.0) or SwitchToFlipMethod == 1):
                # The segment distance is more than the needed distance, but not within tolerance
                # Flip the sign and reduce the amount by 5%
                SwitchToFlipMethod = 1
                IncrementValue = (-0.95) * IncrementValue

            # Now increase the midpoint
            OffsetValue += IncrementValue
            #Update the last run value
            LastSegmentDistanceDifference = SegmentDistanceDifference

        RunNumber += 1
    #end while

    print "Run number: " + str(RunNumber)
    print "Offset Value: " + str(OffsetValue)
    print "SplineDistance: " + str(SplineDistance)

    # Return the total spline distances
    return NewSegmentDistance

############################################################################################################
# Function - CreateSecondarySegments
# This function goes through and creates the secondary spline(s)

def CreateSecondarySegments(PDData, XYZSegmentDataArray):

    #Go through the list of secondary segments and check for any conflicts
    ConflictFlag = 0
    PrimaryConflictFlag = 0

    # KEY 0-red, 1-green, 2-orange, 3-blue, 4-pink, 5-gold, 6-aqua, 7-yellow, 8-white
    # Array should have one less value than total segments, as primary is already created
    AngleOffsetArray = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    DoubleSplineArray = np.array([1,1,1,1,1,1,1])
    LeadInSegmentArray = np.array([0, 0, 0, 0, 0, 0, 0])

    print "Secondary Segment List: " + str(PDData.SecondaryIndexList)

    for i in range(0, len(PDData.SecondaryIndexList)):

        SegmentIndex = PDData.SecondaryIndexList[i]

        XYZSplineDataArray = np.zeros(((PDData.BlockSectionLength/PDData.ZSpatialPrecision)+1, 3), \
                                          dtype=np.float64)

        AngleOffset = AngleOffsetArray[i]
        DoubleSpline = DoubleSplineArray[i]
        SegmentLead = LeadInSegmentArray[i]

        SplineDistance = CreateDoubleSpline(PDData, SegmentIndex, XYZSplineDataArray, \
                           DoubleSpline = DoubleSpline, AngleOffset = AngleOffset, \
                           SegmentLead = SegmentLead)

        TotalDistance = SplineDistance
        print "Segment " + str(SegmentIndex) + " has a distance of " + str(TotalDistance)

        if (SplineDistance > 0):
            XYZSegmentDataArray[SegmentIndex,:,:] = XYZSplineDataArray
            PDData.SegmentCreatedArray[SegmentIndex] = 1

################################################################################################################
# CIRCULAR ARCS
################################################################################################################
# Function - CreatePrimaryArcSegment
# Function to create the primary arc segment

def CreatePrimaryArcSegment(PDData, XYZSegmentDataArray, ArcRadiusDataArray):
    CreateDoubleArcSegment(PDData, XYZSegmentDataArray, PDData.PrimarySegmentNumber)
    ArcRadiusDataArray[PDData.PrimarySegmentNumber] = PDData.PrimarySegmentRadius

################################################################################################################
# Function - CreateDoubleArcSegment
# This function is intended to create a particular segment by using two or three circular arcs connected
# at the intersection of each arc.  Setting parameter MatchLength equal to 1 means the algorithm will
# run recursively until the segment length is matched.

def CreateDoubleArcSegment(PDData, XYZSegmentDataArray, SegNum):

    # The mathematical basis for this function is to minimize the curvature of the segments by representing
    # them as the interconnection of two circular arcs for a basic spline, or if more distance is needed
    # to use three circular arcs.  The math function will be:
    # Radius1*Angle1 + Radius2*Angle2 = Segment Length

    Width = PDData.DirectDistanceArray[SegNum]
    Length = PDData.BlockSectionLength - 2.*PDData.EdgeStraightSectionLength

    # Basic arc would be two circles with identical radius and arc angle
    # Chord length is the hypotenuse formed by the width and length (divided by 2) which is used as
    # the basis for the radius and angle of the arc
    ChordLength = np.sqrt(Length**2 + Width**2)/2.
    # Angle is formed by the length and width and forms the other angle of the triangle
    # formed by the bisector of the chord
    ChordAngle = np.arctan2(Length,Width)
    # The angle and radius of the arc
    ArcAngle = 2.*(np.pi/2. - abs(ChordAngle))
    ArcRadius = (ChordLength/2.)/np.cos(ChordAngle)

    print "********************************************************************"
    print "Primary Segment: " + str(SegNum)
    print "Width of segment is: " + str(Width)

    # Now with this information, we can construct the circle and interpolate points along the arc
    # X below is the single lateral dimension of the arc, which will be translated into X and Y using
    # the angle of x and y formed by the start and end points and Y below is the length or Z dimension
    LeftArcCenterPointX = 0
    LeftArcCenterPointY = 1.*ArcRadius
    XDelta = PDData.ZSpatialPrecision
    XSlices = int(Length/XDelta)
    ArcSegmentDataArray = np.zeros((XSlices+1, 2), dtype=np.float64)
    # The left arc will take care of the arc inflection point at the closest slice
    LeftXArcWidth = np.sin(ChordAngle) * ChordLength
    LeftXSlice = LeftXArcWidth/XDelta
    # The left arc will take care of the center, arc inflection point near halfway
    # Now do a mirror image for the Right Arc with the remaining length
    RightArcCenterPointY = -1.*(ArcRadius-Width)

    for i in range(0, XSlices+1):
        XValue = i*XDelta

        if(i <= LeftXSlice):
            # Equation to find y value is based on x = r*cos(theta) and y = r*sin(theta)
            ThetaValue = np.arcsin(XValue/ArcRadius)
            YValue = LeftArcCenterPointY - (ArcRadius*np.cos(ThetaValue))
        else:
            ThetaValue = np.arcsin((Length - XValue)/ArcRadius)
            YValue = RightArcCenterPointY + (ArcRadius*np.cos(ThetaValue))

        ArcSegmentDataArray[i,0] = XValue
        ArcSegmentDataArray[i,1] = YValue

    #####
    # Now interpolate this curve into X and Y dimensions based on the original line slope
    XYLineAngle = np.arctan2(PDData.XDistanceArray[SegNum],PDData.YDistanceArray[SegNum])
    XStartPos = PDData.PathXStartPosArray[SegNum]
    YStartPos = PDData.PathYStartPosArray[SegNum]
    XEndPos = PDData.PathXEndPosArray[SegNum]
    YEndPos = PDData.PathYEndPosArray[SegNum]

    FirstSectionJoinIndex = int(PDData.EdgeStraightSectionLength/PDData.ZSpatialPrecision)
    XYZSegmentDataArray[SegNum,0:FirstSectionJoinIndex,0] = XStartPos
    XYZSegmentDataArray[SegNum,0:FirstSectionJoinIndex,1] = YStartPos
    ZIncrementArray = np.arange(0.0,PDData.EdgeStraightSectionLength, PDData.ZSpatialPrecision, dtype=np.float64)
    XYZSegmentDataArray[SegNum,0:FirstSectionJoinIndex,2] = ZIncrementArray

