°Master of Science

CORGI: COMPUTE ORIENTED RECUMBENT GENERATION INFRASTRUCTURE

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

in Computer Science

by

Christopher Hunt

March 2017

COMMITTEE MEMBERSHIP

TITLE:	CORGI: Compute Oriented Recumbent Generation Infrastructure
AUTHOR:	Christopher Hunt
DATE SUBMITTED:	March 2017
COMMITTEE CHAIR:	Professor Christopher Lupo, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Alexander Dekhtyar, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Andrew Davol, Ph.D. Department of Mechanical Engineering

ABSTRACT

CORGI: Compute Oriented Recumbent Generation Infrastructure

Christopher Hunt

Creating a bicycle with a rideable geometry is more complicated than it may appear, with today's mainstay designs having evolved through years of iteration. This slow evolution coupled with the bicycle's intricate mechanical system has lead most builders to base their new geometries off of previous work rather than expand into new design spaces. This crutch can lead to slow bicycle iteration rates, often causing bicycles to all look about the same. To combat this, several bicycle design models have been created over the years, with each attempting to define a bicycle's handling characteristics given its physical geometry. However, these models often analyze a single bicycle at a time, and as such, using them in an iterative design process can be cumbersome. This work seeks to improve an existing model used by the Cal Poly Mechanical Engineering department such that it can be used in a proactive, iterative fashion (as opposed to the reactive, single-design paradigm that it currently supports). This is accomplished by expanding the model's inputs to include more bicycle components as well as differently sized riders. This augmented model is then incorporated into several search platforms ranging from a brute-force implementation to several variants using genetic algorithm concepts. These models allow the designer to specify a bicycle design search space as well as a set of riders upfront, from which the algorithms search out and find strong candidates designs to return to the user. This in turn reduces the overhead on the designer while also potentially discovering new bicycle designs which had not been considered previously viable. Finally, a front-end was created to make it easier for the user to access these algorithms and their results.

ACKNOWLEDGMENTS

Thanks to:

- My wife and family for supporting me through the project.
- My advisers for their valuable feedback along the way.
- My puppies for the thesis title inspiration.
- Bicycles for being such a cool topic to explore.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The predominate bicycle design of today, known as the "safety bike", is the result of over a century of incremental iteration. This trial and error approach to bicycle improvement was born out of the lack of scientific understanding of the bicycle as a mechanical system, and even today our knowledge of bicycle physics is still in its infancy compared to other mechanical control fields [31]. However, this is not to say that scientists and engineers have not been working on modeling the bicycle – in fact, over 200 individual models have been described in the literature as of 2007 [24]. Typically, these dynamic models take as input the physical characteristics of a bicycle/rider combination and output graphs of their stability over a range of speeds. By examining the output of these models, a designer can choose to modify certain bicycle parameters and rerun the simulation, eventually iterating to a final solution.

The notable part of this design loop is that it is focused on a single bike/rider configuration at a time. This is primarily due to the large influence that the rider's mass has on the overall bicycle's handling sensitivity, and as a result, a change in the rider can cause a significant change in the handling characteristics of the final bicycle. Because of this variability in design from rider to rider, many designers choose to stick closely to proven bicycle geometries in order to reduce the risk of building something that handles poorly for a given subset of enthusiasts.

However, the prototypical safety bicycle is not the only plausible design that exists for the bicycle builder. An orthogonal design, known as the recumbent bicycle (as

shown in Figure 1.1), provides similar handling to the safety bicycle but can often reach higher speeds due to its lower profile and improved aerodynamics.

Figure 1.1: An example recumbent bicycle (Cal Poly's Gemini frame)

These recumbent bicycles have their rider in a more reclined position and keep the pedals in front of the rider's hips, and as a result are forced to use more unconventional wheel sizes and odd frame geometry. Unfortunately, due to the limited body of knowledge regarding the design and handling characteristics of these recumbent bicycles, many designers still shy away from these geometries.

This work seeks to try and reduce the burden on the bicycle designer by automating the search of an optimum bicycle geometry given a set of design parameter ranges and a set of riders. The intent is that the designer can specify an approximate range for each of the desired design parameters as well as a target bicycle handling curve and the model will return the optimum result in a reasonable amount of time. This work uses as its testbed the Patterson Control Model [28] to model the handling of each bicycle/rider combination. This model has been used for over 20 years by the Cal Poly Human Powered Vehicle (HPV) team to design their recumbent racing bicycles. The Patterson Control Model has 9 parameters as inputs, and as such, large searches of the design space can take a prohibitively long time to complete. To expedite the

search, this work seeks to optimize the process through the application of Genetic Algorithms (GA's). This paper contributes the following to the field:

- A modified version of the Patterson Control Model to ease user input and search space iteration.
- A brute force implementation of the bicycle optimization search.
- A study of the application of Genetic Algorithms over the bicycle design field, including experiments and conclusions regarding the optimum Genetic Algorithm operator values for:
  - Population Size
  - Generation Count
  - Selection Percentage
  - Cross Over Percentage
  - Mutation Percentage
- A genetic algorithm implementation of the bicycle optimization search.
- A partitioning algorithm to categorize/improve the genetic algorithm's results.
- A front-end for accessing the modified bicycle control model as well as the saerch platforms.

Myriad tests were conducted in an attempt to find the optimum values for each of the above mentioned genetic operators. These tests were conducted on two different bicycle design spaces: the safety bicycle design space and the recumbent bicycle design space. The results of these experiments were then analyzed and conclusions were drawn regarding the applicability of overall optimum genetic operator values for the whole of the Patterson Control Model design space. Additionally, the solution qualities and overall run-time performance of the brute force solution as well as the genetic implementations were compared. Ultimately it was found that the genetic implementations produced near optimal solutions in a fraction of the brute force algorithm's runtime, making them a viable and useful tool for future bicycle designers. Finally, a visualization component was created using Jupyter Notebooks to allow users to easily interface with both the Modified Patterson Control Model as well as the bicycle design search algorithms.

Chapter 2

Background

2.1 Bicycle History

The first recorded bicycle-like device was invented in Germany in 1817 and was dubbed the *running machine* or *hobby horse* because it consisted of two wheels and a seat, and required the rider to provide locomotion by pushing off the ground with their feet. Due to its cumbersome nature, this device gained little traction and faded out several years later. These running machines laid dormant for the following 50 years until a French tinkerer added a crank and pedals to the front wheel, creating a more practical machine which could be ridden for several miles. These new bicycles, known as *Ordinary's* or *Penny Farthings*, had higher speeds than their predecessors and could actually be used as a viable mode of transportation. However, this increased top speed came at the cost of rider safety; the Ordinary, being a single speed front-wheel-drive bicycle, required that the front wheel be incredibly large (by today's standards) in order to reach appreciable top speeds. This forced the rider to sit high up on the bicycle, making crashes (known as *headers*) catastrophic. It was not until the late 1870's that the modern *safety bike* was developed. This bicycle had the rider seated between two equally sized wheels and used a chain to drive the rear wheel. Since its inception, the safety bike has seen overwhelming acceptance by the general public and little has changed in its design over the past century [4].

Figure 2.1: Milestone designs in bicycle evolution

It is worth noting that the safety bike's persistence does not mean that it is the pinnacle of bicycle design. For instance, one appreciable problem with the safety bike is its sizable frontal area (due mainly to the rider's upright body), which causes a large amount of wind resistance and throttles the top speed of the bicycle. Another radically different bicycle design, known as the *recumbent* bicycle was born out of bicycle racing and places the rider low between the wheels with the crank in front of the front wheel. This lowered position coupled with the more reclined nature of the rider's seat greatly reduces the frontal area of the bicycle, allowing for its rider to reach higher top speeds. The effects of these designs were noticed immediately in the professional community during the 1932 racing season where amateur cyclist Charles Mochet out-rode the best professional riders primarily due to his recumbent being more aerodynamic than their safety bikes. Unfortunately, the governing body of bicycle racing, known as the Union Cycliste Internationale (UCI), thought the recumbent was unfair and banned it from the racing circuit in 1934 [5]. Many speculate that this banning is the predominant reason that recumbents are not a widespread bicycle design today and while recumbents have been making a comeback in other areas of bicycle racing and recreation, the number of recumbents being developed each year is still quite small, leaving most of that area of bicycle design unexplored.

2.2 Bicycle Models

2.2.1 Whipple Model

The design of safety bikes over the past century followed in the footsteps of the Ordinary with design guidelines and general rules being developed predominately through trial and error. However, there were some academics who tried to model the behavior of the bicycle using mathematics and engineering concepts. The first landmark bicycle model was introduced at the end of the 19th century by both mathematician Emmanuel Carvallo (1897) and Cambridge undergraduate Francis Whipple (1899) [18]. While Carvallo and Whipple developed their work independently, they both made similar assumptions which lead to similar models. However, Whipple's model had more detail due to its larger number of bicycle parameter inputs (25 in total) and is still used today by many to design bicycles [5]. Whipple's model idealizes a bicycle as 4 rigid bodies: the front fork assembly, the rear frame assembly and the front and rear wheels (as shown in Figure 2.2).

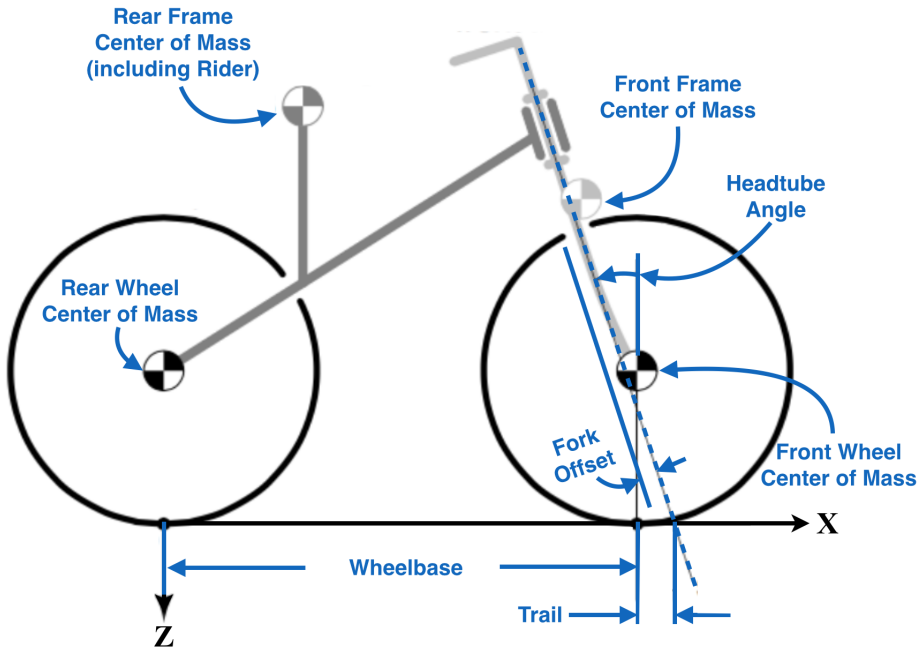


Figure 2.2: Whipple bicycle model

This model includes the rider as a single-point mass and attempts to characterize

a bicycle design's *stability* by determining if it is self-stabilizing without rider input. Another way of saying this is that a bicycle is considered *stable* if it will return to an upright position from a turn without the rider having to compensate (through handlebar input or body lean). While this model has been criticized for greatly simplifying the rider (and thus loosing fidelity), it has still been used to great success by the scientific and bicycle design communities. Examples include work done at UC Davis comparing instrumented bicycles to the model [22] as well as recumbent bicycle design performed by Delft University [5]. However, a good bike design does not necessarily have to be self-stabilizing because there is always a rider who can accommodate and correct for some inherent stability issues within the design (if those issues are not egregious).

2.2.2 Patterson Control Model

Other researchers agreed that the Whipple definition of stability was too strict and instead developed different models that focused on predicting how *rideable* a bicycle design was using various other metrics. One such model, known as the Patterson Control Model, was developed by Bill Patterson and the Mechanical Engineering Department at California Polytechnic State University, San Luis Obispo [28]. This model characterizes a bicycle's handling characteristics as a function of *Control Spring* and *Control Sensitivity*. Control Spring is an abstraction surrounding how likely it is that the front wheel of a bicycle will want to stay running in line with the motion of the bicycle. At low speeds, many bicycles with short amounts of trail (the distance between the fork and the center of the steered wheel) will experience *fork flop*, where the front wheel wants to over-rotate and flip around backwards. In the Patterson Control Model, a negative control spring signifies that the fork will never flip around on the rider (which adds to the bicycle design's stability). Control Sensitivity on the other hand, is an attempt to model the roll rate of a bicycle design due to a rider's input at the handlebars. An example of this is when a rider notices that they are falling to their left side. In order to compensate for this fall, they turn left into the fall which causes them to begin moving in a circle. This circular motion produces a centrifugal force on the bicycle and rider which pushes them upright and stops the

fall [21]. The rate of this roll is what is being modeled as Control Sensitivity. A good bicycle design can have a range of control sensitivities, and it is not uncommon for designers to want higher sensitivities at low speed ranges while still desiring lower sensitivities at high speeds to avoid a crash due to instability. The Patterson Control Model computes a Control Spring and Control Sensitivity value for a single bicycle speed, and as such, it is common to compute these values for a range of speeds and plot them as shown in Figure 2.3.

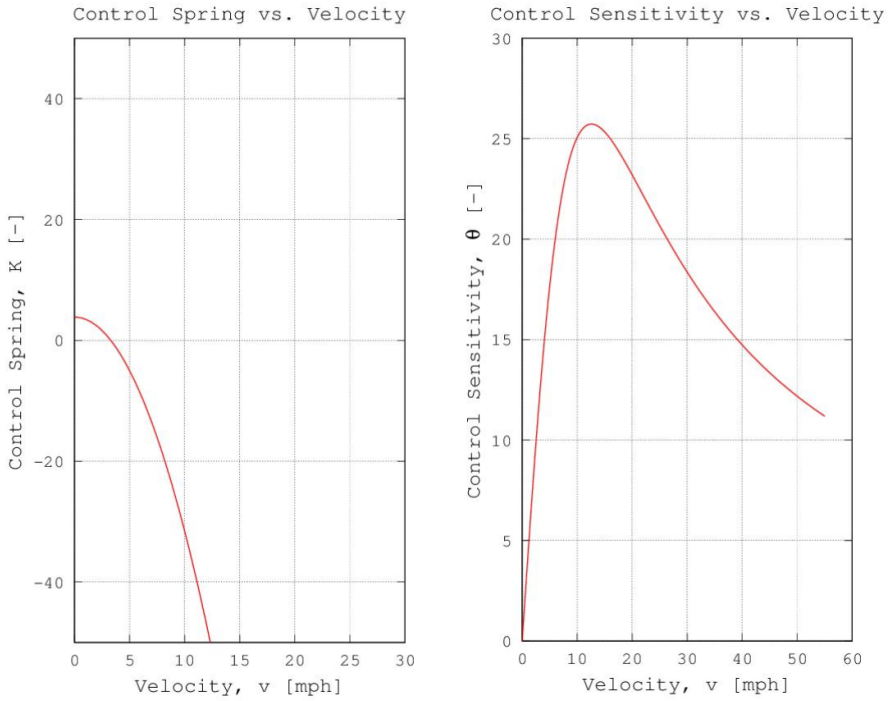


Figure 2.3: Example Patterson Control Model curves for the Cal Poly HPV team's 2012 Gemini frame

Of note, the Control Spring for this bicycle design starts positive before quickly becoming negative; as mentioned above this is a common feature of many bicycles and is not deemed detrimental as long as the value crosses zero under 5 miles per hour. Additionally, the Control Sensitivity of this design peaks at a value of 27 around 13 miles per hour and then gradually becomes lower as the bike's speed increases.

Cal Poly has a team of students who build a recumbent each year and race it against other colleges from around the world. This team, known as the Human Powered Vehicle (HPV) team, has used the Patterson Control Model for over a decade to help guide the design of their recumbent bicycles, and the curves in Figure 2.3

describe the team's 2012 Gemini frame. Interpreting what a *good sensitivity curve* looks like is generally up to the experience of the team both from a building and a riding perspective. For example, Bill Patterson noted that an average safety bike has a max sensitivity of around 15 but the Gemini frame tops out around 27, and some cyclists who were unfamiliar with riding recumbents found the Gemini frame to be somewhat unstable. However, the team also found that this increased sensitivity allowed for the bike to be more maneuverable during low speed cornering which gave them an edge while racing. This trade-off shows that the notion of Control Sensitivity is subjective in nature, but also that once a rider knows what sensitivities they are looking for, they can produce a design with specific handling properties over different speed ranges. In order to allow for quicker iteration, the Patterson Control Model requires only 9 separate parameters to characterize the bicycle as outlined in Table 2.1 and Figure 2.4.

Parameter	**Symbol**	**Units**
Wheelbase	A	meters
Fork Offset	e	meters
Headtube Angle	β	degrees
Center of Mass X-Offset	B	meters
Center of Mass Z-Offset	h	meters
Radius of Gyration	K_{xx}	meters
Handlebar Radius	R_h	meters
Front Wheel Radius	R_f	meters
Bike Mass	R_r	kilograms

Table 2.1: Patterson Control Model parameter inputs

The `wheelbase` is the distance between the front and rear wheels of the bicycle, and as the wheelbase decreases the sensitivity of the bicycle tends to increase. The `fork offset` is the perpendicular distance between the fork axis and the point where the fork connects to the wheel hub. This distance causes the wheel to intersect the ground behind the steering axis, and this distance is known as `trail`. Trail is also

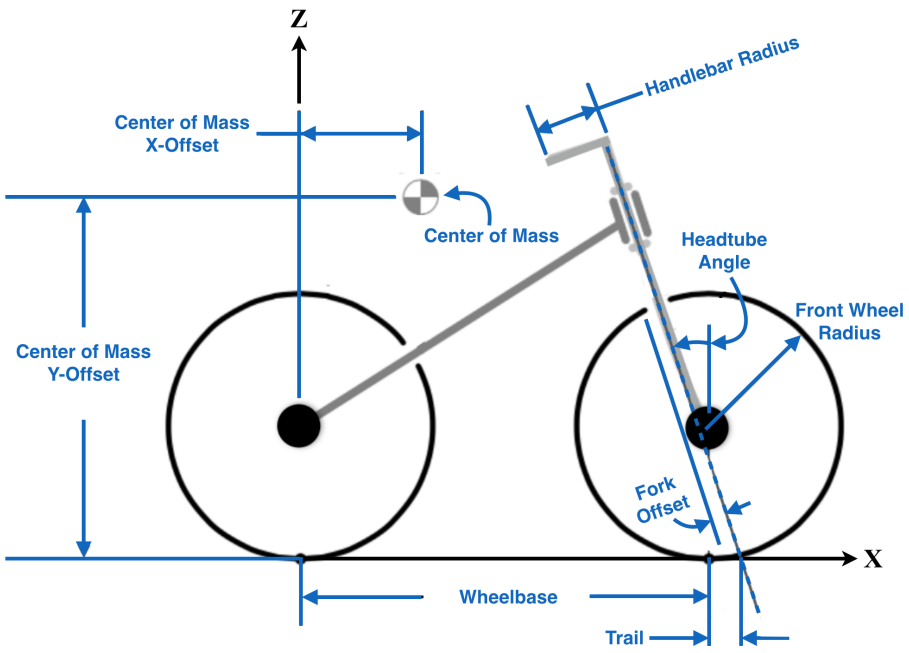


Figure 2.4: Diagram of Patterson Control Model parameter inputs

affected by the `Headtube Angle`, which is the angle of the steering axis relative to the vertical. Trail is considered crucial by many to the stability of a bicycle design [21]. The `Center of Mass X and Z-Offsets` are required to locate the center of gravity of the bicycle and rider, and is usually dominated by the rider's mass. Shifting the center of mass upwards or backwards will tend to lower the bicycle's roll rate. The magnitude and location of the bicycle's center of mass greatly influence the overall design's `Radius of Gyration`, which is a measure of the tendency of the body to rotate about a given axis. In the Patterson Control Model, this is the primary way to represent the rotational inertia of the system about the ground plane. A good analogy to this rotational inertia is the linear inertia that a simple block has. As the inertia of the block is increased, it becomes increasingly hard to start (or stop) the block. The Radius of Gyration is a similar measure of how hard it is to start (or stop) the bicycle and rider from rolling from side to side. The `Handlebar Radius` specifies the distance between the handlebars grip and the fork tube, and can be extended to represent normal handlebars (which stick out to the side) or tiller handlebars (which stick backwards towards the rider) as shown in Figure 2.5. Finally the `Front Wheel Radius` is simply the distance between the front wheel hub and the outside tire tread

and the `Bike Mass` is the combined mass of the bicycle and its rider.