    # Loop through ArcSegmentDataArray to extrapolate segment into X and Y values
    for i in range(0, len(ArcSegmentDataArray[:,0])):
        # The Z value equals the x value for the arc segment data array
        XYZSegmentDataArray[SegNum,i+FirstSectionJoinIndex,2] = ArcSegmentDataArray[i,0] + \
                                                                 PDData.EdgeStraightSectionLength
        # Increment the values by X and Y based on line angle
        XYZSegmentDataArray[SegNum,i+FirstSectionJoinIndex,0] = XStartPos + \
                                            (ArcSegmentDataArray[i,1]*np.sin(XYLineAngle))
        XYZSegmentDataArray[SegNum,i+FirstSectionJoinIndex,1] = YStartPos + \
                                    (ArcSegmentDataArray[i,1]*np.cos(XYLineAngle))

    #Add in the end section
    SecondSectionJoinIndex = int((PDData.EdgeStraightSectionLength + Length)/PDData.ZSpatialPrecision)+1
    XYZSegmentDataArray[SegNum,SecondSectionJoinIndex:,2] = ZIncrementArray + \
                                    (PDData.EdgeStraightSectionLength + \
                                    Length + PDData.ZSpatialPrecision)
    XYZSegmentDataArray[SegNum,SecondSectionJoinIndex:,0] = XEndPos
    XYZSegmentDataArray[SegNum,SecondSectionJoinIndex:,1] = YEndPos
    #####

    totalPathDistance = CalculateSegmentLength(PDData, XYZSegmentDataArray, SegNum)
    print "Segment length: " + str(totalPathDistance)

    # Primary segment
    PDData.CalculatedPathDistance = totalPathDistance
    PDData.PrimarySegmentRadius = ArcRadius
    print "Radius length: " + str(ArcRadius)

################################################################################################################
# Function - CreateSecondaryArcSegments
# Function to create the secondary waveguides.  Breaks the 3D waveguide creation up into x prime and y prime
# arcs.  These are interpolated to form a 3D waveguide with respect to the x,y prime axis.  This is then
# translated to x and y based on the radius offset that the endpoints site with respect to the x,y axis.

def CreateSecondaryArcSegments(PDData, XYZSegmentDataArray, ArcRadiusDataArray):

    ReverseXOn = 0
    # This scaling factor works as follows: The x dimension uses the current radius, starting with the
    # radius of the primary waveguide.  The current radius is then multiplied by the scaling factor to
    # give the radius to be used in the y dimension.  The respective triple arc waveguides are created
    # using those radii and then interpolated to give the three dimensional waveguide.  A larger number
    # will scale the y dimension less, whereas a value of 1 should give a 45 degree "rotation".  A
    # negative scaling factor will move the y-value in the negative direction.  If zero, there will
    # be no additional y value

    # KEY 0-red, 1-green, 2-orange, 3-blue, 4-pink, 5-gold, 6-aqua, 7-yellow, 8-white
    RadiusOffsetArray = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    LeadInSegmentArray = np.array([0, 0, 0, 0, 0, 0, 0])

    print "Secondary Segment List: " + str(PDData.SecondaryIndexList)
    for i in range(0, len(PDData.SecondaryIndexList)):
        SegmentIndex = PDData.SecondaryIndexList[i]
        SegmentLead = LeadInSegmentArray[i]
        NetArcSectionLength = PDData.BlockSectionLength - (PDData.EdgeStraightSectionLength*2.0) - \
                              SegmentLead * PDData.ZSpatialPrecision
        XArcDataArray = np.zeros(((NetArcSectionLength/PDData.ZSpatialPrecision)+1, 2), \
                                  dtype=np.float64)
        YArcDataArray = np.zeros(((NetArcSectionLength/PDData.ZSpatialPrecision)+1, 1), \
                                  dtype=np.float64)
        TmpXArcDataArray = np.zeros(((NetArcSectionLength/PDData.ZSpatialPrecision)+1, 1), \
                                  dtype=np.float64)
        PreAdjustYArcDataArray = np.zeros(((NetArcSectionLength/PDData.ZSpatialPrecision)+1, 2), \
                                  dtype=np.float64)
        RadiusOffset = RadiusOffsetArray[i]

        # Angle to scale into x and y
        XYLineAngle = np.arctan2(PDData.XDistanceArray[SegmentIndex],PDData.YDistanceArray[SegmentIndex])
        XStartPos = PDData.PathXStartPosArray[SegmentIndex]
        YStartPos = PDData.PathYStartPosArray[SegmentIndex]
        XEndPos = PDData.PathXEndPosArray[SegmentIndex]
        YEndPos = PDData.PathYEndPosArray[SegmentIndex]
        XWidth = PDData.XDistanceArray[SegmentIndex]
        YWidth = PDData.YDistanceArray[SegmentIndex]
        Length = PDData.BlockSectionLength - 2.*PDData.EdgeStraightSectionLength

        print "******************************************************************"
        print "Segment: " + str(SegmentIndex)

        # The Radius here is based on the primary radius calculated for the primary arc segment
        # The radius (for all 3 circles) will be incrementally decreased, which increases the length of the
        # segment.  When the segment length is achieved, the loop exits
        CurrentRadius = PDData.PrimarySegmentRadius
        OffsetIncrement = CurrentRadius/1000.
        CurrentRadius -= OffsetIncrement
        CurrentYRadius = abs(CurrentRadius * RadiusOffset)
        DistanceMatched = 0
        SegmentDistanceDifference = 0.0
        LastSliceValue = 0
        LastSegmentDistanceDifference = 0.0
        RunNumber = 1
        SwitchToFineMethod = 0
        PreviousCntr2Slice = 0 #TEMP

        while(DistanceMatched == 0):

            # Create the arc based waveguides for x and y.
            CreateTripleArcSegment(CurrentRadius, XArcDataArray, XWidth, NetArcSectionLength, \
                                   PDData.ZSpatialPrecision)
            CreateTripleArcSegment(CurrentYRadius, PreAdjustYArcDataArray, abs(YWidth), NetArcSectionLength, \
                                   PDData.ZSpatialPrecision)

            # The y data needs to be scaled based on the value of the width, as well as the y radius scaling factor
            if(YWidth > 0.0 and RadiusOffset < 0.0):
```

```
    # Flip the array data left to right
    YArcDataArray = np.copy(PreAdjustYArcDataArray[:,1])
    YArcDataArray = YArcDataArray[::-1]
    # Add the width
    YArcDataArray -= YWidth
    # Now flip top to bottom
    YArcDataArray = YArcDataArray[:] * -1.0
    YArcDataArray = YArcDataArray.reshape(len(YArcDataArray[:]),1)
elif(YWidth < 0.0 and RadiusOffset > 0.0):
    # Flip the array data left to right
    YArcDataArray = np.copy(PreAdjustYArcDataArray[:,1])
    YArcDataArray = YArcDataArray[::-1]
    # Add the negative width
    YArcDataArray += YWidth
    YArcDataArray = YArcDataArray.reshape(len(YArcDataArray[:]),1)
elif(YWidth < 0.0 and RadiusOffset <= 0.0):
    # Flip the array data top to bottom
    YArcDataArray = np.copy(PreAdjustYArcDataArray[:,1])
    YArcDataArray = YArcDataArray[:] * -1.0
    YArcDataArray = YArcDataArray.reshape(len(YArcDataArray[:]),1)
else:
    YArcDataArray = np.copy(PreAdjustYArcDataArray[:,1])