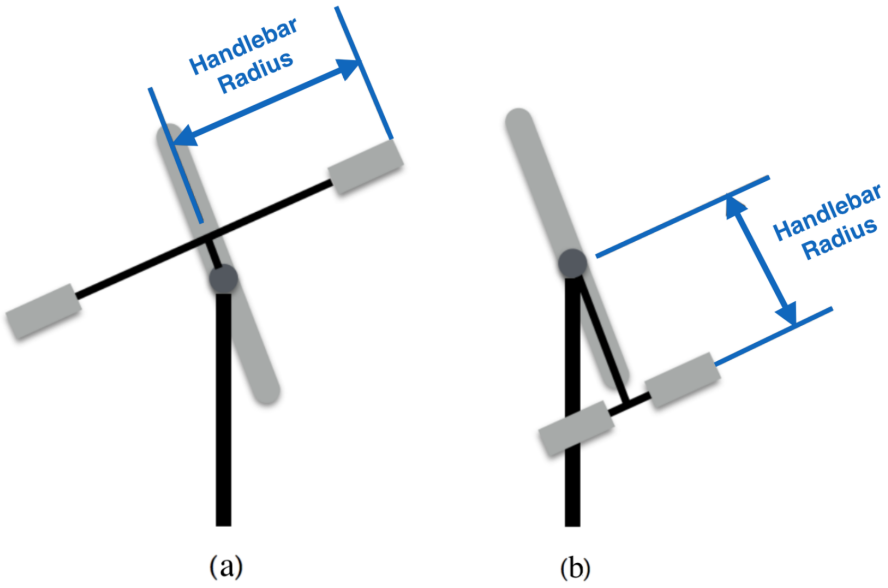


Figure 2.5: Handlebar types showing examples of a) regular handlebars and b) tiller handlebars.

Ultimately, the bike designer has these 9 parameters to tweak and modify in order to change the handling response of their final design. Also, it is important to note that many bikes are not made for the same sized person, and while the underlying bicycle geometry may remain fixed, the mass and location of the center of gravity of the bicycle will change from rider to rider. Thus, the designer must take into consideration several different riders when creating a frame in order to avoid making a bicycle which is only controllable by a subset of the target riders. Unfortunately, the base Patterson Control Model does not take into account the shape or physical dimensions of a bicycle's rider, and instead puts the burden on the designer to compute the appropriate center-of-mass and radius of gyration values for each bicycle/rider configuration (which can be time consuming and error prone). However, for designing a versatile bike that achieves the desired stability for multiple riders this is often necessary to avoid problematic bicycle handling corner cases. A good example of this is Cal Poly's 2007 Athena frame. As shown in Figure 2.6, while the sensitivity for the male rider peaks at around 27, the sensitivity for the female rider continues to climb until it reaches a value of 35 around 40 mph. The result of this was a bike that was essentially unrideable at speeds above 20 mph for the female

riders due to their lower radius of gyration values.

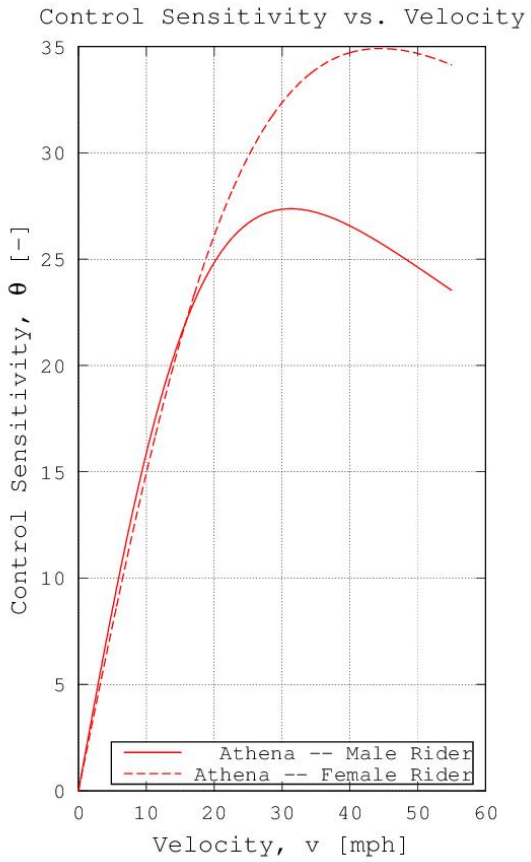


Figure 2.6: Control Sensitivity for the 2007 Cal Poly Athena frame

As such, for the Patterson Control Model to produce worthwhile results, the rider must be adequately modeled first, and given the complexities of modeling the human body this is a non-trivial task. Fortunately, several decades of work have been put into creating mathematical human body models which can be leveraged during this stage of the design process.

2.3 Human Body Models

The mass distribution of the human body is complex and difficult to analyze and model accurately, and as such there have been many studies attempting to characterize its features. These studies date back to the early 1860's with simple body

measurements, and have evolved into everything from measuring the human body through use of balancing apparatuses, photo-measurement, x-ray measurement and geometric modeling, among others [14]. Many initial measurements were conducted on cadavers in an attempt to isolate the inertial properties of each segment of the human body, and while expensive and inherently inaccurate, studies along these lines formed a large basis of body measurements moving forward since the early 1900's. Other researchers have attempted to use the data from previous studies to draw statistical conclusions and create mathematical models of the human body – however these models are often only for the 50th and 95th percentile population, and as such their applicability is still somewhat limited. Regardless of these limitations, these models have still been used to great effect in modeling humans in space-craft and car-crash simulations [23].

Since a bicycle rider usually makes up the majority of the bicycle system's mass, it is important to adequately model the rider's body in order to ensure that the effects of their mass and inertia are properly included in the system's handling analysis. As such, several contemporary mathematical body models were investigated including those developed by Yeadon [33], and Moore [25]. Yeadon modeled the human body as a set of 39 stadiums and a single ellipsoid, which required upwards of 95 measurements to characterize. While this high level of detail provided a more accurate mass distribution than prior models, Yeadon's design does have some shortcomings; specifically, it can have problems modeling limbs at the extremes of their travel and it often puts too much computational effort into calculating the smaller portions of the human body (such as the hands which contribute little to the overall inertia of the system). Regardless, Yeadon's model has seen a large amount of use in the modeling community, including being used in a study by Moore to model bicycle/rider interactions [19]. However, it is also noted that the model is somewhat unwieldy due to the number of components and that simpler models exists which can also model a rider in a usable fashion.

One such model was developed by Moore during his work with the Whipple Bicycle model. Moore's initial model represented the human body as a series of simple geometric shapes (10 in total), each with constant density as shown in Figure 2.7 [25]. This model required the measurement of a few key body dimensions, and as-

sumed that the mass distribution of a person followed the findings of Dempster who cataloged the mass of each body segment as a percentage of the person's overall mass [11]. A similar type of body model to Moore's was also used in [29] to model a human user transferring from one position to another. In [29], the authors showed that their body representation was roughly 95% accurate when compared to more complex solid models of the human body, and as such this form of model is likely suitable for many engineering applications. Additionally, the simplicity and flexibility of Moore's body model enables it to be paired well with the aforementioned Patterson Control Model to give a more holistic tool for designers to design bikes through. However, this tool could easily become unwieldy due to the large number of possible parameter combinations, and as such search algorithms would be needed to prune the design space for the designer. One such search concept, known as genetic algorithms, is adept at finding solutions to problems with high-dimensionality through iterative refinement as outlined in Section 2.4.

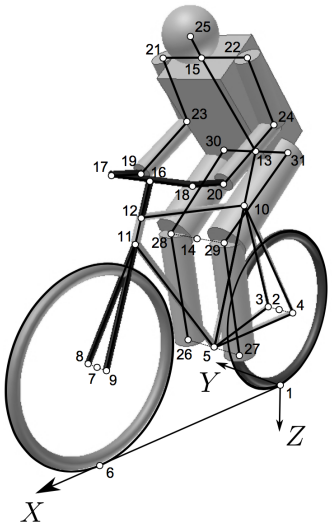


Figure 2.7: Moore's simple geometric representation of a bicycle rider [25].

2.4 Genetic Algorithms

Genetic Algorithms (GA's) are a family of search heuristics which try to discover near optimal designs by mimicking the process of evolution and natural selection.

The general belief is that these natural processes have allowed the organisms of this world to slowly iterate to near optimal configurations for their given environment (if left generally undisturbed), and that there is some implicit efficiency in the process of evolution which has been refined by nature over millennia [8]. Many problem spaces have been explored and optimized by GA techniques including the development of F1 cars [32] and the design of aircraft wing airfoils [30]. However, GA methods are not general purpose and cannot be applied to all problem domains.

In order for a GA approach to be applied to a problem, that problem must be able to be modeled in the context of evolution. The core element of any GA is the concept of an `individual`, which is a single solution to the problem being solved. These individuals are then grouped into `populations`, where each individual is usually created in a stochastic way. These populations are meant to mirror populations of organisms in the real world, with each solution in the GA population being akin to a given organism in the real world population. The GA approach extends this metaphor further by requiring that each individual be composed of discrete parameters (known as `genes`, `chromosomes`, or `attributes`) which can be manipulated to produce new solutions (known as `offspring`). The manipulation of an individual's genes can be done in a variety of ways, with each way being called a `genetic operator` or `operator`. These fundamental operators include:

- Selection
- Cross-Over
- Mutation

with each working in concert to try and mimic real-world evolutionary practices. Once specified, the set of genetic operators are applied to a given population of fixed sized over a series of generations, with each generation ideally improving based on the findings of those that came before it. Ultimately, genetic algorithms reduce down to being optimization problems with the goal of finding the best configuration of selection, cross-over and mutation rates combined with the appropriate size population and number of generations to uncover an individual solution that maximizes or minimizes whatever goal the researcher is focusing on.

2.4.1 Selection

`Selection` is the process of determining which individuals to combine (known as `breeding`) to create the next generation's offspring. This can be done in a variety of ways including choosing a random set of individuals from the population (known as *roulette* selection) to selecting the "top" individuals by percentage (known as *elitism* selection). The driving idea behind this is that the offspring of the best individuals will have a good chance of being better than their parents. However, not all offspring must come from the previous generation's breeding; some have found it beneficial to include randomly created individuals to keep some variety in the population. The reason this variety is important is because it can help prevent the GA from converging to a local optimum individual [27].

2.4.2 Cross-Over

`Cross-Over` plays a crucial role in determining how to combine two parent individuals to produce a child. The term comes from the field of genetics where individual genes would be combined from each parent to produce the resulting child. In general, the parameters that define each parent are arranged in an array. In *single-point cross-over*, an index is chosen in the array and the child receives all genes to the left of that index from parent A and the rest of its genes from parent B. In *two-point cross-over*, a second index is chosen and parent A supplies the genes that do not fall within the two indexes (while parent B supplies the remainder). This multi-point cross-over can be expanded up to *n-point cross-over*, where each gene can come from either parent. The determination of where to put these indexes can be done in a variety of ways, starting at defining the index in a pseudo-random way all the way to defining the indexes using heuristics which change as the GA runs [17].

2.4.3 Mutation

`Mutation` is the final primary operator at the GA's disposal, and it is responsible for randomly altering genes in a new offspring to be a value which might not

necessarily be that of either parent. In theory, mutation is the only operator which ensures that all possible parts of the design space can be iterated over because it can create individuals that have genes not present in the current population. This is important because it allows the GA to be generalized and gives the GA implementer the reassurance that their algorithm is not going to naturally exclude a portion of the design space which may eventually end up being optimal. However, mutation can be a tricky operator to tune: too little mutation and the algorithm will be slow to expand its search past the current local optimum but too much mutation and the algorithm devolves into random search [17].

2.4.4 Fitness

All of the above GA mechanics, from making a population of individuals to selecting a subset of them to produce the next generation, as well as the way all of the GA operators are combined hinge on a single final attribute of all GA models, and that is that there must be some way of comparing each individual in the population to rank them. In many cases, individual solutions are assigned a *score* via a `fitness function`, which is an objective function defined by the implementer. Fitness functions are usually heavily dependent on the problem space that is being analyzed and as such, good fitness functions often require the aid of domain experts to isolate the important metrics to optimize around. However, in general a good fitness function generates a wide enough spread in its rankings so there is enough resolution in the fitness values of each individual to allow the GA to make meaningful decisions from generation to generation. With all of these parameters defined, the GA will then try to find an individual with the maximal or minimal score (as specified by the designer).

2.5 Tuning Genetic Algorithms

The next step after configuring a problem to work within the bounds of the genetic algorithm paradigm is to specify the values of the various genetic operators, since selection of the appropriate values can vastly improve run-time performance and the

algorithm's final solution. Unfortunately, there is currently no set of genetic operator values that will work for all genetic algorithm applications. This is mainly due to the high variety of problem spaces that these algorithms can be applied to, with each having its own specific set of factors, responses and sensitivities. However, there are many different approaches to determining what a possible optimal parameter set could be, and these all revolve around exploring the responses of the genetic algorithm through *iterative tuning*.

Iterative tuning is a straightforward concept where a set of inputs is developed and then run through the genetic algorithm and the outputs are analyzed to see if any general trends occur. If trends are present, the tuner can then leverage them to create a more informed set of inputs to run through the genetic algorithm, thus honing its results. Ideally, this process will eventually converge to a set of parameter solutions which provide optimal (or near optimal) performance for the genetic algorithm over the specified problem space [12].

2.5.1 Types of Genetic Algorithm Tuners

There are a variety of tuners in the literature, each with its benefits and shortcomings, but in general they are all trying to find an optimal set of genetic algorithm parameters with as little computation as possible. The following tuners are some of the mainstays in use today by researchers:

Factorial Tuners

`Factorial Tuners` are the most basic and also likely the most informative of the tuner implementations. A factorial tuner determines what the optimal genetic operator vectors are by *exhaustively searching the design space*, and by trying each of the possible parameter combinations it allows trends to be analyzed without having to generalize or fill in gaps in the search space. However, the downside to this method is that it can be incredibly computationally expensive, especially if a large parameter space is to be investigated.

Sampling Tuners

`Sampling Tuners` try and reduce the total run-time required to explore the design space by breaking their search into two main components. The first component, known as the sampling stage, takes a uniform random sample of parameter combinations from the design space and runs each through the genetic algorithm to see how these combinations perform. Parameter selection can be performed through a variety of methods, with one of the most common being the use of the *Graeco-Latin Square*.

Graeco-Latin Squares are a way of combining a set of parameters such that no two combinations are the same. They derive their name from the way that they are commonly written, with one parameter space being described using letters from the Greek alphabet and the other parameter space being specified via letters from the Latin alphabet. For example, assume we had 4 parameter vectors we were looking to sample:

$$PopulationSize = [A, B, C, D, E]$$

$$GenerationCount = [I, II, III, IV, V]$$

$$SelectionPercentage = [a, b, c, d, e]$$

$$CrossOverPercentage = [\alpha, \beta, \gamma, \delta, \epsilon]$$

An example of the resulting Graeco-Latin Square is shown in Figure 2.8, with each cell being a combination of the parameters that is completely unique from the others in the set. Example configurations which would be sampled from this square include: $AIa\alpha$, $AIIB\beta$, $AIIIc\gamma$, $AIVd\delta$ and $AVe\epsilon$ (going down the A column). Additionally, this 5x5 matrix has 25 possible combinations which must be tested to sample the design space, a number which is vastly smaller than the $5^4 = 625$ possible combinations which would have to be tested if using a Factorial Tuner.

Graeco-Latin squares are useful for determining general trends of variables across a wide range of values, and for isolating the response of variables from one another. That is to say, the results of a Graeco-Latin square analysis by default do not take into account the general interactions of one variable with regards to another, but rather just sample the design space in a way that limits other factor's influences on the final results. While this may not seem an appropriate sampling method for use in

	A	B	C	D	E
I	aα	bδ	cβ	dϵ	eγ
II	bβ	cϵ	dγ	eα	aδ
III	cγ	dα	eδ	aβ	bϵ
IV	dδ	eβ	aϵ	bγ	cα
V	eϵ	aγ	bα	cδ	dβ

Figure 2.8: 4 Parameter Set Graeco-Latin Square

tuning a genetic algorithm due to the large amount of interaction between all of the genetic operators, studies have shown that they can provide an informative overview of the parameter space for some problem domains [26].

These results are then analyzed and the most promising of them are recorded. The design space is then reduced by a factor and hopefully begins to tighten around the most promising solutions. This process repeats until a stopping condition is met (either number of iterations or the space is no longer changing significantly), at which point a full factorial tuner is run over the remaining design spaces. While this factorial analysis can still be time-consuming, it is ideally run over a much smaller design space than what the algorithm began with, and as such tends to produce near optimal solutions in a much shorter time than a full factorial search would require (if the sampling methods honed in on the appropriate parameter ranges) [12]. Common examples of iterative sampling methods include *CALIBRA* [6] and the methods outlined in *Emperical Modeling of Genetic Algorithms* [26].

Model Based Tuners

`Model Based Tuners` try to extend the simplicity of sampling based tuning methods by creating a *model* of the genetic algorithm's *response landscape*. A response landscape is a multi-dimensional surface which shows how different parameters interact with one another and thus what trends exist in the data, an example of which is

shown in Figure 2.9. This example landscape shows that as *population_size* and *generation_count* are increased, the error in the final solution decreases (and visa versa). Additionally, it shows that while increasing either of the two factors independently will reduce the error to a degree, it appears that there exists a strong coupling between the two inputs with the error drastically reducing as both are increased together.

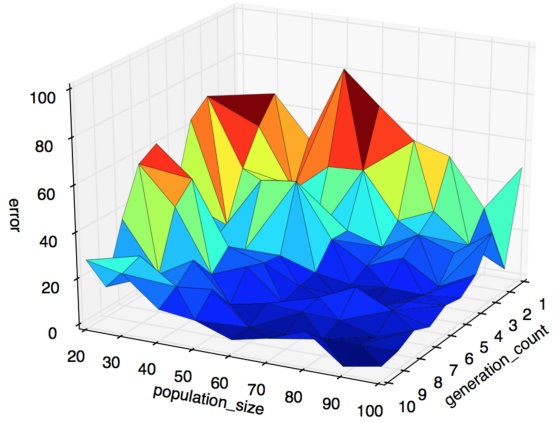


Figure 2.9: An example response landscape

Once a mathematical model of the genetic algorithm's parameter response space has been created from the initial samples, promising parameter subspaces are determined and then iteratively explored. This iterative exploration continues to sample the new subspaces, further refining the model until a stopping condition is met (number of iterations or the solution is no longer improving). Common examples of Model Based Tuners include *Coy's Procedure* [10].

2.6 Data Partitioning

Alongside developing tuning methods for genetic algorithms, many implementations also include a clustering/partitioning step in order to try and gain more insight into the trends each population might be exhibiting. This concept is used in data science to group data points together with the objective of creating collections of points with high *similarity*. This idea of *similarity* is not fixed, and many clustering algorithms use their own *similarity metrics* to judge how alike two data points are to one another. Additionally, the idea of clustering has many different implementations,

each with their own strengths and weaknesses. With that in mind, it is important to realize that there is not a single correct clustering scheme that applies to all problem domains, but rather there are myriad choices that exist and it is up to the researcher to pick the algorithm which best fits their data. As such, a general overview of the major clustering algorithms is presented below to provide perspective to the reader.

2.6.1 Centroid Based Clustering

`Centroid Based Clustering` is a widely used clustering concept that aims to partition a data set of `N` elements into `K` clusters such that the *mean squared distance* between each point and its *cluster center is minimized*. This process of finding the cluster configuration with the minimal mean squared distance is known to be NP-Hard, and as such only approximations of centroid based clustering are used in practice. The *K-Means* clustering algorithm is one of the more widely used implementations and requires the user to specify the number of clusters at run-time. Since the algorithm is only an approximation, the clusters it produces from run to run may not have the global minimum mean squared distance. As such, it is common to run several iterations of the algorithm (as time allows) with different data point orderings in order to try and find the optimum cluster configuration [15].

Due to the nature of their objective function, *K-Means* and its variants tend to produce clusters which are of roughly equal size and shape – properties which may or may not be indicative of the underlying data being analyzed. Additionally, the requirement that the user specify the number of clusters at run-time can be cumbersome in some situations (especially when the researcher is not familiar with the domain/data set). If the user can run several iterations of the algorithm with different cluster counts then it is often possible to determine the ideal number of clusters (knowledge which may be applicable to future data sets from the same domain), however, this process can become tedious and time-consuming, making centroid based clustering a poor choice for some problem domains.

2.6.2 Density Based Clustering

`Density Based Clustering` is a clustering variant which groups points that are closely packed together into clusters, with the idea that *point density* is the most important factor in point similarity. The most common implementation of density based clustering is known as `DBScan`. DBScan characterizes *point density* by analyzing the number of points within a specified radius from each point in the data set. If Point B falls within the circle inscribed by Point A's radius, then B is said to be *reachable by A*. If A can reach above a set threshold number of points, then A is said to be a *Core Point*. Core Points form the interior of each of the data set's clusters, and act as a backbone for their cluster, allowing it to expand and form amorphous shapes as needed. Points that are reachable by a Core Point but are not themselves Core Points (they can not reach above the minimum threshold number of points required) are known as *Edge* or *Fringe Points*. These points often lie on the outside edges of each cluster in the data set, and as such define the cluster's boundaries. Finally, if a point is not reachable by a core point then it is considered noise and is not included in any of the clusters in the data set [15].

In order to determine how to classify point density, the DBScan algorithm requires that the user specify the following parameters at run-time:

- The radius about a point in which other points are deemed reachable.
- The number of points needed to be reachable from a given point to consider that point a Core Point.

Due to the nature of the DBScan algorithm, the number of clusters is deterministic and will not change as long as the above parameters are kept constant. The only thing that may change is that fringe points may swap cluster membership if the order the data points are processed in is changed. While noteworthy, this caveat often is inconsequential, and in general the DBScan algorithm only needs to be run once to find the optimum cluster configuration (as opposed to multiple times for K-Means). Additionally, since DBScan works on a point by point basis, it can form clusters of arbitrary shapes based purely on the point density of the input data set, making it often more flexible than K-Means clustering. However, this flexible can also be a

disadvantage and can lead to odd results such as two seemingly disparate clusters being combined into one due to a single dense *point bridge* which connects the two groups of points. However, the robust and flexible nature of DBScan make it a good starting point for most researchers looking to classify their data.

2.6.3 Hierarchical Clustering

`Hierarchical Clustering` is a clustering mechanism that organizes all data points into a tree of clusters. This process can be completed in an *agglomerative manner* (starting with one element and building the tree from there) or a *divisive manner* (starting with all data points and splitting them recursively until the tree is formed). In either case, the resulting tree can consist of many different clusters, where a cluster may contain one or more points and one or more other clusters. As such, hierarchical clustering differs from the aforementioned clustering algorithms in that it provides a type of ordering to the final cluster set, where some clusters will be a natural subset of other clusters [15]. This information can then be used to determine things like lineage or family relationships between data points in the data set. However, for data sets where this type of information is not needed (or where the data does not fit this type of modeling), the previously mentioned clustering algorithms will likely produce better results.

2.6.4 ClusPro

`ClusPro` is a clustering algorithm used by biologists to try and identify the structure of proteins based on the density of their receptor-ligand structures. This algorithm works by searching the data set for structures with the highest density of neighbors within a fixed radius (as specified by the user at run-time). This element is considered the cluster center and all neighbors which fall within the specified radius are considered members of the new cluster. These elements are then removed from the data set and the process is repeated until a stopping condition is met (either all the points are clustered or a set number of clusters are formed) [20]. This method of clustering has been shown to be both simple and effective, with successful results in

several experiments such as the first Critical Assessment of PRedicted Interactions (CAPRI) experiment [9].

As noted by the authors of this algorithm, the most important factor in achieving meaningful clustering is the selection of the correct clustering radius. The authors present several domain specific methods for selecting the proper distance metric, with their results showing sensitivity to changes in 1 distance unit (in this example the unit is an Angstrom due to their problem space). Regardless, if there is adequate domain knowledge this can be a useful clustering technique to find *hot spots* of cluster centers that can be given priority over the rest of the data set.

2.7 Parallel-Coordinate Graphing

Once a search platform is implemented and tuned, it can still be difficult for the researcher to easily see the trends that their data may be exhibiting. This difficulty comes primarily from the high dimensionality of genetic algorithm implementations, and while data visualization comes in a variety of forms ranging from simple 2D scatter plots and curves to complex surfaces with coloring and shading, it can be quite difficult to capture all the variable interactions in a single plot. One option is to reduce the number of variables that are being depicted in a single plot, and instead produce multiple plots that cover the range of parameter interactions. While this is a viable option, it is often tedious and variable interactions can be lost in the process. In order to get around these issues, the concept of *Parallel-Coordinate Graphing* was created as early as 1880, and has since become a commonly employed technique for analyzing data with high-dimensionality [16]. This form of graphing consists of creating a vertical axis for each of the variables in the problem space. Then, each vector of parameters that makes up one-data point is drawn as a line through each of the vertical axes, with the intersection of the line and the axis being that data-point's value for that parameter. Figure 2.10 shows an example of a parallel-coordinate graph of some genetic algorithm data and highlights the performance of data-points with high *Cross-Over Percentage*, high *Mutation Percentage* and low *Selection Percentage*.

In general, parallel-coordinate graphs require some manipulation of the order and

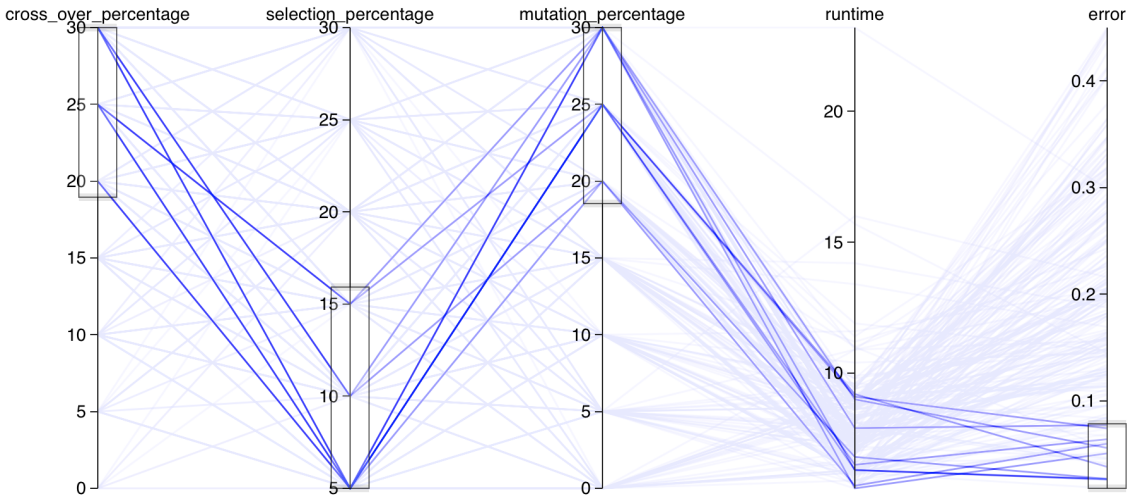


Figure 2.10: An example of a parallel-coordinate graph with multiple data points (each being described by a single line)

inversion of the axes to see some trends which may be hidden by all of the data points. As such, it is common for researchers to plot data in several different orders/orientations in order to gain more insight into their results. While somewhat time consuming, this still only requires tweaking a single graph (as opposed to many graphs), and ultimately allows the researcher to capture all of their desired interactions in a single, meaningful way.

Chapter 3

Design

3.1 Modified Patterson Control Model

The objective of this study is to use the Patterson Control Model (PCM) to create and analyze bike configurations in order to find a design that matches the desired handling characteristics for a specified set of riders. However, the base Patterson Control Model does not directly lend itself to iterative methods, mainly because many different real world factors are absorbed into a single variable within the model. For instance, one of the model's input variables is the *Center of Mass of the bike & rider in the X-direction* (which we specified as *Center of Mass X-Offset* in Section 2.2.2). The value of this variable is influenced by myriad factors including:

- The angle of the rider's seat
- The length of the rider's legs
- The mass of the rider
- The mass of the bicycle
- The center of mass of the bicycle
- The relative height of the seat and the cranks

with all of these being parameters that may need to be independently tweaked during the design process. This, combined with the need to be able to develop a bicycle alongside the rider who was to fit onto it (instead of designing a bicycle and

then attempting to fit a rider to it), meant that the Patterson Control Model needed to be modified to expose many useful parameters to the designer.

3.1.1 Model Modifications

The *Modified Patterson Control* Model (MPCM) contains upwards of twenty-four input parameters to the PCM's nine as outlined in Table 3.1 and as shown in Figure 3.1. Note that the purple circle denotes the bicycle's pedal circle, and the green lines specify the seat height and angle.

Parameter	**Symbol**	**Units**
Wheelbase	A	meters
Fork Offset	e	meters
Handlebar Radius	R_h	meters
Front Wheel Radius	R_f	meters
Rear Wheel Radius	R_r	meters
Crank Radius	C_r	meters
Crank X-Offset	C_x	meters
Crank Z-Offset	C_y	meters
Seat Height	H_z	meters
Hip Angle	α	degrees
Headtube Angle	β	degrees
Frame Mass	m_{frame}	kilograms
Crank Mass	m_{crank}	kilograms
Front Wheel Mass	m_{front}	kilograms
Rear Wheel Mass	m_{rear}	kilograms

Table 3.1: Modified Patterson Control Model parameter inputs.

Out of the nine original PCM parameters, the following five are retained un-

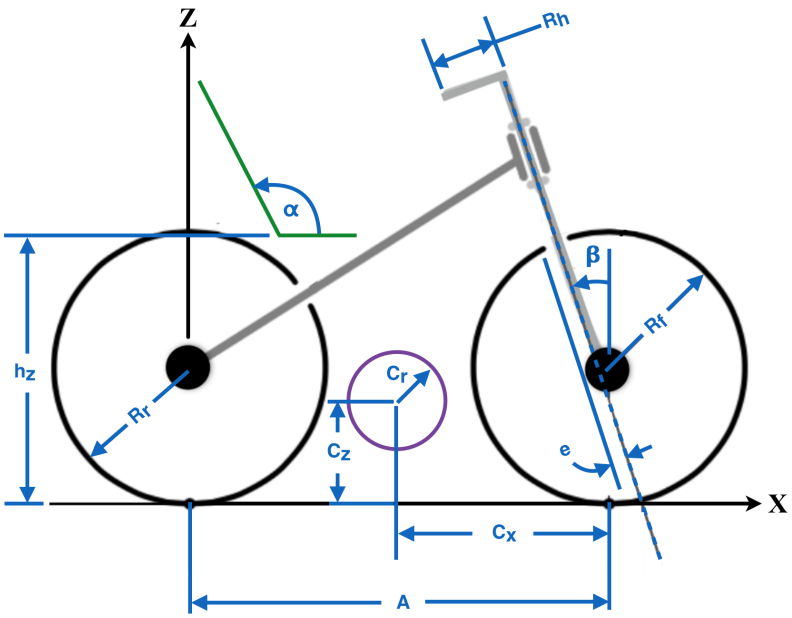


Figure 3.1: Modified Patterson Control Model parameter mappings.

changed:

- Wheelbase
- Headtube Angle
- Fork Offset
- Handlebar Radius
- Front Wheel Radius

The four other parameters in the PCM are then expanded in the MPCM as described in the following sections and as illustrated in Figure 3.2.

Mass Values and Location

In the PCM, the mass of the bicycle and rider are combined into a single sum. While this works for simple runs of the PCM, having all of the masses lumped together makes iterating through designs cumbersome. As such, the mass of the bicycle was broken into the masses of its components, namely: frame, crank, front wheel and rear wheel. Additionally, the mass of the rider was also specified as its own distinct value, making it easier to account for different riders in the iterative model.

The location of the center of mass of the entire assembly is still something that

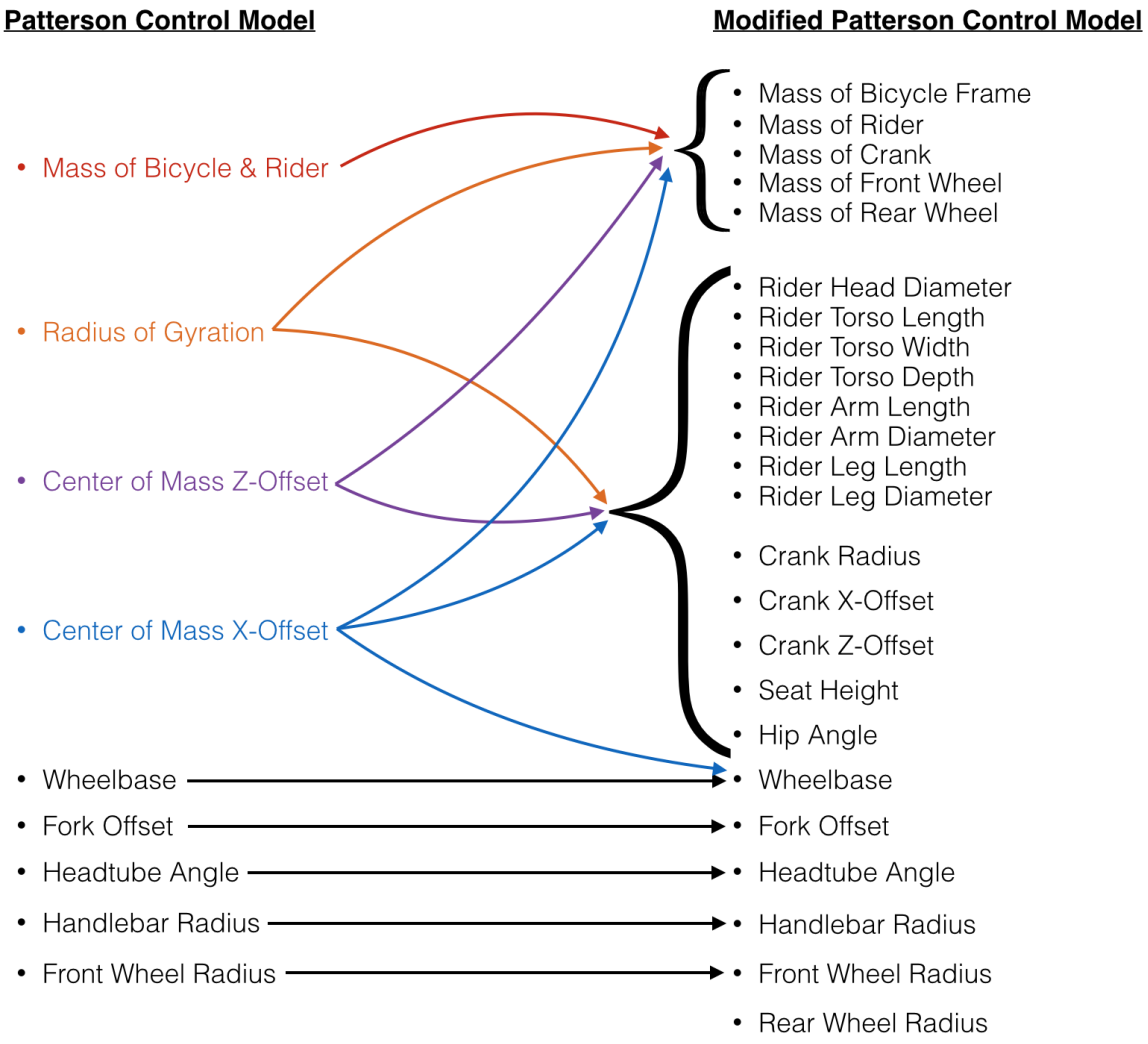


Figure 3.2: Dependency graph showing how the variables in the original Patterson Control Model map to the expanded parameter set in the Modified Patterson Control Model

must be computed in order to run the model, however, there are a large number of factors that influence the final location. While the frame makes up a meaningful percentage of the mass for a bicycle, in general the rider's body will be many times more massive than the frame. As such, the orientation of the rider which the bike is being modeled around will greatly effect the final handling of a bicycle design. Section 3.1.4 details how each rider is fit to their bike frame in a sequential manner and sheds light on why the center of mass computations depend on so many input variables.