# FLIP X FOR REVERSE BEND
if(ReverseXOn == 1):
    TmpXArcDataArray = np.copy(XArcDataArray[:,1])
    # Flip the array data left to right
    TmpXArcDataArray = TmpXArcDataArray[::-1]
    # Now flip top to bottom
    TmpXArcDataArray = TmpXArcDataArray[:] * -1.0
    # Add the width
    TmpXArcDataArray += XWidth
    XArcDataArray[:,1] = TmpXArcDataArray

# Move the segment interpolation to this section, along with the creation of a triple arc
# x, triple arc y curve.  The portion before this should consist of using the ROC from
# the primary arc, and adjusting that in each triple arc creation section.

# Now interpolate this curve into X and Y dimensions based on the original line slope
FirstSectionJoinIndex = int(PDData.EdgeStraightSectionLength/PDData.ZSpatialPrecision + \
                            SegmentLead)
XYZSegmentDataArray[SegmentIndex,0:FirstSectionJoinIndex,0] = XStartPos
XYZSegmentDataArray[SegmentIndex,0:FirstSectionJoinIndex,1] = YStartPos
ZIncrementArray = np.arange(0.0,PDData.EdgeStraightSectionLength+(SegmentLead * \
                        PDData.ZSpatialPrecision), PDData.ZSpatialPrecision, dtype=np.float64)
XYZSegmentDataArray[SegmentIndex,0:FirstSectionJoinIndex,2] = ZIncrementArray

# Loop through ArcSegmentDataArray to extrapolate segment into X and Y values
for i in range(0, len(XArcDataArray[:,0])):
    # The Z value equals the x value for the arc segment data array
    XYZSegmentDataArray[SegmentIndex,i+FirstSectionJoinIndex,2] = XArcDataArray[i,0] + \
                            PDData.EdgeStraightSectionLength + \
                            (SegmentLead * PDData.ZSpatialPrecision)
    # Now insert the x and y waveguide data
    XYZSegmentDataArray[SegmentIndex,i+FirstSectionJoinIndex,0] = XStartPos + XArcDataArray[i,1]
    XYZSegmentDataArray[SegmentIndex,i+FirstSectionJoinIndex,1] = YStartPos + YArcDataArray[i]

#Add in the end section
SecondSectionJoinIndex = int((PDData.EdgeStraightSectionLength + NetArcSectionLength) \
                             /PDData.ZSpatialPrecision) + SegmentLead + 1
ZIncrementArray2 = np.arange(0.0,PDData.EdgeStraightSectionLength, PDData.ZSpatialPrecision, \
                             dtype=np.float64)
XYZSegmentDataArray[SegmentIndex,SecondSectionJoinIndex:,2] = ZIncrementArray2+ \
                (PDData.EdgeStraightSectionLength + NetArcSectionLength + \
                (1+SegmentLead)*PDData.ZSpatialPrecision)
XYZSegmentDataArray[SegmentIndex,SecondSectionJoinIndex:,0] = XEndPos
XYZSegmentDataArray[SegmentIndex,SecondSectionJoinIndex:,1] = YEndPos
#####

totalPathDistance = CalculateSegmentLength(PDData, XYZSegmentDataArray, SegmentIndex)

# Determine if path length is within tolerance
SegmentDistanceDifference = PDData.CalculatedPathDistance - totalPathDistance
if(abs(SegmentDistanceDifference) <= PDData.SegmentLengthPrecision):
    DistanceMatched = 1
else:
    ThisRun = np.sign(SegmentDistanceDifference)
    LastRun = np.sign(LastSegmentDistanceDifference)

    if ((ThisRun != LastRun and LastSegmentDistanceDifference != 0.0) or SwitchToFineMethod == 1):
        # The segment distance is more than the needed distance, but not within tolerance
        # To fix this we go back by twice the amount of the offset increment and then adjust
        # the radius at an offset of one tenth the amount of the previous offset
        if(SwitchToFineMethod == 0):
            print "FINE ADJUSTMENT"
            CurrentRadius += 2*OffsetIncrement
            OffsetIncrement = OffsetIncrement*0.1
        SwitchToFineMethod = 1

    #Update the last run value
    LastSegmentDistanceDifference = SegmentDistanceDifference
    CurrentRadius -= OffsetIncrement
    CurrentYRadius = abs(CurrentRadius * RadiusOffset)

RunNumber += 1
```

```
        print CurrentRadius
        print CurrentYRadius

        ArcRadiusDataArray[SegmentIndex] = CurrentRadius

        print SegmentLead, NetArcSectionLength

        print "Radius length: " + str(CurrentRadius)
        print "Total runs: " + str(RunNumber)

################################################################################################################
# Function - CreateTripleArcSegment
# This function is intended to create a particular segment by using three circular arcs connected at the
# intersection of each arc.  In addition it uses an additional angle to rotate the shapes in space.

def CreateTripleArcSegment(CurrentRadius, ArcDataArray, Width, Length, XDelta):

    # The segment here is created using three arcs based on circles of identical radii.  Looking from
    # the side, the first arc will be constructed from the bottom-right portion of circle 1, with
    # the bottom at the input, the third arc will be constructed from the bottom-left portion of circle
    # 3, with the remaining portion of the segment being created by the top part of circle 2, with
    # the arc going from where the circle touches the other two.
    # The math function will be:
    # Radius1*Angle1 + Radius2*Angle2 + Radius3*Angle3 = Segment Length
    # The centers of circles 1 through 3 all have an X and Y coordinate
    Center1X = 0.0
    Center3X = Length

    # If the current radius is set to zero, calculate what radius will make for a simple double arc curve
    if(CurrentRadius == 0.0):
        ChordLength = np.sqrt(Length**2 + Width**2)/2.
        ChordAngle = np.arctan2(Length,Width)
        CurrentRadius = (ChordLength/2.)/np.cos(ChordAngle)

    # First calculate the locations of circles 1 and 3.  Start point on circle 1 is 0,0
    Center1Y = CurrentRadius
    Center3Y = CurrentRadius + Width
    # This is the distance between the two centers of circles 1 and 3
    DistanceCenters13 = np.sqrt((Center3Y-Center1Y)**2 + (Center3X-Center1X)**2)
    # Angles 1-3 are the arc angles that will be used for circles 1-3 in the segment
    # Angle 4 is calculated by using the bisect of the Distance of the centers
    # Angle 4 is for both arcs, as the bisect creates an isoceles triangle
    Angle4 = np.arccos((DistanceCenters13/2)/(2*CurrentRadius))
    Angle1 = np.pi/2. + np.arcsin(abs(Center3Y-Center1Y)/DistanceCenters13) - Angle4
    Angle3 = np.arccos(abs(Center3Y-Center1Y)/DistanceCenters13) - Angle4
    Center2X = 2*CurrentRadius*np.sin(Angle1) + Center1X
    Center2Y = -2*CurrentRadius*np.cos(Angle1) + Center1Y