Radius of Gyration

`Radius of Gyration` is a concept who's main goal is to give a numeric value to the mass composition of a body about a specific axis. In the case of the PCM and MPCM, the radius of gyration is taken about the axis that the bicycle wheels come into contact with the ground plane (denoted as the X-axis in Figure 2.4), with the fundamental radius of gyration being defined by Equation 3.1:

$$K_{xx} = \sqrt{\frac{I_{xx}}{m_{bicycle}}} \tag{3.1}$$

where K_{xx} denotes the radius of gyration about the X-axis, I_{xx} denotes the assembly's *mass moment of inertia* about the X-axis and $m_{bicycle}$ denotes the mass of the bicycle/rider assembly. Interestingly enough, by dividing the inertia of the system by its mass, the resulting radius of gyration is a massless quantity. However, changes in the system's mass distribution can still greatly effect its final value. This can be seen by looking more closely at Equation 3.2 which shows the definition of mass moment of inertia:

$$I_{xx} = \int_Q r^2 dm \tag{3.2}$$

where r is the perpendicular distance from each piece of mass Q to the specified axis x and dm is each piece's infinitesimal amount of mass. As such, changes to the geometry of the bicycle or rider will alter the distance value of particles within the integral, leading to a change in the mass moment of inertia, and ultimately in the resulting radius of gyration. All of this combines to make the radius of gyration value one of the most complex and tightly integrated computations in the MPCM.

3.1.2 Modeling the Frame

A flexible frame design model was needed in order to ensure that the MPCM would produce reasonable bicycle models ranging from recumbent designs all the way to safety bike configurations. While there exists bikes which have organic geometry (such as the frame shown in Figure 1.1), in general, most bicycle frames are built

from a series of tubes that connect the key components of the vehicle together into a rigid body. These primary tubes and their general connection points are shown in Figure 3.3.

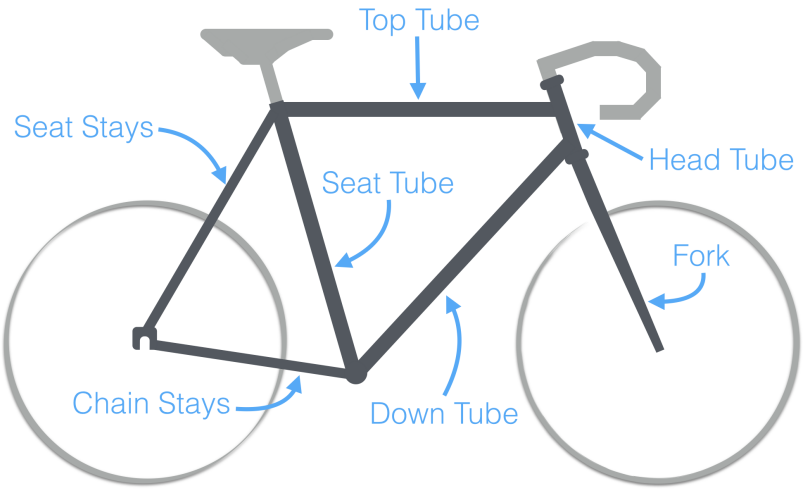


Figure 3.3: Commonplace bicycle tubes and connection points.

Since this is the primary configuration of most bicycle tubes, it was chosen as a template for use in computing frame geometry within the MPCM. The steps to computing the frame tube locations proceed as follows:

a) Compute the crank location from the given `X` and `Z-Crank Offset` inputs.

b) Compute the chain stays by connecting the rear wheel and the crank.

c) Compute the seat stays by connecting the rear wheel and the bottom corner of the seat.

d) Compute the fork/headtube by connecting the front wheel mounts with a line that exits the front wheel at the specified headtube angle. If the exit point is lower than the seat height, extend the tube until it is at the seat height.

e) Compute the top tube by connecting the seat to the top of the fork/headtube.

f) Compute the down tube by connecting the top of the fork/headtube with the crank.

This process is shown for both a recumbent design (Figure 3.4) as well as a safety bike design (Figure 3.5) with the wheels represented as black circles, the pedal circle denoted by a purple circle and the seat height specified by the blue dotted line. Note that in the safety bike frame building process, the headtube is extended upward to the seat height while in the recumbent design it is not.

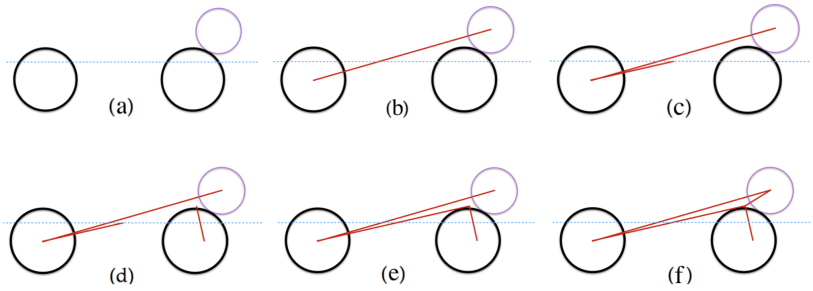


Figure 3.4: Steps for constructing a recumbent frame.

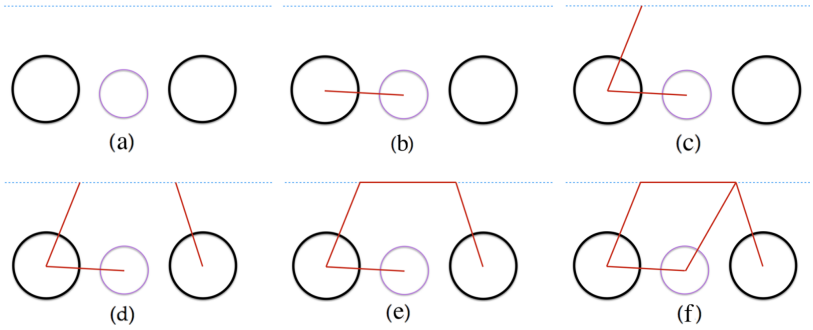


Figure 3.5: Steps for constructing a safety bike frame.

Assumptions

As with any model, assumptions must be taken into account to determine what scenarios it is applicable. This model makes the following major assumptions:

- All frame tubes have constant density.
- All frame tubes consist of solid lines with no radius.
- The seat tube can be omitted without drastically effecting the model's accuracy.

Assuming that the frame tubes have constant density and no radius effectively reduces the inertia they add to the bicycle frame computation. However, this inertia is often small due to the light weight of frame tubes, and as such omitting this factor does not hurt the overall inertia computation. Instead, the mass center of each tube is determined and the overall frame center of gravity is computed and then factored into the final inertia computation. Note that in this computation it is assumed that the frame consists of 2 seat stays, 2 chain stays and a 2 sided fork. Removing the singular seat tube from the frame computation was done because this tube does not help locate any of the other major components of the frame or rider, and since the model is only a rough approximation of an actual frame (and the frame accounts for a low percentage of the final bike's inertia), it was deemed unnecessary to add the extra computation. Overall the inertia added by the frame to the overall bicycle/rider assembly is often quite small, but it does scale as the frame increases in mass and/or size.

3.1.3 Modeling the Rider

After considering the mathematical body models listed in Section 2.3, the simpler representation used in Moore's initial work [25] was chosen as it provides sufficient accuracy while reducing modeling complexity.

Assumptions

The final rider body model adheres to the following assumptions:

- The human body can be decomposed into geometric primitives as outlined in Figure 3.6.
- The density of each body segment is constant throughout.
- The mass of each body segment is determined by the mass percentages outlined in Table 3.2 for 95th percentile men.
- The human body is symmetric about its Sagittal plane[1].

[1]The plane that bisects the human body into right and left halves.

Body Segment	Body Mass Percentage
Arm	6
Head	7
Leg	17
Torso	53

Table 3.2: Body Segment Mass as Percentage of Overall Body Mass for 95th Percentile Men [29]

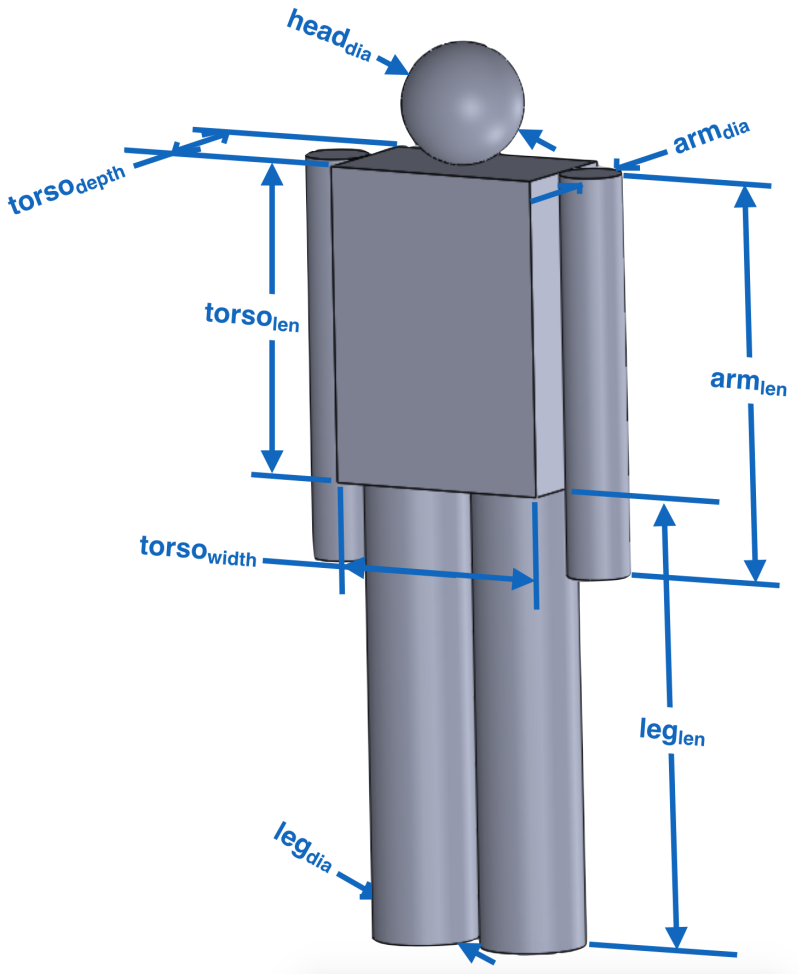


Figure 3.6: Human body model using geometric primitives.

This model represents the rider's head as a sphere, their torso as a rectangular prism and each of the rider's four limbs as cylinders of constant diameter. When comparing this model to those used in [25] and [29], the only major difference is

that this model represents the rider's legs and arms as single cylinders without a joint in the middle. This was done mainly to simplify the model, since having joints present adds extra complexity due to the multiple possible joint configurations that can exist for any given orientation. While this simplification does not allow for the rider's arms and legs to be bent in the most natural way, it is not uncommon for one (if not both) of the rider's limbs to be at their fullest reach in many bicycle riding configurations (such as sitting up with arms fully extended to the handlebars for example). Additionally, the limbs that would have the most effect on the overall inertia of the rider are the legs, and there are some configurations where at least one is fully extended while the other is not. Combine this generalization with how the PCM is modeling a dynamically moving rider (constantly pedalling and adjusting position), and it is likely that this model simplification has limited effect on the overall solution correctness. In a similar vein, treating the body segments as having constant density according to the recorded data in Table 3.2 should not hamper model accuracy as illustrated by both [25] and [29].

With this in mind, the dimensions of each of the body's components must be specified as inputs to the MPCM. These parameters are outlined in Table 3.3 along with their units. In general, it is advised to have the measurements be the average of the body segment being measured (so when measuring the arm diameter, take the average of the bicep and forearm diameters). Additionally, since the rider is assumed to be symmetric about their Sagittal plane, only a single arm and leg need to be specified (as their values will be reflected to the other limb as well). Once all of this data is input into the model, the rider is constructed and fit to the bicycle frame design that is being analyzed as outlined in Section 3.1.4.

3.1.4 Fitting a Rider to a Frame

Since most bicycles are designed to fit a range of riders, it was important for the MPCM to seamlessly accommodate fitting multiple riders to a common frame as well. In the majority of bicycles, rider fit is adjusted by changing the handlebar dimensions and the seat location. The process of *fitting* a rider to their bicycle is a time consuming and somewhat tedious task with lots of measurements and small

Parameter	Symbol	Units
Rider Mass	m_{rider}	kilograms
Rider Head Diameter	$head_{dia}$	meters
Rider Torso Length	$torso_{len}$	meters
Rider Torso Width	$torso_{width}$	meters
Rider Torso Depth	$torso_{depth}$	meters
Rider Arm Length	arm_{len}	meters
Rider Arm Diameter	arm_{dia}	meters
Rider Leg Length	leg_{len}	meters
Rider Leg Diameter	leg_{dia}	meters

Table 3.3: Rider Parameter Inputs

iterations, and as such when bikes are fit to a rider they are rarely altered moving forwards. However, this form of fit tends to work only when there is a one-to-one mapping of bicycle to rider, which is not always the case. For instance, the Cal Poly HPV team builds a single recumbent bicycle each year to race against other collegiate teams across the nation. Their bicycle needs to be able to accommodate a wide range of riders without having to go through the tedium of fitting each rider at race time. As such, many teams adopt a strategy of using a sliding seat, which allows multiple riders to be easily fit into the bike by moving a single component. This is the model that is adopted and simulated in the MPCM, and as such all riders are fit to bikes at a fixed *seat Z-offset*, but at a variable *seat X-offset*.

With that in mind, fitting a rider to a bicycle frame follows the following steps:

a) Create the general bicycle from the MPCM bicycle component inputs as described in Section 3.1.2.

b) Lock the rider's legs to the circumference inscribed by the crank pedals positioning their feet near the assumed down-stroke side of the pedal stroke.

c) Keeping the rider's legs constrained to the pedal circle, rotate the rider's body

until the bottom of their hip is at the specified seat Z-offset and the rider's body is behind the cranks.

d) Rotate the rider's upper body so that the specified rider hip angle is achieved and outstretch the rider's arms so that they are reaching towards the headtube of the bicycle frame (simulating reaching for the handlebars).

These steps are outlined visually in Figure 3.7, with the frame being depicted by red lines, the rider's body being drawn by blue lines, the wheels of the bicycle being represented as black circles and the pedal circle of the bicycle being shown as a purple circle.

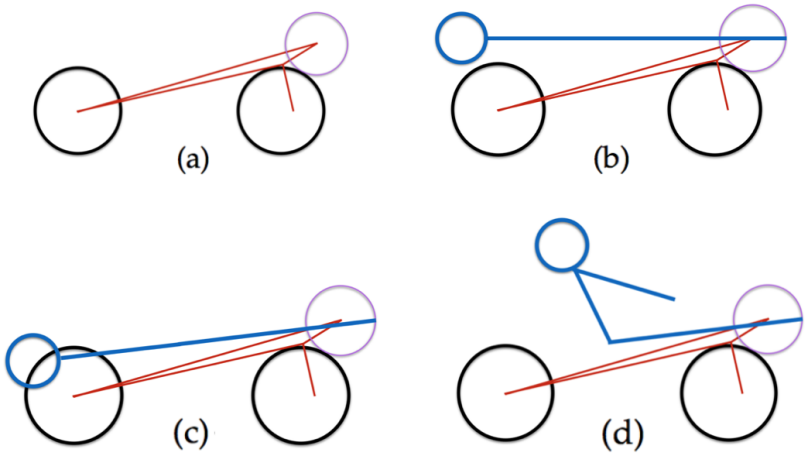


Figure 3.7: Visualization of the 4 steps of fitting a rider to a bicycle frame.

Additionally, the thickness of the rider's body parts are also taken into account when fitting the rider to the bicycle. This can be easily visualized if we include the rider's seat in the image as shown in Figure 3.8 which depicts the rider's center-line offset from their seat by half their torso and leg thickness.

3.1.5 Modeling Constraints

Since the MPCM is being developed to generate multitudes of bicycle designs on the fly without direct supervision, it is likely that many designs that it generates will not be physically feasible to ride. Even by enforcing that the rider can always reach the pedals from their seat, there are still numerous conditions which may result in a

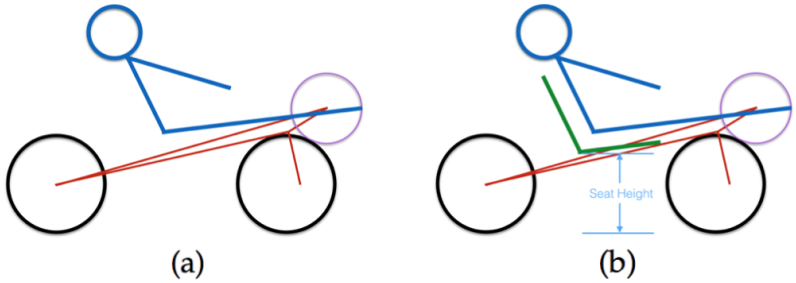


Figure 3.8: Figure a) shows the rider's center-line without the seat, and b) includes the seat to show that the rider's body thickness is taken into account by the model.

bike that a rider could not operate. As such, the following constraints checks were put in place:

- The rider's torso/head cannot be inside the front or rear wheels of the bicycle.
- The fork cannot be inverted (as in the fork must always point downward towards the ground).
- The bicycle's wheels cannot overlap.
- The rider's legs must be longer than the crank diameter (or else they could not pedal the bike).
- The crank center cannot be inside the front or rear wheels (crank overlap is allowed however).
- The seat must be above the floor plane.
- The crank pedals must not hit the ground.

As such, each bicycle/rider combination will be checked against this list, and if any constraint check fails then the design is given an error of infinity so that it will not be chosen by any algorithm/designer that is using the MPCM to generate bicycles.

3.2 Establishing a Fitness Function

In order to make use of the Modified Patterson Control Model in a bicycle design search algorithm, an idea of *fitness* needed to be established for each candidate design. Since the output of the MPCM is the same as the output of the PCM, namely a set of curves showing the bicycle design's *Control Sensitivity vs. Bike Speed* and *Control Spring vs. Bike Speed*, utilizing one or both of these outputs seemed sensible. While both are meaningful in their own right, in general a bicycle's handling characteristics are characterized more by its control sensitivity curve than its control spring curve. This is because the control spring graph only predicts at which point a bicycle's fork will not flop back around on the rider, and usually the curve's value drops below 0 under $2m/s$ (at which point the graph gives no more meaningful information regarding the design). On the other hand, the control sensitivity curve shows useful steering sensitivity data across the entirety of the bike speed space which ultimately makes it more useful as a predictor of bicycle fitness.

Thus, all of the following bicycle search algorithms accept as one of their inputs a *Target Control Sensitivity Curve*. This curve should be supplied by the user as a series of sensitivity data-points, each spaced apart from one another in $1m/s$ increments, and is meant to be the target handling curve for all bicycle designs to match. As such, the *error* of each bicycle design is taken to be the sum of the difference of squares of the design's control sensitivity values versus the corresponding target control sensitivity curve, as specified in Equation 3.3:

$$error = \sum_{1}^{n} (sensitivity_{target} - sensitivity_{canidate})^2 \tag{3.3}$$

where n represents the number of data points to test per curve, $sensitivity_{target}$ is the target control sensitivity value per speed increment, and $sensitivity_{canidate}$ is the sensitivity value per speed increment of the current design that is being analyzed. Note that in this design we are measuring the error of the resulting curve with respect to the target design's curve, with the design with the minimum error being the *best design* from the algorithm's perspective. Additionally, using the sum of the difference of squares of each data-point was chosen over other methods as it is both simple and

resilient to data that jumps from one side of the target curve to the other (something that the sum of differences is not resilient to). Finally, by squaring the error the resulting values have more range than they would otherwise have, making it easier for better solutions to stand out numerically.

3.3 Defining a Search Space

Along with the method of ranking bicycle designs that was discussed in Section 3.2, the search platforms in this work require the user to specify a search space for their bicycles, riders and in some cases genetic algorithm operators. Defining these ranges often requires the specification of many values for each input parameter. As such, a set of standardized configuration files was established (along with a parser) to make it simple for the user to specify their parameter ranges at run time. The types of configuration files are listed in the sections to follow, but it is worth noting that they all support a subset of the following three grammars:

- Option 1: Set the parameter `param` equal to a specific number `value`.

  ```
  param = value
  ```

- Option 2: Set the parameter `param` equal to a range of values between `x` & `y` stepping by `z`.

  ```
  param = x to y by z
  ```

- Option 3: Set the parameter `param` equal to the values `x`, `y`, `z`.

  ```
  param = [x, y, z]
  ```

Additionally, lines in the configuration file can also be comments or empty. Comments are specified using the # symbol, and are also supported inline. By default, each of the parameter setting options support integer and decimal values, and the range computation (Option 2) can step by either a positive or a negative non-zero constant. However, some parameters may have value restrictions, and each configuration file section specifies the range and value restrictions for each of their parameters. Furthermore, each configuration file requires that the proper parameter names

be specified by their exact name, otherwise they will be considered omitted. Once values are defined for each parameter, their collective values combine to define the parameter space that each search algorithm will explore in its own fashion. The following sections describe the different configuration files that exist for each of the search platforms, as well as the parameters required within each.

3.3.1 Bicycle Configuration Files

The bicycle configuration file's purpose is to specify each of the 14 parameters outlined in Figure 3.1. However, in order to ease parsing of the file, the parameter names listed in Table 3.4 must be used, and their assigned values must fall within the specified ranges. For an example bicycle configuration file, see Appendix A.1.

3.3.2 Rider Configuration Files

The rider configuration file allows the designer to specify the 9 body measurements outlined in Table 3.3 for each rider that is to be fit to a bicycle design. Note that since riders generally have a fixed set of dimensions, iteration over a set of parameters per rider is not a common use case. As a result, the rider configuration file only supports setting parameters to a single value as shown in *Option 1* in Section 3.3.

Table 3.5 shows the exact parameter names that must be specified in each rider configuration file in order for proper parsing, and Appendix A.2 shows an example rider configuration file in full detail. Additionally, if the designer would like to specify multiple riders for a bicycle design, they can do so by adding another block of rider parameters to the configuration file as shown in Appendix A.3. Note that in this case, each rider block is parsed based around where the `rider_name` parameter assignment begins, so its important to keep the rider parameter blocks separated as the Appendix example shows.

Parameter	Range	Units
wheelbase	$[0, \infty)$	meters
fork_offset	$(-\infty, \infty)$	meters
handlebar_radius	$[0, \infty)$	meters
front_wheel_radius	$[0, \infty)$	meters
rear_wheel_radius	$[0, \infty)$	meters
crank_radius	$[0, \infty)$	meters
crank_x_offset	$(-\infty, \infty)$	meters
crank_z_offset	$(0, \infty)$	meters
seat_height	$(0, \infty)$	meters
hip_angle	$[0, 360)$	degrees
headtube_angle	$[0, 360)$	degrees
frame_mass	$(0, \infty)$	kilograms
crank_mass	$(0, \infty)$	kilograms
front_wheel_mass	$(0, \infty)$	kilograms
rear_wheel_mass	$(0, \infty)$	kilograms

Table 3.4: Bicycle Configuration File Parameter Names and Value Ranges.

3.3.3 Genetic Algorithm Configuration Files

Similar to the bicycle and rider configuration files, the genetic algorithm configuration file specifies values for each of the genetic operators needed to run the search algorithm. As with the other configuration files, there is a strict set of parameter names that must be used as outlined in Table 3.6.

The *population_size*, *generation_count*, *selection_percentage*, *cross_over_percentage* and *mutation_percentage* variables all have a direct mapping to the genetic operators discussed in Section 2.4. Additionally, the *cross_over_gene_count* and *mutation_gene_count* variables specify the number of parameter values that will be swapped or mutated during their respective operation. So if the *mutation_gene_count* value was

Parameter	Range	Units
rider_name	–	string
rider_mass	$(0, \infty)$	kilograms
head_diameter	$(0, \infty)$	meters
torso_length	$(0, \infty)$	meters
torso_width	$(0, \infty)$	meters
torso_depth	$(0, \infty)$	meters
arm_length	$(0, \infty)$	meters
arm_diameter	$(0, \infty)$	meters
leg_length	$(0, \infty)$	meters
leg_diameter	$(0, \infty)$	meters

Table 3.5: Rider Configuration File Parameter Names and Value Ranges.

Parameter	Range	Type
population_size	$(1, \infty)$	Integer
generation_count	$(1, \infty)$	Integer
selection_percentage	$[0, 100]$	Decimal
cross_over_percentage	$[0, 100]$	Decimal
mutation_percentage	$[0, 100]$	Decimal
cross_over_gene_count	$(1, \infty)$	Integer
mutation_gene_count	$(1, \infty)$	Integer
num_runs	$(1, \infty)$	Integer

Table 3.6: Genetic Algorithm Configuration File Parameter Names and Value Ranges.

set to 3 and there are 14 bicycle parameters, 3 would randomly be mutated for each bicycle design that was picked for mutation (say *wheelbase*, *seat_height* and *hip_angle* for instance). Finally, the *num_runs* variable allows the user to specify how many

times each trial should be conducted, allowing them to attain more statistical certainty regarding their results.

An example genetic algorithm configuration file is shown in Appendix A.4. Note that each parameter is set to a single value – this is because these parameters define a single tuning configuration for the search algorithm, and in that setting multiple values do not make sense. If the designer would like to explore the genetic operator space they can do so by setting ranges for each of the genetic operators, with an example of this shown in Appendix D.

3.3.4 Partitioning Configuration Files

The final configuration file in this project is used to define the input parameters for the partitioning algorithm. The first parameter that the user must specify is how large of a radius should be used about the partition seed point when determining which designs belong to a given partition. Since this radius is fixed during the algorithm's run, the only viable entry for this parameter is a single decimal value. The second parameter that must be specified is the list of bicycle attributes listed in Table 3.4. These attributes are then used when computing the distance between two points in the design space (to determine whether or not to include a point in a partition). For an example partitioning configuration file, see Appendix A.5.

Parameter	**Range**	**Type**
radius	$(1, \infty)$	Integer
attributes	[1+ Patterson Parameters]	List of Strings

Table 3.7: Partitioning Configuration File Parameter Names and Value Ranges.

3.4 Brute Force Search Platform

The brute force search platform is designed to exhaustively explore the user specified bicycle parameter design space and return the configuration whose control sensi-

tivity curve best matches the one the user specified at run-time. As such, this search platform requires the following inputs in order to run:

- A bicycle parameter configuration file.
- A rider configuration file.
- A target control sensitivity curve to try and match.
- The number of top designs to return to the caller.
- A file name to write the results to.

Once the inputs are properly specified, the algorithm performs a full factorial search of the space by arranging each of the parameter ranges to be tested as a series of nested loops, ensuring that all combinations will be tried. Each bicycle design is initially given an error score of 0.0, at which point each rider specified by the user is fitted to the bike and the associated control sensitivity curve is generated. Each curve is then compared against the target control sensitivity curve using the fitness function outlined in Section 3.2, and the errors are summed and then averaged. This average error is then used as the bicycle design's *error value*, and it is this value that bikes are ranked by. The algorithm keeps a running list of the top N bicycle designs (where N is the number of designs to output as specified by the user) with the lowest average error value, continuously updating the list as better designs are discovered. Once the simulation is complete, the top N designs are output to a file of the user's choice as a JSON object. This object can then be parsed by an included `Bike Plotter` application to produce the following outputs:

- A list of the control sensitivity curve values for each rider.
- A diagram of the bicycle with all riders super-imposed over it.
- A graph of the datum control sensitivity curve as well as all of the rider's control sensitivity curves. This graph additionally includes the average error of the design across all of the riders.

3.5 Genetic Algorithm Search Platform

While the brute force search platform is guaranteed to generate the best set of solutions, the long run-time associated with exhaustively searching any sizable space makes it a cumbersome tool at best. The genetic algorithm search platform was created as a alternative solution that aims to leverage the strengths of evolutionary algorithms as a way to find close to optimal bicycle designs in a more reasonable time period. In addition to adding support for genetic algorithm operators, this platform was separated into two major versions: a standard genetic platform and one that also applies partitioning to each of its generational steps. Both of these platforms are described in detail below.

3.5.1 Unpartitioned Genetic Search

The goal of this search platform is to support all of the major genetic operators including:

1) Population Size

2) Generation Count

3) Selection Percentage

4) Cross-Over Percentage

5) Mutation Percentage

6) Cross-Over Gene Count

7) Mutation Gene Count

While the main definitions of many of these genetic operators were described in Section 2.4, the design decisions on how to enact each of these steps is worth noting. Specifically, *Elitism Selection* was chosen as the selection procedure for this algorithm, specifically because it ensures that the optimum bicycles that are generated won't be lost from generation to generation. Additionally, *N-Point Cross-Over* was chosen

because it kept the model flexible and freed its results from the order of *genes* (the different bicycle parameters) used to describe bicycle configurations. In both the cross-over and mutation cases, the number of genes to operate on is specified by the `Cross-Over Gene Count` and the `Mutation Gene Count` parameters, respectively. So in the case that the Mutation Gene Count was set to 8, 8 out of the 14 genes would be randomly mutated within the bounds of the design space.

The value ranges for each of the genetic operators are listed in detail in Table 3.5.1 with a notable restriction on the selection percentage, cross-over percentage and mutation percentage operators. While each of these three operators can technically have any value between 0% and 100%, it must be noted that the sum of the three parameter's values cannot exceed 100%. This is because the steps in the genetic algorithm sequence proceed as follows:

```
## Generate an initial population of random individuals.
new_pop = create_random_population(population_size)

## For each generation
for generation_count in range(0, max_generation_count):

  ## Update the old population to be the previous one.
  old_pop = new_pop
  new_pop.clear()

  ## Perform elitist selection.
  elitist_selection(old_pop, new_pop, selection_percentage)

  ## Randomly add individuals from the old population
  ## to the new population, leaving room for those to
  ## be generated via cross-over and mutation.
  add_random_individuals(old_pop, new_pop,
                         random_selection_percentage)

  ## Perform cross_over.
  cross_over(new_pop, cross_over_percentage,
             cross_over_gene_count)

  ## Perform mutation.
  mutate(new_pop, mutation_percentage,
         mutation_gene_count)
```

For each iteration, the population from the previous generation is set aside so it can be used to generate individuals in the new population. Then, the genetic operators are applied in the order of elitist selection, random addition, cross-over and finally mutate. Note that there is the seemingly extra step of *randomly adding individuals* from the old population to *fill out* the new population. This was done because there needs to be a way for the algorithm to ensure that the number of individuals from generation to generation is constant, and it is possible for the user to specify a set of percentages for the primary genetic operators (selection, cross-over and mutation) that would result in a generation that was smaller than its predecessor. Additionally, note that the elitist selection and random addition steps come before the cross-over and mutation steps – this was done to ensure that any individual that is chosen to persist to the next generation has a chance to interact with the cross-over and mutation operators, giving more diversity to the resulting population. If the order had been changed and individuals were randomly chosen from the old population to fill out the population after the other genetic operators had been run, the effect would be the same as not adding the individuals at all. This is because these individuals are by default ranked lower than those chosen by the elitist selection step, and as such will not be chosen to interact with cross-over or mutation steps in future generations. This would effectively result in a shrinking population because a large portion of the designs would never influence the progress of the group as a whole, and would likely lead to poor performance. Finally, in the event that the genetic operators add up to more than 100%, the search platform does not even run and instead returns an error. This was done because if the operators were to be run in this configuration then the population size would continue to grow from generation to generation, resulting in an ever expanding search algorithm which could become hard to reason about.

Similar to that of the Brute Force Search design, this platform requires the user to specify the following inputs:

- A bicycle parameter configuration file.
- A rider configuration file.
- A genetic algorithm configuration file.
- A target control sensitivity curve to try and match.

- The number of top designs to return to the caller.
- A file name to write the results to.

The configuration files are as described in Section 3.3 and the target control sensitivity curve is the datum by which all bicycle designs will be ranked against one another using the fitness function described in Section 3.2. Once run, the search platform will output the top N bicycle designs (where N is specified by the user) as a single JSON object which can be visualized using a `Bike Plotter` application (similar to how the Brute Force solution's output is visualized).

3.5.2 Partitioned Genetic Search

One of the shortcomings of any genetic search algorithm is that there is a possibility for the computations to get fixated on a single, local optimum solution. This fixation can lead to poor results from run to run and degrade the user's overall experience. One way to combat this problem is to try and add a large amount of mutation to each run, however it is important that the user be careful because adding too much mutation can cause their algorithm to devolve into a random search. Another option is to try and survey the fitness landscape of each population that is created, and select individuals that look to be promising to persist into the next generation (and therefore allowing their design spaces to continue to be explored). Promising individuals can take on many forms, but a good general metric for selecting them is to pick those designs that form *fitness peaks* (or valleys depending on your fitness function) in the population's fitness landscape. One way of doing this is by using smart partitioning to create groups of individuals that surround each peak (or valley), and then using those partitions during the next selection process. This allows the selection routine to make more informed choices when deciding what designs to include in the next generation – for instance it could choose to favor new and upcoming designs over more established designs (those with more individuals in their partition or with a better fitness) in order to see if the new designs prosper, or it could choose to take the top N individuals from each partition in order to give each design the same percent chance to be combined with the other genetic operators moving forwards.

This partitioned genetic search platform is an extension of the base genetic search platform, and adds the concept of *partitioning* to each iteration of the original algorithm as described above. Specifically, this platform applies the same genetic operators to each generation of individuals in the same order as seen in the base implementation. However, once the operators have been run per generation, the resulting population is partitioned into groups using the R-Partitioning algorithm described in Section 3.7. This partitioning of the population creates groups of similar physical form, and the selection operator in the next iteration takes the top N number of designs from each partition (where top designs are those with the lowest error). This allows the algorithm to value designs with a good fitness value while also prioritizing diversity with the aim to support fledgling designs which may have been overshadowed by more prominent designs if partitioning was not added. Pseudo-code for the partitioned genetic search algorithm is shown below:

```
## Generate an initial population of random individuals.
new_pop = create_random_population(population_size)

## Create a base partition set.
partitions = [new_pop]

## For each generation
for generation_count in range(0, max_generation_count):
  ## Clear the population so it can accept new individuals.
  new_pop.clear()

  ## Perform elitist selection.
  elitist_selection(partitions, new_pop, selection_percentage)

  ## Randomly add individuals from the old population
  ## to the new population, leaving room for those to
  ## be generated via cross-over and mutation.
  add_random_individuals(partitions, new_pop,
                         random_selection_percentage)

  ## Perform cross_over.
  cross_over(new_pop, cross_over_percentage,
             cross_over_gene_count)

  ## Perform mutation.
  mutate(new_pop, mutation_percentage,
```

```
        mutation_gene_count)

    ## Perform partitioning.
    partitions = partition(new_pop, partition_config)
```

Note that the inputs to the partitioned genetic search platform include all the inputs to the unpartitioned genetic search platform with the addition of a *partition configuration file*. This file is needed in order to set the partitioning radius as well as to define which bicycle attributes to partition based on. These partitioning parameters are described in detail in Section 3.7, and the format of the partition configuration file is described in Section 3.3.4.

3.6 Genetic Algorithm Tuner Design

In order to determine the best genetic algorithm parameters for each bicycle design space, two separate tuners were created. The first completes a full factorial search over the design space in order to ensure that no parameter vector is left unexplored. While thorough, this method can be incredibly time consuming, so a more advanced tuner was also investigated in an attempt to find a tuning solution that would run in a much shorter time. After reviewing several of the aforementioned tuner implementations in Section 2.5, an iterative sampling design along the lines of what is described in [26] was chosen for the second tuner implementation due to its ability to cover a large design space in a small number of runs.

These two tuners can then be run over the same design space, with the full factorial version acting as a datum for the sampling tuner. The results of the sampling tuner can then be compared against this datum, with the idea that if the sampling tuner aligns with the datum results, then it can likely be used in future tuning situations across unexplored portions of the bicycle design space. The sampling tuner begins by generating a Graeco-Latin Square of the input data. In order to keep the design simple, the square generated by the tuner has the following attributes:

- It accepts between 3 and 5 independent factors for testing.
- Each factor must have 5 different values.

Thus, the generated squares will look like one of the three depicted in Figure I.1.