    # Intersection point 1 is where arc 1 meets arc 2, point 2 is where arc 2 meets arc 3
    IntersectPoint1X = CurrentRadius*np.sin(Angle1)
    IntersectPoint1Y = Center1Y - CurrentRadius*np.cos(Angle1)
    IntersectPoint2X = Length - CurrentRadius*np.sin(Angle3)
    IntersectPoint2Y = Center3Y - CurrentRadius*np.cos(Angle3)

    # Now construct the segment starting from Arc 1 and build the data array.
    XSlices = int(Length/XDelta)

    # The intersect points may or may not coincide with an x slice value.  This is OK, but
    # the surrounding slices need to be identified as to which arc to use for each
    Arc1Length = np.sin(Angle1) * CurrentRadius
    Arc1XSlice = Arc1Length/XDelta

    # Arc2 X Slices go from Arc1 Slices + 1 to Arc2 X Slices.  Arc 2 end point is at Arc 3
    Arc3Length = np.sin(Angle3) * CurrentRadius
    Arc2XSlice = XSlices - Arc3Length/XDelta

    for i in range(0, XSlices+1):
        XValue = i*XDelta
        if(i <= Arc1XSlice):
            # Equation to find y value is based on x = r*cos(theta) and y = r*sin(theta)
            ThetaValue = np.arcsin(XValue/CurrentRadius)
            YValue = Center1Y - (CurrentRadius*np.cos(ThetaValue))
        elif(i > Arc1XSlice and i <= Arc2XSlice):
            # Use arc 2
            ThetaValue = np.arcsin(abs(Center2X-(i*XDelta))/CurrentRadius)
            YValue = Center2Y + (CurrentRadius*np.cos(ThetaValue))
        else:
            # Use arc 3
            ThetaValue = np.arcsin((Length-XValue)/CurrentRadius)
            YValue = Center3Y - (CurrentRadius*np.cos(ThetaValue))

        ArcDataArray[i,0] = XValue
        ArcDataArray[i,1] = YValue

################################################################################################################
################################################################################################################
# FINAL CHECK

def SegmentDistanceCurvatureCheck(XYZSegmentDataArray,PDData, WPData, ArcRadiusDataArray):

    print "***** CURVATURES *****"
    SegmentCreatedArray = PDData.SegmentCreatedArray
    MinimumSeparationDistance = PDData.MinimumPathSeparationDist
    ZSpatialPrecision = PDData.ZSpatialPrecision
```

```
PrimarySegmentNumber = PDData.PrimarySegmentNumber
PrimaryWaveguideLength = PDData.CalculatedPathDistance

# Loop through each segment and at each point in the segment, compare the separation distance
# to each of the other segments.
ClosestDistance = 10000.
NoSegmentConflictFound = 1
MinimumYDistance = 10000.
MaximumYDistance = 0.
MinYDistZValue = 0.
MaxYDistZValue = 0.
MinimumYDistSegment = 0
MaximumYDistSegment = 0
ClosestSegment1 = 0
ClosestSegment2 = 0
ClosestZValue = 0.
NumberOfSegments = len(SegmentCreatedArray)
NumPoints = len(XYZSegmentDataArray[0,:,0])
IsPrimary = 1

# Outer loop is the segment to check
for i in range(0,NumberOfSegments):
    #Middle loop is each segment to verify against.  Need a flag variable to exit loop if conflict found
    for j in range(0,NumberOfSegments):
        if(i != j):
            # Not the same segment, so step through each point in the array.  Inner loop is the z values
            for k in range(0,len(XYZSegmentDataArray[0,:,0])):
                XDiff = XYZSegmentDataArray[i,k,0] - XYZSegmentDataArray[j,k,0]
                YDiff = XYZSegmentDataArray[i,k,1] - XYZSegmentDataArray[j,k,1]
                DirectDifference = np.sqrt((XDiff)**2 + (YDiff)**2)

                # Also want to record the closest distance between any two lines
                if(DirectDifference < ClosestDistance):
                    ClosestDistance = DirectDifference
                    ClosestSegment1 = i
                    ClosestSegment2 = j
                    ClosestZValue = XYZSegmentDataArray[j,k,2]

                # Find the minimum and maximum y values for laser burn distance
                if(XYZSegmentDataArray[i,k,1] < MinimumYDistance):
                    MinimumYDistance = XYZSegmentDataArray[i,k,1]
                    MinimumYDistSegment = i
                    MinYDistZValue = XYZSegmentDataArray[j,k,2]
                if(XYZSegmentDataArray[i,k,1] > MaximumYDistance):
                    MaximumYDistance = XYZSegmentDataArray[i,k,1]
                    MaximumYDistSegment = i
                    MaxYDistZValue = XYZSegmentDataArray[j,k,2]

    # Also need to calculate the radius of curvature.  Take the difference in x and y and calculate
    # it as a radial difference versus the distance in z.  This gives an angle in cylindrical coordinates
    # versus z.  The arctan of that gives the angle.  Once the angle is calculated, take the difference
    # in angle versus the difference in length of the segment for the radius of curvature.
    MinimumROC = 1.0e30
    MaximumCurvature = 0.
    ROCZValue = 0
    CurvatureZValue = 0

    SlopeAngleArray = np.zeros(NumPoints, dtype=np.float64)
    SlopeAngleArray[0] = 0  # due to horizontal input
    CurvatureArray = np.zeros(NumPoints, dtype=np.float64)
    AngleCurvatureArray = np.zeros(NumPoints, dtype=np.float64)
    RadiusOfCurvatureArray = np.zeros(NumPoints, dtype=np.float64)
    SegmentDistanceArray = np.zeros(NumPoints+1, dtype=np.float64)
    XDifferenceArray = np.zeros(NumPoints+1, dtype=np.float64)

    # Calculate the segment distances for the entire array
    # End segment distances are arbitrarily set to a value to calculate the angle difference
    ArbitraryEndSegLength = 25.
    SegmentDistanceArray[0] = ArbitraryEndSegLength
    SegmentDistanceArray[NumPoints] = ArbitraryEndSegLength

    for m in range(1, NumPoints):
        SegmentDistanceArray[m] = \
                np.sqrt((XYZSegmentDataArray[i,m,0] - XYZSegmentDataArray[i,m-1,0])**2 + \
                        (XYZSegmentDataArray[i,m,1] - XYZSegmentDataArray[i,m-1,1])**2 + \
                        (XYZSegmentDataArray[i,m,2] - XYZSegmentDataArray[i,m-1,2])**2)


    for m in range(0, NumPoints):
        LeftVector = np.zeros(3, dtype=np.float64)
        RightVector = np.zeros(3, dtype=np.float64)

        # Calculate vector direction
        if(m == 0):
            LeftVector[2] = ArbitraryEndSegLength
        else:
            LeftVector[0] = XYZSegmentDataArray[i,m,0] - XYZSegmentDataArray[i,m-1,0]
            LeftVector[1] = XYZSegmentDataArray[i,m,1] - XYZSegmentDataArray[i,m-1,1]
            LeftVector[2] = XYZSegmentDataArray[i,m,2] - XYZSegmentDataArray[i,m-1,2]

        if(m == NumPoints-1):
            RightVector[0] = XYZSegmentDataArray[i,m,0] - XYZSegmentDataArray[i,m-1,0]
            RightVector[1] = XYZSegmentDataArray[i,m,0] - XYZSegmentDataArray[i,m-1,0]
            RightVector[2] = ArbitraryEndSegLength
        else:
                RightVector[0] = XYZSegmentDataArray[i,m+1,0] - XYZSegmentDataArray[i,m,0]
                RightVector[1] = XYZSegmentDataArray[i,m+1,1] - XYZSegmentDataArray[i,m,1]
                RightVector[2] = XYZSegmentDataArray[i,m+1,2] - XYZSegmentDataArray[i,m,2]

            if(SegmentDistanceArray[m] > 0 and SegmentDistanceArray[m+1] > 0):
                VectorDifference = ((LeftVector[0]*RightVector[0]) + \
                                             (LeftVector[1]*RightVector[1]) + \
                                             (LeftVector[2]*RightVector[2]))/ \
                                             (SegmentDistanceArray[m]*SegmentDistanceArray[m+1])

                # arc cosine of 90 degrees is infinity
                if(int(VectorDifference) == 1):
                    AngleCurvatureArray[m] = 0
                else:
                    AngleBetweenSegs = np.arccos(VectorDifference)
                    # Curvature is based on angle and length of the two segments
                    AngleCurvatureArray[m] = (2.*AngleBetweenSegs)/ \
                          (SegmentDistanceArray[m] + SegmentDistanceArray[m+1])

                if(AngleCurvatureArray[m] == 0):
                    RadiusOfCurvatureArray[m] = 1.0e30
                else:
                    RadiusOfCurvatureArray[m] = 1/AngleCurvatureArray[m]

                # The X Difference is to see if a transition event has occurred in which the curve
                # in the x direction changes direction.  Only done in the x dimension, since it
                # accounts for the majority of the curvature.
                XDifferenceArray[m] = RightVector[0] - LeftVector[0]

                if(XDifferenceArray[m] < 0.0):
                    RadiusOfCurvatureArray[m] = -1.0 * RadiusOfCurvatureArray[m]

            else:
                AngleCurvatureArray[m] = 0.
                RadiusOfCurvatureArray[m] = 0.

            if(abs(AngleCurvatureArray[m]) > MaximumCurvature):
                MaximumCurvature = abs(AngleCurvatureArray[m])
                CurvatureZValue = XYZSegmentDataArray[i,m,2]

            if(abs(RadiusOfCurvatureArray[m]) < MinimumROC and RadiusOfCurvatureArray[m] != 0):
                MinimumROC = abs(RadiusOfCurvatureArray[m])
                ROCZValue = XYZSegmentDataArray[i,m,2]

        # Now we can calculate the power loss for this waveguide
        if(i == PrimarySegmentNumber):
            IsPrimary = 1
        else:
            IsPrimary = 0
        CalculatePowerLoss(ZSpatialPrecision, RadiusOfCurvatureArray, ArcRadiusDataArray[i], IsPrimary, \
                           PrimaryWaveguideLength, WPData)

        print "CURVATURE FOR WAVEGUIDE " + str(i)
        print "Minimum radius of curvature is " + str(MinimumROC) + \
            " at a Z distance of " + str(ROCZValue) + " microns."

    # Print Summary Values
    print "PROXIMITY AND HEIGHT VALUES"
    print "Closest Lateral Distance in the array is " + str(ClosestDistance) + " microns between segment " + \
          str(ClosestSegment1) + " and segment " + str(ClosestSegment2) + " at a Z distance of " + \
          str(ClosestZValue) + " microns."
    print "Maximum Y Value (closest to top) is " + str(MaximumYDistance) + " for segment " + \
          str(MaximumYDistSegment) + " at a Z distance of " + str(MaxYDistZValue) + " microns."
    print "Minimum Y Value (furthest from top) is " + str(MinimumYDistance)  + " for segment " + \
          str(MinimumYDistSegment) + " at a Z distance of " + str(MinYDistZValue) + " microns."

    print "************************************************************"

##############################################################################################################
# CalculatePowerLoss
# This function attempts to estimate the power loss in each waveguide, based on calculations from
# Snyder and Love's textbook.  It uses the calculated waveguide radius of curvature at each point.

def CalculatePowerLoss(ZSpatialPrecision, ROCArray, ArcRadius, IsPrimary, PrimaryWaveguideLength, WPData):

    # The normalized power is set to 1.0 and then the loss from that input power is incremented
    # as z increases.  This gives the total power lost FROM CURVATURE ONLY at the end.

    NumSegments = len(ROCArray)
    PowerLossArray = np.zeros(NumSegments, dtype=np.float64)
    StartInputPower = 1.0
    TotalPowerLoss = 0.0
    PreviousROC = 0.0
    TotalGamma = 0.0
    TotalBendLossExp = 0.0
    TotalTransPower = 1.0
    TotalBendPower = 1.0

    # Loop through each segment piece that makes up the waveguide
    for i in range(0,NumSegments):
        CurrentROC = ROCArray[i]

        # Calculate the loss due to bends here (for a step profile)
        if(CurrentROC != 0 and abs(CurrentROC) != 1.0e30):
            # For Step profiles
            CalculateSectionBendPowerLoss(CurrentROC, ZSpatialPrecision, WPData)

            # For Gaussian profiles
##            CalculateSectionBendPowerLossGaussian(CurrentROC, ZSpatialPrecision, WPData)

            # To calculate bend loss for an HPO profile based on physical measurements
##            BendPower = HPOLookupBendLoss(CurrentROC, ZSpatialPrecision)

        else:
            GammaCoefficient = 0
            BendPower = 1.0

        TotalBendPower = TotalBendPower*BendPower

        # Calculate Transition Loss for splines
##        TransPowerLoss = CalculateSectionsTransitionLoss(CurrentROC, PreviousROC, WPData)
##        TotalTransPower = TotalTransPower * (1.0 - TransPowerLoss)

        PreviousROC = CurrentROC

    # Or for arc waveguide, just calculate the transition loss for the entire waveguide
    TotalTransPower = CalculateArcTransitionLoss(WPData, ArcRadius, IsPrimary)

    print "Power After Bend Loss: " + str(TotalBendPower)
    print "Power After Transition Loss: " + str(TotalTransPower)
    EndPower = TotalBendPower * TotalTransPower
    print "Total Power after Bend & Transition Losses: " + str(EndPower)
    # Last bit, incorporate the coupling and bend losses
    print "PrimaryWaveguideLength: " + str(PrimaryWaveguideLength)
    AddedLossPower = 0.91 *  EndPower * np.exp(-0.0075 * (PrimaryWaveguideLength/1000.))