	A	B	C	D	E
I	a	b	c	d	e
II	b	c	d	e	a
III	c	d	e	a	b
IV	d	e	a	b	c
V	e	a	b	c	d

a)

	A	B	C	D	E
I	aα	bδ	cβ	dϵ	eγ
II	bβ	cϵ	dγ	eα	aδ
III	cγ	dα	eδ	aβ	bϵ
IV	dδ	eβ	aϵ	bγ	cα
V	eϵ	aγ	bα	cδ	dβ

b)

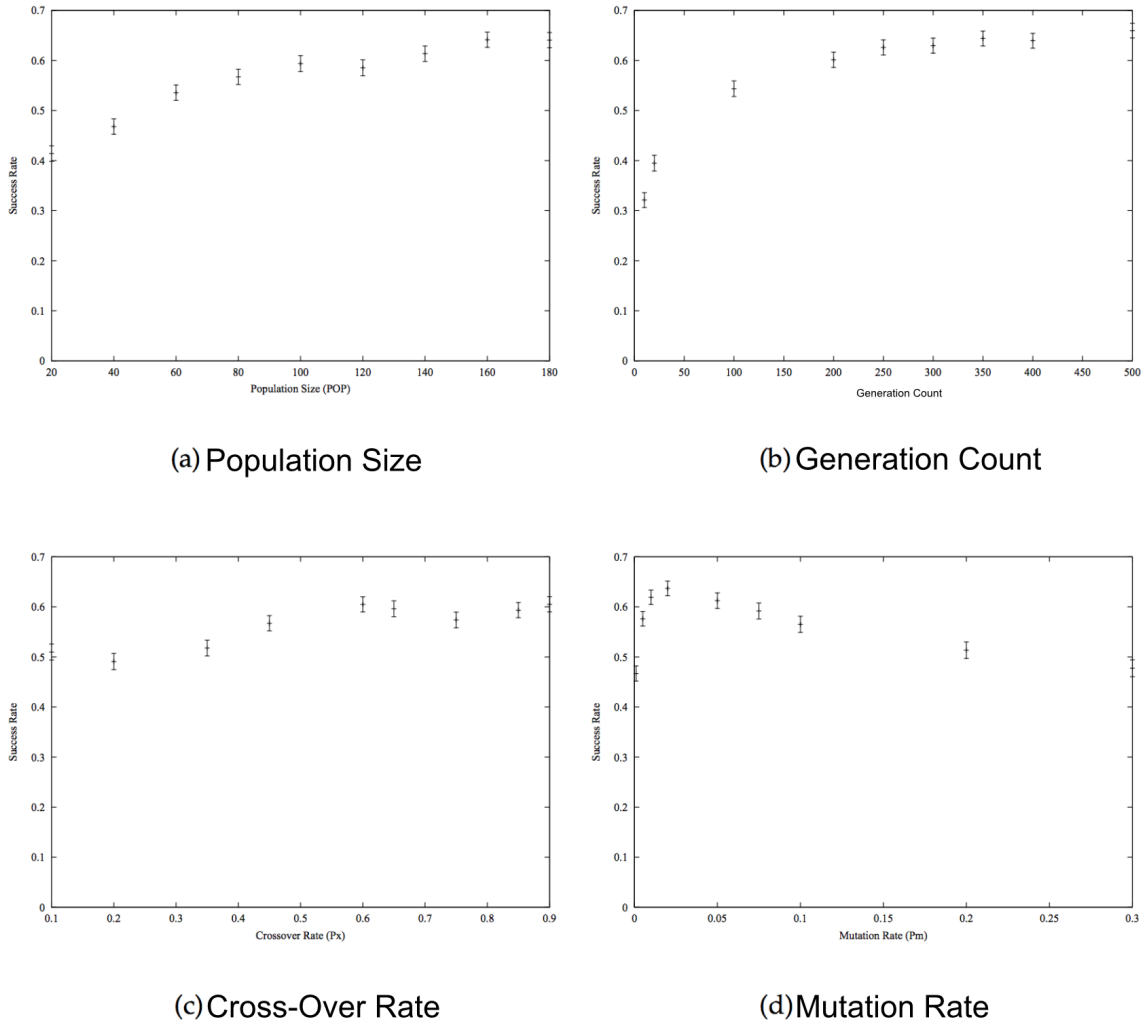
	A	B	C	D	E
I	aα1	bβ2	cγ3	dδ4	eϵ5
II	bγ4	cδ5	dϵ1	eα2	aβ3
III	cϵ2	dα3	eβ4	aγ5	bδ1
IV	dβ5	eγ1	aδ2	bϵ3	cα4
V	eδ3	aϵ4	bα5	cβ1	dγ2

c)

Figure 3.9: Depiction of the three different Graeco-Latin squares used in the sampling tuner. Example a) shows the square when 3 input parameters are present, b) for when 4 are present and c) for when 5 are present.

By convention, the two parameter vectors outside the square itself are usually set to be *Population Size* and *Generation Count*, respectively, with the interior parameters being *Selection Percentage*, *Cross-Over Percentage* and/or *Mutation Percentage*. Once the appropriate square has been generated, each of its cells are run through the specified genetic algorithm platform (partitioned or unpartitioned), and the results are collected and analyzed. At the end of the sampling stage, the fitness values for each of the tuning attributes are collected and independently graphed so that their trends can be made more apparent to the user. An example of such graphs is shown in Figure 3.10, with each plot giving insight into how its parameter influences the final output of the algorithm.

The second analysis stage is where the tuner reduces the design space for each parameter and continues to repeat its sampling until an appropriate stopping condition is met. Since the full factorial tuner will discover the optimum genetic operator vector on its own, it was chosen not to implement the second stage of the sampling tuner. Instead, the first stage of the tuner is run on the design space to see if it detects trends that reflect the findings in the full factorial search, which will help inform whether or not it is an applicable turning approach for this design space.



(a) Population Size (b) Generation Count

(c) Cross-Over Rate (d) Mutation Rate

Figure 3.10: Example output graphs from the sampling stage of a sampling tuner [26].

3.7 R-Partition

This section outlines the motivations and design considerations taken when developing the *R-Partition*[1] *algorithm*. Additionally, it focuses on examining how the R-Partition algorithm differs from other common clustering implementations cited in the literature, with the aim to help the user determine whether or not R-Partition is useful for their application.

[1]The algorithm is named R-Partition because it partitions data based on the rankings of points within the data set.

3.7.1 Why Partition At All?

After running some simulations with the base genetic algorithm implementation, it became clear that partitioning would be needed to make the process have more meaningful results. In this context, *meaningful results* has two main facets. The first is to try and ensure that for each stage of the genetic algorithm's iteration, that the key operators of *selection*, *cross-over* and *mutation* would be applied to the data set in a way that did not fully favor one grouping of individuals at the expense of others. For instance, consider the case where there are two different *genetic peaks* in a given population (two sets of highly different individuals that are scoring well), and one peak has slightly more individuals with slightly better scores than the other peak. Without partitioning, the slightly higher ranking peak was likely to be overrepresented in the next population while the other peak may not persist at all resulting in a possible loss of progress. With partitioning, the genetic operators can be applied to both peaks in a more intelligent way in order to not squelch fledgling designs which may prove to be more fruitful in later generations. The second main reason to partition is to ensure that the simulation can return a set of useful candidate designs back to the user. Without partitioning, it would be difficult to return a set of designs which were guaranteed to be significantly different to not be seen as near duplicates. With partitioning, we can easily return the highest ranked member of each partition and know that they are unique enough to be considered useful by the end user (since they specified the bicycle attributes to partition about).

After trying several different partitioning algorithms in the literature, it became apparent that this problem space needed a more domain specific algorithm. More specifically, an algorithm was needed which would form partitions of fixed radii around the genetic peaks in each population generated by the genetic algorithm simulation. By making each genetic peak be the seed of a partition, it would ensure that it was capturing all of the individuals that were producing the most promising designs, and by using a fixed radius it ensured that all of the individuals which were included in a partition were similar in physical form to the partition's seed (which is in contrast to clusters formed by DBScan which can be large and amorphous, resulting in partitions of data which may have large variations). Finally, a partitioning rank *threshold* was included as well in order to remove noise from the partitioning process (as low scoring

points likely should not be included in the resulting partitions).

3.7.2 R-Partition Design

The R-Partition design has three primary inputs that must be specified by the user:

- The `Partition Radius`
- The `Partition Attributes List`
- The `Partitioning Rank Threshold`

The partition radius is the maximum distance a point can be from a partition-seed point in order to be included in that seed's partition. The partition attributes list is a collection of bicycle genes (wheelbase, seat_height, etc.) that will be used when computing the *distance* from each candidate point to each partition's seed point. This distance is computed using the Euclidean distance formula, as specified in Equation 3.4:

$$distance(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{3.4}$$

where p and q represent the two points being considered, and the values 1 through n represent attribute values from each of the two designs. Note that this equation supports any number of attributes, and as such R-Partition allows distance computations to be as simple or complex as the user requires. Finally, the partitioning threshold is used to exclude points with poor rankings from being assigned to a partition, thus reducing the amount of noise in the final results.

The algorithm begins by excluding all points from the partitioning process that do not exceed the specified rank threshold and then sorts the remaining points by rank. Then, the highest ranking point in the data set is selected to be a new partition's seed point and is removed from the sorted list. From there, the distance from this seed point to each point in the sorted list is computed, and all points that fall within the specified partition radius are added to this new partition. Note that when a point is added to a partition, it is removed from the sorted partitioning data set. As such, the

R-Partition algorithm only allows a data point to belong to one partition at a time, and it will always belong to the highest ranked partition that it could be a member of. This process continues until all points in the sorted list have been assigned to a partition (even if that partition only contains the single point itself), after which the resulting partitions are returned to the caller in a list ordered by partition seed rank. Additionally, each point within the partition is ordered by rank as well. The resulting algorithm is outlined below in pseudo-code:

```
def partition(data_points, radius, threshold):
  ## Keep only points with scores better than the threshold.
  valid_points = remove_points_below_threshold(data_points, threshold)

  ## Rank the points.
  sorted_points = rank_points(valid_points)

  ## Create an empty list of partitions.
  partitions = []

  ## Loop until all points have been partitioned.
  while sorted_points:

    ## Get the seed individual for the next partition.
    partition_seed = sorted_points.pop()

    ## Form the new partition.
    new_partition = []

    ## Add the seed to the new partition.
    new_partition.append(partition_seed)

    ## Add any of the remaining points to this new partition if they are
    ## within the specified radius from the partition's seed.
    for point in sorted_points:

      ## Compute the distance from the partition seed to the point.
      distance_from_partition_seed = compute_distance(partition_seed, point)

      ## Only consider points which are within the partition radius
      ## to the partition seed.
      if distance_from_partition_seed <= radius:

        ## Add the point to the partition.
        new_partition.append(point)
```

```
            ## Remove the point from the list of possible points.
            sorted_points.remove(point)

        ## Add the new partition to the list of partitions
        partitions.append(new_partition)

    ## Return the final set of sorted partitions to the caller.
    return partitions
```

Assuming that the removal of points from the *sorted_points* is done in constant time, the resulting time complexity of this algorithm is $\mathcal{O}(n^2)$, where n is the number of data points in the input set. If the removal is instead done in linear time, the resulting time complexity is $\mathcal{O}(n^3)$.

3.7.3 Comparison to Other Algorithms

While this algorithm is fairly simplistic, it does share many similarities with other major algorithms that exist in the literature. Similar to K-Means and ClusPro, R-Partition compares all of the points in the data set to a single point in the candidate partition. However, unlike K-Means, R-Partition compares all points to the seed point in the partition which does not necessarily need to be partition's centroid. Also, the user must specify a comparison radius at run-time just like in DBScan and ClusPro. However, the R-Partition radius defines a maximum size of the partition based around the partition seed, where in DBScan the radius is applied to all points in the partition, allowing it to grow into large, unpredictable shapes. Finally, a point can only belong to one partition (as opposed to being a member of many partitions as in Hierarchical Clustering) and each run is fully deterministic (which is different than both DBScan and K-Means). These similarities are captured in Table 3.8, with checkmarks representing what attributes each algorithm has.

As outlined above, the R-Partition algorithm shares many common characteristics with the ClusPro algorithm. Both algorithms build partitions of fixed size which is determined by a preset radius value from the partition's seed. Both algorithms only allow a point to belong to a single partition and both versions have a sense of a threshold to stop partitioning at [20]. However, the main difference between

Table 3.8: Partitioning Algorithm Comparisons

Attribute	K-Means	DBScan	Hierarchical	ClusPro	R-Partition
Unique Partition Membership	✓	✓		✓	✓
Fixed # of Partitions	✓				
Fixed Partition Size				✓	✓
Deterministic		✓	✓	✓	✓

R-Partition and ClusPro is how these algorithms determine the partition seed for each iteration. In ClusPro, the partition seed is chosen as the element which has the highest density of neighbors within the specified radius (which is somewhat similar to DBScan). R-Partition instead determines the partition seed based solely on the *rankings* of each particle in the data set, and has no concept of density.

3.7.4 Algorithm Characteristics

This section outlines some noteworthy points for the researcher to consider when deciding on whether or not to use the R-Partition algorithm.

When is this algorithm applicable?

R-Partition is useful when you have data which can be *ranked*, and where each attribute has an even distribution of values. For instance, in the domain of bicycle design an attribute which may be used in partitioning is *wheelbase*. Bicycle wheelbases can have a variety of values which can be specified as a set as shown below:

$$wheelbase = [1, 1.1, 1.2, 1.3, 1.4, 1.5]$$

Note that there is a constant step value of 0.1 between each wheelbase entry. This is critical to the success of the R-Partition algorithm as it makes it easier for the radius computation to be reasoned about. If for instance, the wheelbase entries had the following values:

$$wheelbase = [1, 1.1, 1.5, 2.0, 3.0]$$

Then it is unlikely that the partition radius would be very effective as the spread in attribute values is sporadic. This is not to say that the user cannot attempt to partition about sporadic attribute sets, but if they choose to it is likely that they will end up with highly isolated partitions. That or they will be forced to use such large partition radii that they will end up with over-sized partitions that do not adequately segment their data in a useful way. In either case, the partition algorithm can still operate but it looses a lot of utility and usefulness.

When is this algorithm not applicable?

If you cannot *rank* all of your data points in a meaningful way then this algorithm will not function properly. Additionally, if you are interested in the resulting partition shapes or partition boundaries then R-Partition will likely not be useful in your study.

3.7.5 Extensibility

The above pseudo-code is intentionally simplified in order to make it easy to follow and because of this it leaves out several additions to the algorithm which give it more flexibility for future problem domains. One such consideration is that it currently only orders data points in a ranking from highest to lowest. This can easily be adapted to handle the opposite by passing a min/max variable as input and performing the appropriate comparisons within. Additionally, the *compute_distance* function is by default set to Euclidean distance, but it could also be expanded to support other distance calculations.

Chapter 4

Implementation

Implementation of the components of this project revolve around creating the Modified Patterson Control Model (MPCM) and building out the search algorithm platforms. All of these models were built and tested using `Python 3.6`, with many calculations in the MPCM using the `NumPy 1.11.3` package. Additionally, the plotting of bicycles was done using the `matplotlib 2.0.0` package. Python was chosen specifically for the great support it has with the aforementioned packages, as well as its ability to run within the `Jupyter Notebook 4.2.1` framework. Jupyter Notebook, an extension of iPython Notebook, is a web-based front-end which enables users to embed HTML and scripts into a simple document which other users can interact with. Due to its ease of use and general acceptance by the scientific community, all of the front-ends for this project were created using Jupyter.

4.1 Implementation Architecture

The high-level package implementation of this project is shown in Figure 4.1, with almost all of the packages utilizing the `Bike`, `Parser` and `SimulationParams` classes. The Bike class is the base unit that is created and manipulated by all of the search modules, and it is this class that handles fitting riders to bicycles and computing all of the components of the Patterson Control Model. Additionally, each

search module requires a set of inputs to specify the design spaces to search, and these spaces are defined using `configuration files` as specified in Section 3.3. It is the Parser's responsibility to convert these configuration files into dictionaries of $param \rightarrow [set\_of\_values]$ for the search platforms to ingest. Since the number of configuration files (and thus dictionaries) can become somewhat unwieldy with the more complex search platforms, the SimulationParams class was created to essentially be a container for all of this information.

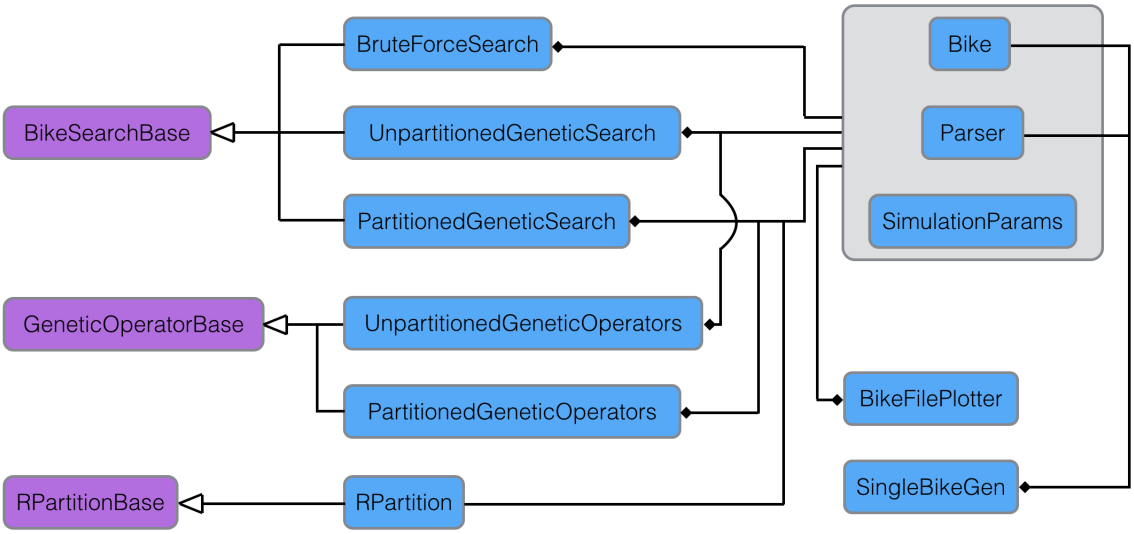


Figure 4.1: Overview of the project package architecture.

While each of the search algorithms attempts to find the optimum bicycle design in a different manner, they are all somewhat similar, and as such inherit from the same `BikeSearchBase` base class. The `Brute Force Search` implementation is the simplest of the group, and involves a series of 15 nested loops, one for each parameter range in the bicycle design space. Then, for each bicycle design, the set of input riders is fit to the bicycle in sequence and their handling curve is computed. The average of these handling curves is then used to *rank* each design with respect to the rest. From there, the top n designs are kept (in sorted order) and once the computation is complete, they are output for the caller to view. If we assume that there are n values for each of the 15 parameters, then this algorithm runs in roughly $\mathcal{O}(n^{15} \log m)$ (with the log due to the sorting operation required when inserting a high scoring run into the output list of length m).While this implementation is incredibly inefficient,

it does ensure that the entire search space is covered and acts as a good baseline for other implementations to compare against.

The two genetic search algorithms circumvent this massive runtime cost by stochastically sampling the design space. As with the brute force implementation, they require a set of configuration files to determine what the design spaces are. While the user will likely define a single value for each genetic operator, the implementations support running through multiple operator ranges (for added flexibility and future tuning if needed). These implementations then work by creating a population of individuals and then enacting the common genetic operators as outlined in Sections 3.5.1 and 3.5.2. These genetic operators function calls are defined in the `GeneticOperatorBase` interface, with an implementation for the unpartitioned and partitioned version being used in their respective search platforms. Additionally, the `RPartition` implementation inherits from a base interface and is used specifically in the partitioned genetic search implementation.