    print "Total Power after all losses: " + str(AddedLossPower)
    print ""

##############################################################################################################
# HPOLookupBendLoss
# This function takes the values of bend loss per mm calculated from the right angle parameter scan result
# and return a bend loss value.

def HPOLookupBendLoss(InputRadius, SegmentLength):
    # The normalized power is set to 1.0 and then the loss from that input power is incremented
    # as z increases.  This gives the total power lost FROM CURVATURE ONLY at the end.
    StartInputPower = 1.0
    CurrentROC = abs(InputRadius)

    # Radius and respective power loss per mm values
    RadiusLookupArray = np.array([10000.0,13300.0,16600.0,20000.0,23300.0,26600.0,30000.0,33300.0,36600.0,\
                                40000.0], dtype=np.float64)
    BendLossLookupArray = np.array([0.437972203,0.18150836,0.078795534,0.040054237,0.017996858,0.013146017,\
                                0.008745052,0.006998815,0.005485012,0.004484166], dtype=np.float64)

    # Calculate the loss due to bends here (for a step profile).  Interpolate between data points or
    # extrapolate from end points
    NumValues = len(BendLossLookupArray)-1
    FindLoss = interpolate.interp1d(RadiusLookupArray, BendLossLookupArray)
    BendLoss = 0.0

    # If the values are outside the range, they will just be linearly interpolated.  Most values will
    # be within this range.  Unlikely to have smaller radii, but lareger ones should not have much effect anyway
    if(CurrentROC < RadiusLookupArray[0]):
        Slope = (BendLossLookupArray[1]-BendLossLookupArray[0])/(RadiusLookupArray[0]-RadiusLookupArray[1])
        BendLoss = (RadiusLookupArray[0]-CurrentROC)*Slope + BendLossLookupArray[0]
    elif(CurrentROC > RadiusLookupArray[NumValues]):
        Slope = (BendLossLookupArray[NumValues]-BendLossLookupArray[NumValues-1])/\
                  (RadiusLookupArray[NumValues]-RadiusLookupArray[NumValues-1])
        BendLoss = BendLossLookupArray[NumValues] - (RadiusLookupArray[NumValues]-CurrentROC)*Slope
        if(BendLoss < 0):
            BendLoss = 0
    else:
        BendLoss = FindLoss(CurrentROC)

    # Scale this bend loss amount by relation of length to 1 mm (1000 microns)
    BendLoss = BendLoss * SegmentLength/1000.0

    # The power loss is Pout = Pin*(1-Loss)
    OutputPower = StartInputPower*(1.-BendLoss)
    return OutputPower

##############################################################################################################
# CalculateSectionBendPowerLoss
# This function attempts to estimate the power loss in each waveguide, based on calculations from
# Snyder and Love's textbook.  It uses the calculated waveguide radius over a whole section with a
# constant radius and specified length.

def CalculateSectionBendPowerLoss(CurrentROC, SegmentLength, WPData):

    # The normalized power is set to 1.0 and then the loss from that input power is incremented
    # as z increases.  This gives the total power lost FROM CURVATURE ONLY at the end.
    StartInputPower = 1.0

    # Calculate the loss due to bends here (for a step profile)
    if(CurrentROC != 0):
        StepProfileCoefficient = (np.sqrt(np.pi)/(2*WPData.WaveguideRadius)) * \
                              (np.sqrt(WPData.WaveguideRadius/CurrentROC))
        AreaCoefficient = WPData.UParameter**2/(WPData.NormalizedFrequency**2 * \
                            WPData.WParameter**1.5 * (scisp.kn(1,WPData.WParameter)**2))
        ExponentCoefficient = (-4./3)*(CurrentROC/WPData.WaveguideRadius) * \
                              ((WPData.WParameter**3 * WPData.DeltaParameter)/(WPData.NormalizedFrequency**2))
        GammaCoefficient =  StepProfileCoefficient * AreaCoefficient * np.exp(ExponentCoefficient)
    else:
        GammaCoefficient = 0

    # The power loss is Pout = Pin*exp(-Gamma*z)
    OutputPower = StartInputPower*np.exp(-1.*GammaCoefficient*SegmentLength)

    return OutputPower

############################################################################################################
# CalculateSectionBendPowerLossGaussian
# This function attempts to estimate the power loss in each waveguide, based on calculations from
# Snyder and Love's textbook.  It uses the calculated waveguide radius over a whole section with a
# constant radius and specified length.

def CalculateSectionBendPowerLossGaussian(CurrentROC, SegmentLength, WPData):

    # The normalized power is set to 1.0 and then the loss from that input power is incremented
    # as z increases.  This gives the total power lost FROM CURVATURE ONLY at the end.
    StartInputPower = 1.0

    # Calculate the loss due to bends here for a gaussian profile
    if(CurrentROC != 0):
        VMinus1 = WPData.NormalizedFrequency - 1.
        VPlus1 = WPData.NormalizedFrequency + 1.

        GaussCoefficient1 = (np.sqrt(np.pi)/(2.*WPData.WaveguideRadius)) * \
                                 (np.sqrt(WPData.WaveguideRadius/CurrentROC))
        GaussCoefficient2 = WPData.NormalizedFrequency**4/(VPlus1**2 * np.sqrt(VMinus1))
        ExponentCoefficient = (VMinus1**2 / VPlus1) - ((4./3)*(abs(CurrentROC)/WPData.WaveguideRadius) * \
                              ((VMinus1**3 * WPData.DeltaParameter)/(WPData.NormalizedFrequency**2)))
        GammaCoefficient =  GaussCoefficient1 * GaussCoefficient2 * np.exp(ExponentCoefficient)
    else:
        GammaCoefficient = 0

    # The power loss is Pout = Pin*exp(-Gamma*z)
    OutputPower = StartInputPower*np.exp(-1.*GammaCoefficient*SegmentLength)
    PowerLoss = 1.0 - OutputPower

    return OutputPower

############################################################################################################
# CalculateSectionsTransitionLoss
# This function attempts to estimate the power loss from transitions between two different radii of curvature
# from Snyder and Love's textbook.  This section calculation allows for two different ROC values to be set.
# Straight section calculated as 1.0e30

def CalculateSectionsTransitionLoss(Radius1, Radius2, WPData):

    TransitionCoefficient = 0.0
    if(Radius1 != 0.0 and Radius1 != 1.0e30 and Radius2 == 1.0e30):
        TransitionCoefficient = 1./Radius1**2
    elif(Radius2 != 0.0 and Radius2 != 1.0e30 and Radius1 == 0.0):
        TransitionCoefficient = 1./Radius2**2
    elif(Radius2 != 0.0 and Radius1 != 0.0 and Radius1 != 1.0e30 and Radius2 != 1.0e30):
        # The current ROC can be positive and the previous ROC can be negative and vice versa
        RadiiSum = 0.0
        if((Radius1 > 0.0 and Radius2 < 0.0) or (Radius2 > 0.0 and Radius1 < 0.0)):
            RadiiSum = abs(Radius1) + abs(Radius2)
        else:
            RadiiSum = Radius1 - Radius2
        TransitionCoefficient = (RadiiSum/(Radius1*Radius2))**2

    TransitionLoss = TransitionCoefficient * \
                   ((WPData.WaveguideRadius**2 * WPData.NormalizedFrequency**4)/ \
                    (8*WPData.DeltaParameter**2)) * \
                   (WPData.RadiusZero/WPData.WaveguideRadius)**6

    return TransitionLoss

############################################################################################################
# CalculateArcTransitionLoss
# This function attempts to estimate the power loss from transitions between two different radii of curvature
# from Snyder and Love's textbook.  Rather than evaluate a discrete curvature at the radius for each section,
# this routine takes into account that the radius is the same over the arc portions of the curve and will
# have three transition events for a primary waveguide and four for a secondary waveguide

def CalculateArcTransitionLoss(WPData, Radius, IsPrimary):

    TransitionCoefficient = 0.0
    TotalTransitionLoss = 1.0  # the remaining power after transition losses

    # All arc curves have two transitions from a straight section to the radius of the arcs at each end

    TransitionCoefficient = 1./Radius**2
    TransitionLoss = TransitionCoefficient * \
                   ((WPData.WaveguideRadius**2 * WPData.NormalizedFrequency**4)/ \
                    (8*WPData.DeltaParameter**2)) * \
                   (WPData.RadiusZero/WPData.WaveguideRadius)**6
    TotalTransitionLoss = TotalTransitionLoss * (1.0 - TransitionLoss)

    TransitionCoefficient = 1./Radius**2
    TransitionLoss = TransitionCoefficient * \
                ((WPData.WaveguideRadius**2 * WPData.NormalizedFrequency**4)/ \
                 (8*WPData.DeltaParameter**2)) * \
                (WPData.RadiusZero/WPData.WaveguideRadius)**6
    TotalTransitionLoss = TotalTransitionLoss * (1.0 - TransitionLoss)

    # Both primary and secondary waveguides have one transition where two opposite bends meet
    TransitionCoefficient = (2*Radius/(Radius*Radius))**2
    TransitionLoss = TransitionCoefficient * \
                ((WPData.WaveguideRadius**2 * WPData.NormalizedFrequency**4)/ \
                 (8*WPData.DeltaParameter**2)) * \
                (WPData.RadiusZero/WPData.WaveguideRadius)**6
    TotalTransitionLoss = TotalTransitionLoss * (1.0 - TransitionLoss)

    # Secondary waveguides have one additional opposite transtion
    if(IsPrimary != 1):
        TransitionCoefficient = (2*Radius/(Radius*Radius))**2
        TransitionLoss = TransitionCoefficient * \
                    ((WPData.WaveguideRadius**2 * WPData.NormalizedFrequency**4)/ \
                     (8*WPData.DeltaParameter**2)) * \
                    (WPData.RadiusZero/WPData.WaveguideRadius)**6
        TotalTransitionLoss = TotalTransitionLoss * (1.0 - TransitionLoss)

    return TotalTransitionLoss

#############################################################################################################
# PLOT SEGMENTS

def PlotAllSegments(XYZSegmentDataArray):
    # Remember that z values are plotting on the y-axis here and y values on z-axis.  The plots obey the
    # right-hand rule, so this orients them in the proper direction.  y and z axes will be swapped on plot
    line0 = mlab.plot3d(XYZSegmentDataArray[0,:,0], XYZSegmentDataArray[0,:,2], XYZSegmentDataArray[0,:,1], \
                    tube_radius=10, tube_sides =12, colormap = 'Spectral', color = (0.5,0,0))
    line1 = mlab.plot3d(XYZSegmentDataArray[1,:,0], XYZSegmentDataArray[1,:,2], XYZSegmentDataArray[1,:,1], \
                    tube_radius=10, tube_sides =12, color = (0,0.5,0))
    line2 = mlab.plot3d(XYZSegmentDataArray[2,:,0], XYZSegmentDataArray[2,:,2], XYZSegmentDataArray[2,:,1], \
                    tube_radius=10, tube_sides =12, color = (1,0.5,0))
    line3 = mlab.plot3d(XYZSegmentDataArray[3,:,0], XYZSegmentDataArray[3,:,2], XYZSegmentDataArray[3,:,1], \
                    tube_radius=10, tube_sides =12, color = (0,0.5,0.9))
    line4 = mlab.plot3d(XYZSegmentDataArray[4,:,0], XYZSegmentDataArray[4,:,2], XYZSegmentDataArray[4,:,1], \
                    tube_radius=10, tube_sides =12, color = (0.8,0.4,0.8))
    line5 = mlab.plot3d(XYZSegmentDataArray[5,:,0], XYZSegmentDataArray[5,:,2], XYZSegmentDataArray[5,:,1], \
                    tube_radius=10, tube_sides =12, color = (1.0,0.75,0.0))
    line6 = mlab.plot3d(XYZSegmentDataArray[6,:,0], XYZSegmentDataArray[6,:,2], XYZSegmentDataArray[6,:,1], \
                    tube_radius=10, tube_sides =12, color = (0.125,0.6,0.65))
    line7 = mlab.plot3d(XYZSegmentDataArray[7,:,0], XYZSegmentDataArray[7,:,2], XYZSegmentDataArray[7,:,1], \
                    tube_radius=10, tube_sides =12, color = (1.0,1.0,0))
    return

#############################################################################################################
# CREATE FILE FEEDS
#
# This function creates the laser write file that will dictate to the laser write mechanism the positions
# to write each segment of the laser.  The segments will be ordered from bottom to top in the "y direction"
# however for this format, the z-direction becomes the y-axis for the laser, the y direction becomes the
# z-axis, and the x direction remains the same.

def CreateLaserWriteAndRSoftFiles(XYZSegmentDataArray):

    # Number of positions is the number of points in the z-direction
    now = datetime.datetime.now()
    NewDirName = "30mmchip" + str(now.year) + str(now.month) + str(now.day) + str(now.hour) + \
                      str(now.minute) + str(now.second)
    print "Directory Created: " + NewDirName

    os.mkdir(NewDirName)
    os.chdir(NewDirName)

    # TURKEY
    FileLaserPoints = open('8GuideArcSideStepDesignBRedundant.txt','w')
    FileLaserPoints.write("8 segment, arc based sidestep, bends reversed, redundant output\n")
    SegmentOrderArray = np.array([3,1,5,0,6,2,4,7])

    # First loop goes through all segments
    NumberOfSegments = len(XYZSegmentDataArray[:,0,0])
    NumberOfPositions = len(XYZSegmentDataArray[0,:,2])

    for i in range(0, NumberOfSegments):
        # Use the segment order array for laser write file
        seg = SegmentOrderArray[i]
        print "Segment # " + str(seg)