Defining the running time of the genetic algorithms requires investigation of the number of individuals that will be generated, with the main factors being the `population size`, `generation count` and `number of runs` (where the number of runs is how many times to repeat the experiment to obtain statistically relevant results). As such, if we define the population size as n, generation count as m and number of runs as p, then the overall running time of the unpartitioned implementation is roughly $\mathcal{O}(nmp)$. If we then extend this nomenclature to the partitioned implementation, we find that the running time is roughly $\mathcal{O}(n^4mp)$, where the extra n^3 comes from the partitioning operation (assuming the worse of the two implementations). All three search implementations output a JSON file of bicycle parameters which can be fed into the `BikeFilePlotter` in order to plot the results as shown in Figure 5.1. Additionally, since these search algorithms all fulfill a similar end-goal, they were wrapped into a single Jupyter Notebook as outlined in Section 4.2.2.

Finally, a utility class was created to allow users to generate the bicycle diagram and handling curves for a single bicycle design. The application, called `SingleBikeGen` is primarily for spot checking the handling of different bicycle/rider combinations, as well as allowing the user to generate handling curves for designs that they want to emulate (for use as datums in future search algorithm runs). This application was

created as a Jupyter Notebook as described in Section 4.2.1.

4.2 User Interface

In order to make the Modified Patterson Control Model (MPCM) more usable, a front-end needed to be created. After considering several front-end environments, the `Jupyter Notebook` platform was chosen primarily due to its simplistic setup, low resource requirements and general ease of use. Jupyter allows the developer to create *notebooks*, which are web-like forms which can have code embedded within them in containers called *cells*. This code can then be run in real-time by the user, allowing for a dynamic experience with a large amount of flexibility. Running code within cells can be done in a variety of ways, with the simplest being to select the cell and then press the *Run Cells, Select Below* button from the top toolbar, as shown in Figure 4.2.

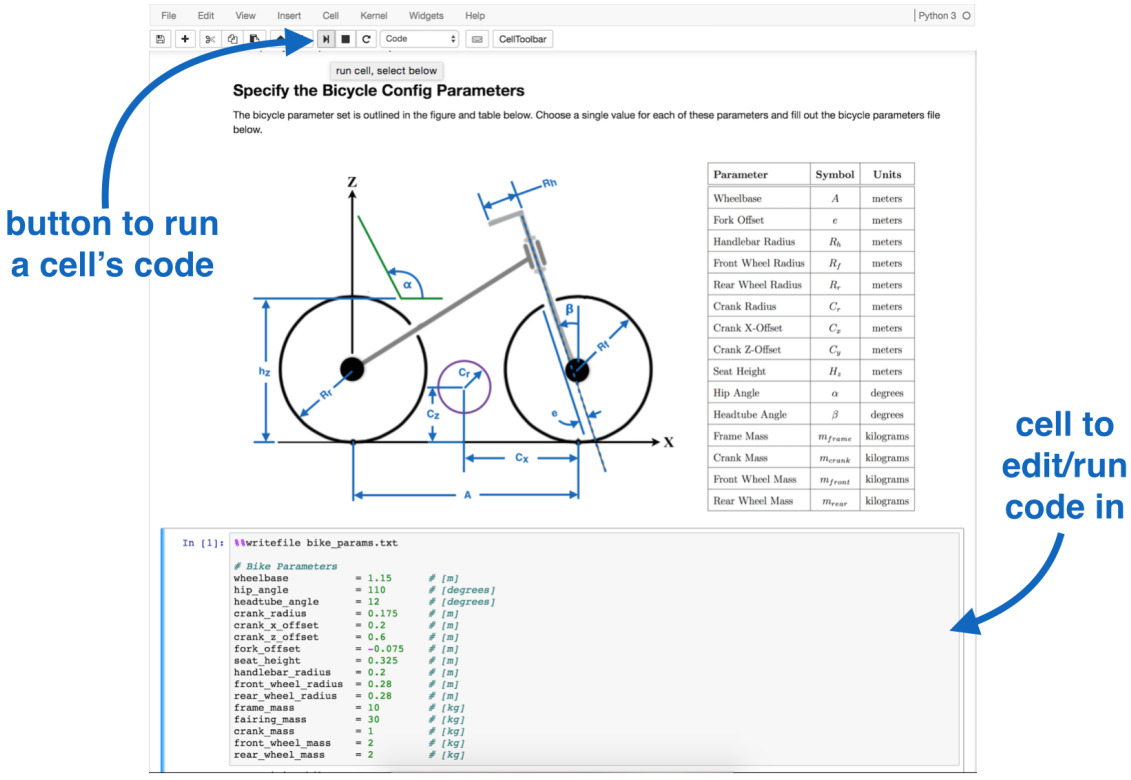


Figure 4.2: Example of the Jupyter interface and how to run code embedded in a notebook cell.

If the user would like to run all of the cells in sequential order, they can do so by selecting the *Cell* menu button, and then selecting the *Run All* drop-down. Additionally, the user can alter code within each of the cells as they see fit. In general, the only code that will be exposed to the user in these notebooks will be the contents of configuration files needed to run the MPCM. As such, it is encouraged for the user to alter these files so that they can tweak the MPCM to examine the bicycles and riders that they are interested in. Changing code in these cells is simple – doubling clicking on a cell allows the code to be edited in place, with hitting the escape key stopping the editing. Note that some cells will have lines of text which are proceeded by % symbols; these are notebook command line and should not be altered by the user. Once editing has been completed, the user can then run the code again as they had previously done to get their new output. For more details on how to use Jupyter see [2]. All of this comes together to create a simple yet powerful user interface that is more than adequate to run the MPCM. In order to logically breakup the different tasks a designer might want to perform with the MPCM, two different notebooks were created with each being elaborated on in the following sections.

4.2.1 Bike Plotter Notebook

The first thing that a designer will need to do when using the MPCM is to create a datum set of control sensitivity curves from which to compare against. This generally entails the designer having to back-calculate all of the Patterson Control Model parameters for past bike designs (if they do not have them already) prior to running the model and generating that bike's handling curves. This notebook provides all of the required functionality to allow the designer to generate a set of handling curves for a single bicycle parameter set. As such, it is broken down into the following sections:

1) Specify the bicycle's component parameters.

2) Specify each rider(s) parameters.

3) Specify the target control sensitivity curve values (1 per km/hr).

4) Run the plotter.

Step 1 has the designer specifying a *single value* for each of the MPCM's 15 bicycle parameters. Step 2 allows the designer to specify any number of riders to ride the bicycle (note that at least a single rider must be specified). Step 3 has the designer input a datum handling curve which will be used to compute the aggregate error between all of the rider's curves (as a way of showing how close the new design is to the desired design). Note that this curve should contain handling sensitivity values in 1 km/hr increments, with the number of points in the datum curve specifying the top speed the bicycle designs will be tested to. Thus if you specify 26 points (including the 0 km/hr point), then each bicycle/rider combination will be tested up to 25 km/hr. Once configured and run, the notebook will produce output which shows the following:

- The control sensitivity curve values (in 1 km/hr increments) for each rider.
- The body model parameters for each rider.
- A plot showing each rider superimposed onto the bicycle design.
- A plot showing each rider's control sensitivity curve as well as the datum curve.

with an example of this output shown in Figure 4.3. The user can then choose to save the file in any form they'd like using Juypter's $File \rightarrow DownloadAs$ menu selection.

4.2.2 Bike Search Notebook

Once the designer has their datum handling curves established, the next step is to search the bicycle design space for candidate designs that have similar control sensitivities. To do this, a *Bike Search Notebook* was created which allows the user to run any of the three search platforms (Brute Force, Unpartitioned Genetic and Partitioned Genetic). As such, the notebook has cells for each of the configuration files needed to run all of the search platforms, with the partitioned genetic search

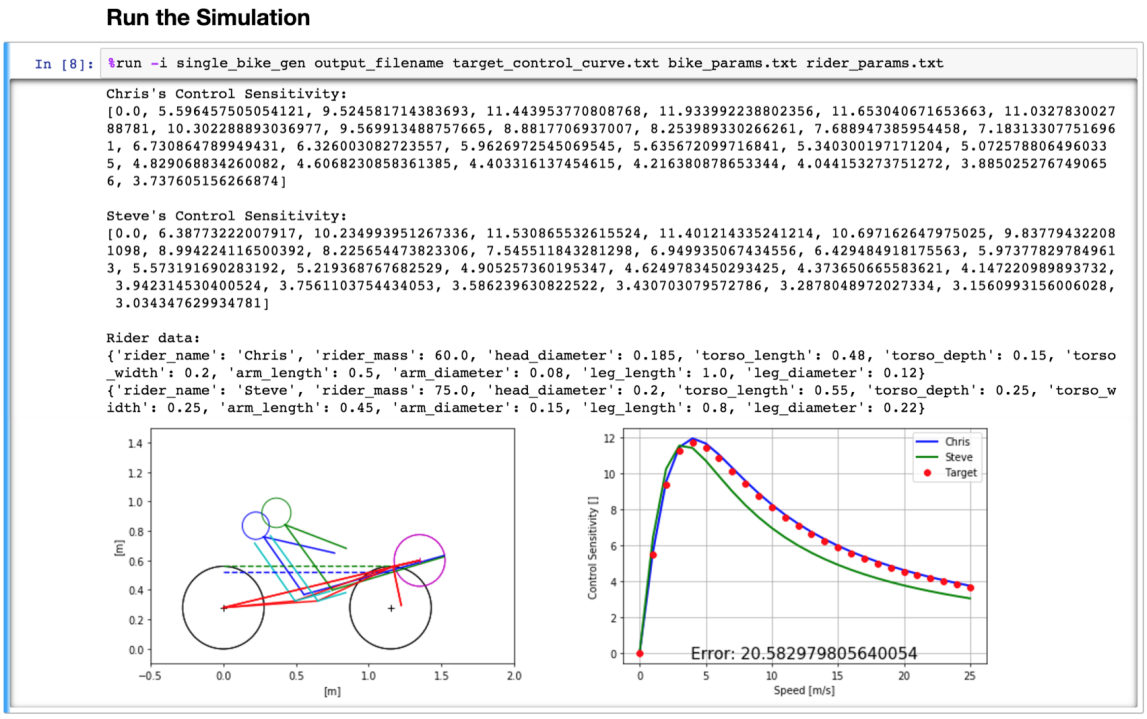


Figure 4.3: Example output from the Jupyter Bike Plotter notebook.

algorithm requiring the most inputs. However, the user only needs to fill out the configuration files required for the search algorithm they are running. Selection of the search algorithm is done by choosing one of the three options from the drop-down menu embedded in a cell near the bottom of the notebook as shown in Figure 4.4.

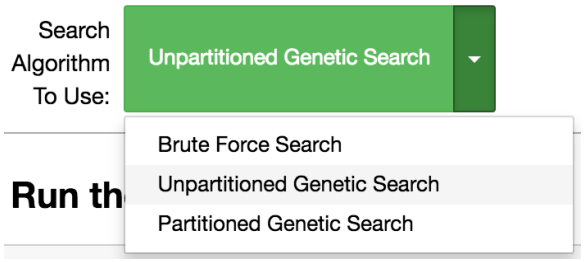


Figure 4.4: Button used to select between the three search algorithms.

From there, the appropriate search algorithm will be run with the results printed out below as shown in Figure 4.5. Note that if the Brute Force or Unpartitioned Search algorithms are run then the top 25 ranking designs will be output to the notebook cell. However, if the partitioned search algorith is run then the top ranking

design from each of the final partitions will be output to the notebook cell so that the user can see how their partitions ended up looking. As with the bike plotter notebook, the search results can be saved by using Juypter's *File* → *DownloadAs* menu selection.

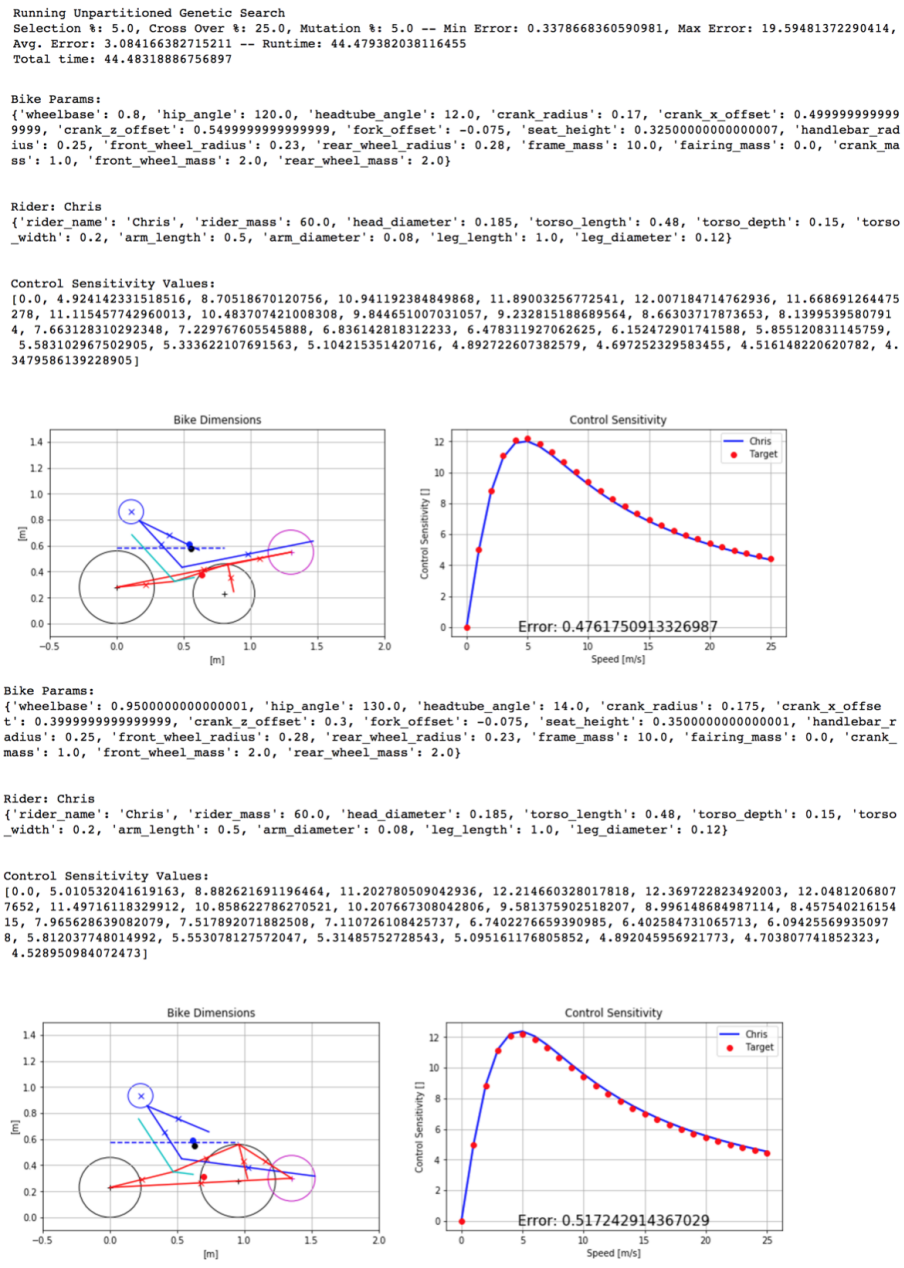


Figure 4.5: Example output from the Jupyter Bike Search notebook.

Chapter 5

Validation

5.1 Bicycle Model Validation

Validation of the bicycle model could take on several different forms. To begin, this thesis makes the assumption that the Patterson Control Model (which all of this work is based off of) is valid and correctly models real world bicycle handling. As such, vetting of the Patterson Control Model will not be part of this work's validation. However, it is imperative that the extensions to the Patterson Control Model accurately configure the bicycle rider in 3D space, and that the outputs of the model are the same as that of the Patterson Control Model (given equivalent input sets).

Validating that the rider and bicycle are correctly oriented in space relative to one another was done by creating a bicycle plotting program which would display a 2D representation of the bicycle and rider (an example of this can be seen in Figure 5.1). This plot shows the rider (blue) overlaid on the bicycle frame (red) with wheels (black) and cranks (purple). Additionally, this bicycle plotter made it easy to check that the center of gravity values for each of the bicycle frame and rider body components was accurately computed as well. In the figure, the centers of gravity of each of the rider's body components are represented by blue x's, the center of gravity of each frame tube is represented by a red x, the center of gravity of the rider is shown as a blue

dot, the center of gravity of the frame is shown as a red dot and the center of gravity of the bicycle + rider is shown as a black dot. Finally, the radius of gyration of the bicycle + rider with respect to the ground plane is shown as a dashed blue line. Then, by running this model across myriad different bicycle + rider configurations, it was straightforward to catch any edge cases and vet that the computations were correct.

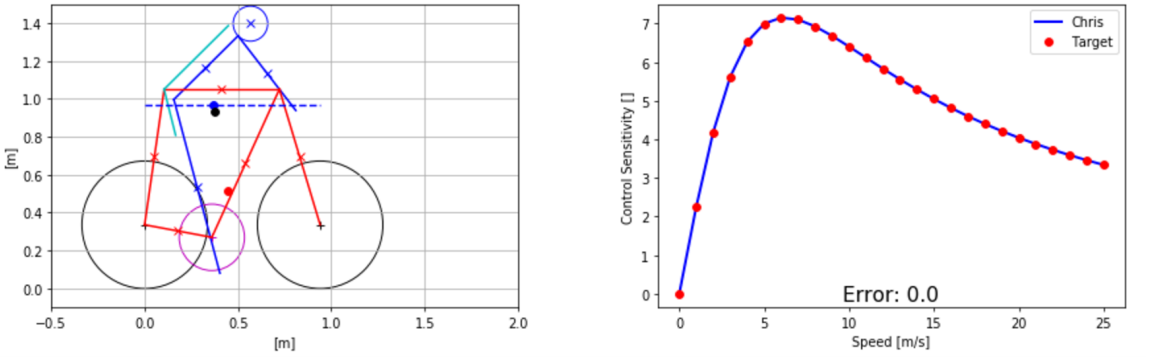


Figure 5.1: Example output from the bicycle plotter with the left plot showing the bicycle and rider configuration and the right plot showing the resulting handling curve graphed alongside a target handling curve. The 0.0 error value shows that this bicycle design had the expected handling characteristics.

The second concern revolved around the proper implementation of the Patterson Control Model (PCM) itself. Since the Modified Patterson Control Model (MPCM) effectively reduces its input parameters into the 9 values that the original PCM takes, confirming that the model correctly computed the proper PCM curves was as simple as checking that the output of the PCM and MPCM matched for equivalent inputs. Here we say equivalent inputs because the two models take different parameters – however, the MPCM reduces its 14 parameters down to the 9 that the PCM takes internally. As such, inputs were fed into the MPCM and the 9 intermediate values that were produced were recorded and fed into a standalone copy of the PCM as well. This was done for myriad bicycle and rider combinations, and the resultant handling curves were compared to ensure they were identical. Figure 5.1 shows one such example run, with the handling curve computed by the MPCM matching the expected PCM curve exactly.