        # Put a note here for the segment number
        FileLaserPoints.write("Segment " + str(i+1) + "\n")
        FileLaserPoints.write("\n")

        # A line needs to be added at the beginning to set the laser write at 2 mm from the input values

        # Change - all values multiplied by 0.001 to get mm values.
        XLead = XYZSegmentDataArray[seg,0,0]*-0.001
        ZLead = XYZSegmentDataArray[seg,0,1]*-0.001
        YLead = (-2.0)
        LineToWrite = "g1\tx\t" + str(XLead) + "\ty\t" + str(YLead) + "\tz\t" + str(ZLead) + "\n"
        FileLaserPoints.write(LineToWrite)
        for j in range(0, NumberOfPositions):
            XValue = XYZSegmentDataArray[seg,j,0]
            YValue = XYZSegmentDataArray[seg,j,2]
            ZValue = XYZSegmentDataArray[seg,j,1]

            # All x and z values should be multiplied by -1 for laser write file
            if (XValue != 0.0):
                XValue = XValue * -0.001
            if (ZValue != 0.0):
                ZValue = ZValue * -0.001
            YValue = YValue * 0.001

            XString = "%.4f" % XValue
            YString = "%.4f" % YValue
            ZString = "%.4f" % ZValue
            LineToWrite = "g1\tx\t" + XString + "\ty\t" + YString + "\tz\t" + ZString + "\n"
            FileLaserPoints.write(LineToWrite)

        # the end should have an empty line
        FileLaserPoints.write("\n")

    FileLaserPoints.close()
    print "FILE WRITE COMPLETE"

##############################################################################################################
##############################################################################################################
# MAIN PROGRAM VARIABLES
##############################################################################################################

import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate, randn
import scipy.special as scisp
from enthought.mayavi import mlab
from enthought import mayavi
import datetime
import os
##from pytools import nmpfit

# Number of segments to be created
NumberOfSegments = 8

# Declare position and dimension data object
PosDimData = PositionDimensionDataObject(NumberOfSegments)
WavePropData = WaveguidePropertiesDataObject

# Optical Section Dimensions (microns)
PosDimData.BlockSectionWidth = 8000   # "X" dimension   #Physical chip width 8000
PosDimData.BlockSectionHeight = 1000  # "Y" dimension   #Physical chip height 1100
PosDimData.BlockSectionLength = 30000 # "Z" dimension

# Number of segments to be created
PosDimData.NumberOfSegments = NumberOfSegments

# Initialize and calculate waveguide properties data
CalculateWaveguideProperties(WavePropData)

##############################################################################################################
# START POINTS
##############################################################################################################

### DESIGN FOR APRIL 2011

# New Design, 8 hole, blue dots of diagram
PosDimData.PathXStartPosArray = [-77.94,-51.96,-51.96,0.0,25.98,51.96,77.94,77.94]
PosDimData.PathYStartPosArray = [15.0,60.0,-60.0,90.0,-45.0,60.0,15.0,-45.0]

##############################################################################################################
# END POINTS
##############################################################################################################

# 8 segment redundant array, shifted in +x direction by 5 mm
PosDimData.PathXEndPosArray = [4875.0,4375.0,5375.0,4125.0,5625.0,4625.0,5125.0,5875.0]   # Splines
##PosDimData.PathXEndPosArray = [4125.0,4375.0,5375.0,4625.0,5625.0,4875.0,5125.0,5875.0]   # Arcs
PosDimData.PathYEndPosArray = [0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]


##############################################################################################################

# Z Direction finite measurement amount (micron)
# The layout of the program is that the chip is modelled as slices of x-y coordinates for each segment
# created in the Z direction.  This variable specifies the precision (grain) to calculate with.  This
# will increase the number of slices created, increasing the precision, but also the computing time.

# For now - make this an even multiple of the block section length
PosDimData.ZSpatialPrecision = 25.0
PosDimData.NumberZSlices = (PosDimData.BlockSectionLength/PosDimData.ZSpatialPrecision)

# length precision is how close each segment lengths must be (microns)
PosDimData.SegmentLengthPrecision = 0.1

# Minimum Path Separation Distance
# This variable is an optional input parameter.  The program will do an optional check to see that
# the created paths are separated from each other by this distance.  The program is optimised to
# maximize the separation distance, so if it is not specified (set to zero), it will assume it is fine.
```

```
PosDimData.MinimumPathSeparationDist = 30.0

# The calculated path distance is the length that each segment needs to have.  Usually determined
# by the primary segment.  If created as an array, the value can be returned from functions.

# The current Dragonfly has a 1 mm straight section at either end of the chip to maximize the light
inpuPosDimData.PathXEndPosArray = [4125.0,4375.0,5375.0,4625.0,5625.0,4875.0,5125.0,5875.0]   # Arcs
# into the chip.  These variables will specify how large of a straight section on either end, as well
# as a straight section in the middle to bridge the creation of a double spline section
# (in microns)
PosDimData.EdgeStraightSectionLength = 1000.0
PosDimData.CenterBridgeLength = 6000.0

# This is the 4 dimensional array that will hold the complete set of XYZ Data for all splines
# [X,Y,Z,Segment] - X,Y,Z = 3D coordinates, Segment = number of the segment
XYZSegmentDataArray = np.zeros((PosDimData.NumberOfSegments, PosDimData.NumberZSlices+1, 3), dtype=np.float64)
ArcRadiusDataArray = np.zeros(PosDimData.NumberOfSegments, dtype=np.float64)

############################################################################################################
# PROGRAM DESIGN
############################################################################################################

# To calculate the path, the program uses the spline function.  This takes the starting and end
# points of the path, then returns the centerpoints of the function for each z-slice.

############################################################################################################
# Main logic loops
############################################################################################################

# This is the main program that calls the sub functions and keeps track of the flow.
# Input/Output Coordinates
CalculateInputOutputDistance(PosDimData)

print PosDimData.XDistanceArray
print PosDimData.YDistanceArray
print PosDimData.DirectDistanceArray

PosDimData.PrimarySegmentNumber = DeterminePrimarySegments(PosDimData.DirectDistanceArray)

# ARC BASED CODE
CreatePrimaryArcSegment(PosDimData, XYZSegmentDataArray, ArcRadiusDataArray)
DetermineSecondarySegments(PosDimData)
CreateSecondaryArcSegments(PosDimData, XYZSegmentDataArray, ArcRadiusDataArray)

# SPLINE BASED CODE
##primaryOK = CreatePrimarySegments(PosDimData, XYZSegmentDataArray)
##DetermineSecondarySegments(PosDimData)
##secondaryOK = CreateSecondarySegments(PosDimData, XYZSegmentDataArray)

SegmentDistanceCurvatureCheck(XYZSegmentDataArray, PosDimData, WavePropData, ArcRadiusDataArray)

#Show the final 3d plot
PlotAllSegments(XYZSegmentDataArray)

# Create files if specified
CreateLaserWriteAndRSoftFiles(XYZSegmentDataArray)

print "*** END ***"
```