Ultimately the methods described above are not 100% fool proof, and some corner cases could exist that the testing did not uncover. However, since the output of the

MPCM shows the center of gravity values for all of the associated model components, it is likely that the engineer using the model would be able to at least detect an error and not move forward with incorrect values (if such a case were to occur).

5.2 Body Model Validation

Validation of the body model revolved around determining if the python implementation correctly computed the inertia values of each of the rider's geometric components, as well as the final radius of gyration of the rider's body about the ground plane. Note that this is different than trying to validate that the body model itself properly represents real world riders – this validation work was done by the work in [25] and [29]. To check whether the python code was correctly computing the body's inertia values, a SolidWorks model of a human body was generated and its inertial values were recorded as shown in Figure 5.2. The same parameters were then fed into the MPCM's body model and the resulting inertial values for each of the body's components as well as the overall radius of gyration were computed. Comparing the results of the two models showed less than a 1% deviation from one another, helping to bolster the accuracy of the python implementation.

5.3 Design of Experiments

5.3.1 Overview

The concept of *Experimental Design* or *Design of Experiments (DOE)* can be conducted in a variety of ways with varying levels of rigor. In a general sense, DOE is a series of documented steps which others can reproduce to obtain the same findings as the original experimenter. In a stricter sense, DOE is a rigid framework of steps outlined as follows:

- Step 1: Declare the Experiment Objective
- Step 2: Declare the Process Variables

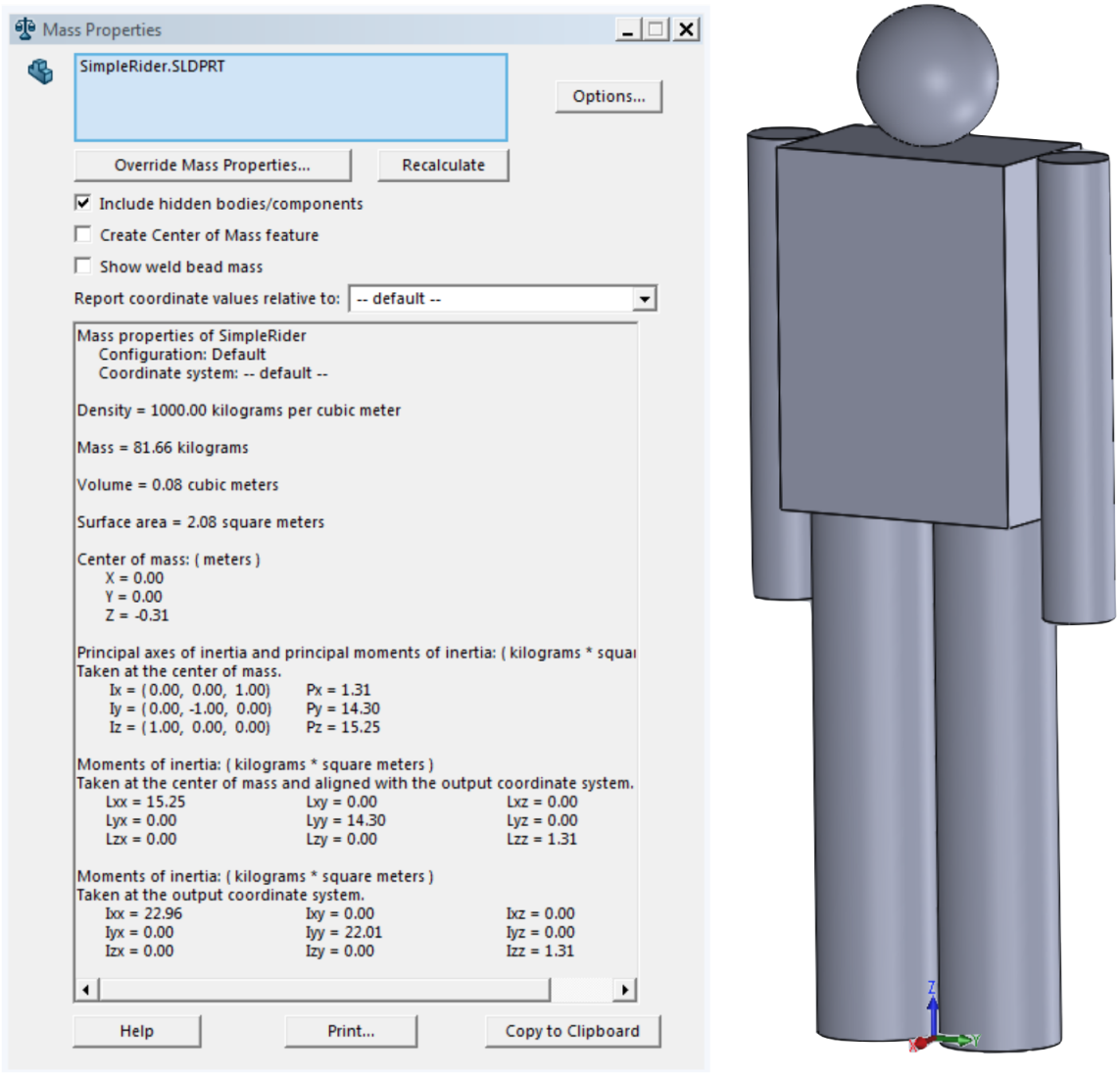


Figure 5.2: SolidWorks model of a rider along with its inertial properties.

- Step 3: Select the Experimental Design
- Step 4: Execute the Experiment
- Step 5: Analyze the Experimental Results
- Step 6: Report Results or Continue Iterating from Step 3

where each step is rigorously documented and iteration is expected/encouraged in many cases [3]. The *Experiment Objective* states the main goals of the experiment in clear, concise detail so that others can understand them without ambiguity. Each

experiment will revolve around a set of *Process Variables*, which can be broken down into *Inputs* (known as `Factors`), and *Outputs* (known as `Responses`). Common experimental factors include the problems being analyzed, the parameters to those problems which are being manipulated and the algorithm used to solve the given problem. Responses on the other hand are the results of the experiment that the researcher is looking to analyze and hopefully draw conclusions from in order to make statements regarding the relation between the observed response and the chosen experimental factors [7]. Selection of an *Experimental Design* should take into account the original experiment objective, since certain experimental setups lend themselves to certain objectives more than others. Common experimental designs include:

- *Comparative*: determines if one factor is important in the presence or absence of another.
- *Screening*: determines what factors are most important in a design and what factors are just noise.
- *Response Surface*: determines interactions between factors in order to come to an optimal factor combination.

Execution of the Experiment should be done in a way that can be easily replicated with as little ambiguity as possible, and *Experimental Analysis* should be conducted in a way that benchmarks other designs/implementations so that meaningful comparisons can be drawn when the results are reported to the rest of the community.

5.3.2 Experimental Setup

Unless otherwise mentioned, all experiments in this work were conducted under the Python3 runtime environment on a Intel i7-920 2.66 GHz quad-core chip with 8 gigabytes of DDR3-1333 ram running the Arch Linux kernel. At times, some experiments were conducted along side one another, with one executable running per core.

5.4 Brute Force Experiments

All experiments need a base benchmark to be compared against, and discovering the optimal bike design using Brute Force search is both the simplest, and the most computationally expensive method to start with. As such, a baseline implementation of this algorithm was created and run in order to show that there does exist an optimal bike which can be generated using the Patterson Control Model, and to provide a runtime benchmark to compare all other algorithms against.

5.4.1 Experiment Objectives

The objectives of the brute force experiments are twofold:

- To show that the model can find an optimal bicycle design through iteration.
- To measure the running time of the brute force implementation.

Having the model find the optimum bicycle design has two different practical uses. The first occurs when we are trying to show that the model does converge to the optimal solution, and in this case we would input a handling curve from a known bicycle and see if the model finds the *anticipated* design. The other case occurs when we do not know what the optimal bicycle design is for an arbitrary curve or design space, and are looking for the best bicycle design to use as a benchmark for other search implementations. Along a similar vein of thought, we look to also record the running time of this algorithm in order to compare its time efficiency to other search implementations.

5.4.2 Process Variables

In this experiment the factors include the following:

- Target Control Sensitivity Curve
- Bike Design Space

- List of Riders to fit to the generated bikes

The *Target Control Sensitivity Curve* describes the handling characteristics of the bicycle we would like the Brute Force search to create. The *Bike Design Space* is the range of values for each input variable to the Modified Patterson Control Model, and their cross-product defines the searchable bicycle design space for this experiment. Finally, the *List of Riders* describes the shape and size of each rider that the final bike design is to be built for.

The responses for the experiments include:

- The optimal bike design generated by the Brute Force search algorithm.
- The required runtime to exhaust the search space.

5.4.3 Experimental Design

In an ideal scenario, it would be feasible to run the Brute Force simulation on all of the problem sets that the other algorithms will be run on. Unfortunately, it is likely that the Brute Force implementation will take far too long to finish enumerating every possible option in these large search spaces for this plan to be feasible, and as such, a full brute force search of all design spaces was not conducted. Instead, the Brute Force implementation was used to search a *reduced design space* in order to show that the algorithm can indeed converge to the optimum bicycle design. To do this, a bicycle handling curve was first generated using the Modified Patterson Control Model, and was then fed into the Brute Force algorithm along with an appropriate bicycle design space and rider configuration. Once the simulation ended, the top design was compared against the datum design to ensure that they matched. Additionally, the number of bicycle combinations was divided by the experiment's total runtime to get an estimate of (runtime / iteration), and this estimate was then used to extrapolate the amount of runtime needed for the larger problem sets (so it can be used as a pseudo-benchmark).

For this benchmark, the Gemini recumbent bicycle parameters (see Appendix B.1) were fed into the Modified Patterson Control Model with the rider specified as

Chris (see Appendix C.1). This resulted in the handling curve values shown in Table E.1. Additionally, the parameter set outlined in Appendix B.3 was used to generate a meaningful reduced search space that ensured that the Brute Force algorithm would finish in a reasonable amount of time.

5.4.4 Experimental Results

The brute force implementation described above running on the machine described in Section 5.3.2 took ≈ 5760 seconds to complete, and successfully found the optimal design (which was the design with zero error that directly reflected the Gemini bicycle frame parameters used as inputs) as shown in Figure 5.3. The total number of iterations taken was ≈ 12440000, resulting in an average time per iteration of ≈ 0.463 milliseconds. If the same experiment had been conducted on the design search spaces used for the genetic algorithm experiments, it would have taken ≈ 40060 seconds (≈ 11 hours), showing the infeasibility of the brute force method as a practical search method for most large studies.

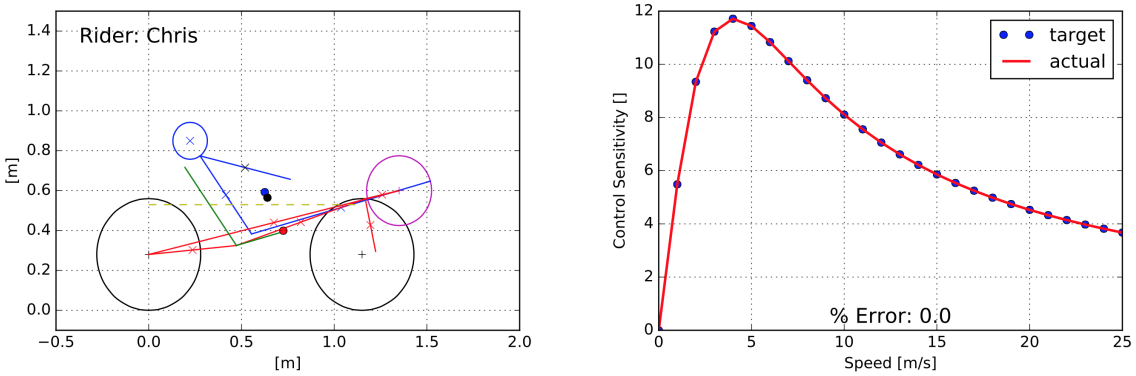


Figure 5.3: Resulting bicycle design generated by Brute Force search with Chris and Gemini as the datums.

5.5 Unpartitioned Genetic Algorithm Experiments

The *Unpartitioned Genetic Algorithm Experiments* act as an initial step into using the concept of Genetic Algorithms and Evolutionary Computing to search the bicycle

design space. The ultimate objective of this implementation is to create a platform to allow designers to use the Modified Patterson Control Model as a proactive design tool, hopefully helping to expose new designs while cutting down on the runtime required by the Brute Force implementation.

5.5.1 Experiment Objectives

The objectives of these experiments are to:

- Determine the optimal combination of genetic operators for each design space.
- Determine how operators change as bicycle design spaces change.
- Determine what the time to output-quality trade-offs are.

In this context, an *optimal combination of genetic operators* is a vector of operators that consistently produces bicycle designs whose handling curves are closest to that of the target handling curve. These parameter vectors may change as the problem space changes, and as such multiple design spaces needed to be tried in order to see what trends emerge. Finally, it is likely that the more computationally expensive runs will produce higher quality results, so analyzing how the final bicycle design quality changes with runtime was conducted to see where the practical break-even point was.

5.5.2 Process Variables

The experimental factors for these tests include:

- Population Size
- Generation Count
- Selection Percentage
- Cross-Over Percentage
- Mutation Percentage
- Target Control Sensitivity Curve
- Bike Design Space
- List of Riders to fit to the generated bikes

where all of the genetic algorithm parameters are as they were described in Section 2.4. Two major bicycle design spaces were analyzed – the recumbent space and the safety bike space. In both cases, an existing bicycle's parameters were run through the Modified Patterson Control Model to produce a *target control sensitivity curve* for use in the subsequent experiment. In order to reduce ambiguity, only a single rider was used in the experiments so that the exact solution would exist in the design space.

The responses that were recorded include:

- The vector of genetic algorithm parameters which produced the optimal bicycle design.
- A breakdown of the runtimes and bicycle design scores for all attempted genetic parameter vectors.

The results of each trial were then aggregated and analyzed to discern trends between genetic operator vectors and algorithm performance, with the ultimate aim to discover an optimum vector which would be useful across multiple bicycle design spaces.

5.5.3 Experimental Design

Since this is a problem which looks to find the optimal set of parameters (namely the vector of genetic algorithm parameters to consistently generate the best bikes), it appeared that either a *Full Factorial Search* or *Response Surface Modeling* would align best with the experiment's objectives. Fortunately, since the number of combinations of genetic operators are relatively small, a full factorial search could be conducted to ensure that the optimum operator vector is not accidentally overlooked. In addition, the genetic algorithm tuner described in Section 3.6 was also run over the design space to see what general trends appeared on a per operator basis, and to see if that tuner would have correctly located the range of values containing the optimum operator vector. Thus, this search study allowed for the optimum operator vector to be discovered and helped to vet the viability of using Response Surface Modeling in

this problem domain. The list of genetic operators used in this experiment, as well as their associated value options are outlined in Table 5.1 with their genetic operator configuration file being shown in Appendix D.1.

Table 5.1: Genetic Algorithm Parameter Space

Variable	Values
Population Size	100, 200, 300, 400, 500
Generation Count	2, 4, 6, 8
Selection Percentage	5, 10, 15, 20, 25
Cross-Over Percentage	5, 10, 15, 20, 25
Mutation Percentage	5, 10, 15, 20, 25

As for problem spaces, two were chosen for consideration: the recumbent design space and the safety bike design space.

For the recumbent space, the Gemini recumbent bicycle parameters (see Appendix B.1) were fed into the Modified Patterson Control Model with the rider specified as Chris (see Appendix C.1). This resulted in the handling curve shown in Appendix E.1. These parameters were used as inputs to the recumbent space experiment along with the bicycle parameter space outlined in Appendix B.4.

For the safety bike space, the Cervelo R3 Team parameters (see Appendix B.2) were fed into the Modified Patterson Control Model with the rider specified as Chris (see Appendix C.1). This resulted in the handling curve shown in Appendix E.2. These parameters were used as inputs to the recumbent space experiment along with the bicycle parameter space outlined in Appendix B.5.

5.5.4 Experimental Results

Factorial Search Results

A full factorial search was conducted across the genetic design space outlined in Table 5.1, resulting in 2500 different parameter combinations that needed to be

tried per run. Due to the stochastic nature of genetic algorithms, 20 full factorial searches were run over the design space with the average results being reported below. Additionally, the following sections focus on analyzing their experimental results to determine the best genetic operator vector, where *best* is the vector that produces results with the lowest average error (as described in Section 3.2). Much of the results are expressed using parallel-coordinates graphing, with line coloring helping to represent the solution quality (where *hot colors* such as red - yellow are solutions with low error and *cool colors* (green - blue) are solutions with higher error). Finally, all experiments were run on the machine setup described in Section 5.3.2.

Recumbent Full Factorial Trial

The results of the full factorial search over the recumbent design space are displayed in Figures 5.4 and 5.5, which show the top 1000 genetic vectors as well as the optimum vector. The average runtime for a full factorial pass over the design space was ≈ 1490 seconds, with the optimal genetic operator vector consisting of the values outlined in Table 5.2, which on average produced recumbent designs with an error value of 0.035 and took 1.31 seconds to run.

Table 5.2: Optimum Genetic Algorithm Recumbent Parameters

Variable	Values
Population Size	500
Generation Count	8
Selection Percentage	5
Cross-Over Percentage	25
Mutation Percentage	10

Given that the runtime is relatively low, this is the genetic operator vector that would be recommended for studies moving forward. However, looking at a single run does not give much insight into the overall characteristics of the genetic platform over the recumbent design space. To do this, we must look at several different ranges of the design space to see if the trend of high cross-over and low selection and mutation

persists. Figure 5.4 shows the solutions with filters set on the population size and the generation count, and helps to give an idea of how influential generation count is on the final solution's quality. Figure 5.5 shows a similar set of graphs, however there is filtering set to only show the best set of solutions per graph to help give more insight into the underlying operator trends. Note that in both figures the population size was fixed to 500 – this was done to reduce the clutter of lines in the graph, allowing colors to show through and trends to more easily emerge (and the larger populations tend to yield the best results so this generalization holds there as well). The correlations between variables in Figures 5.4 and 5.5 are summarized as follows:

- ↑ Generation Count yields ↓ Error
- ↑ Cross-Over Percentage yields ↓ Error
- ↓ Selection Percentage yields ↓ Error
- ↓ Generation Count requires ↑ Mutation Percentage to yield ↓ Error

Note that there is not a strong correlation between error and mutation percentage as shown in this data set, with mutation percentage tending to jump around seemingly randomly. This may be indicative of one of several factors including the size of the search space and the characteristics of its fitness landscape. For instance, it could be the case that the fitness landscape has many different local optimums that the algorithm can locate and converge to, and as such a higher mutation rate may be beneficial to increase the chance of finding a better solution from generation to generation. This would also explain why the mutation rate tends to increase as the generation count decreases; the algorithm needs to up its random search to find a decent solution in a shorter number of iterations.

5.5.5 Recumbent Sampling Tuner Search Results

A sampling tuner study was also run over the genetic parameter set within the context of the recumbent design space outlined in the above experiment. The intention of this secondary experiment was to determine whether or not a sampling tuner

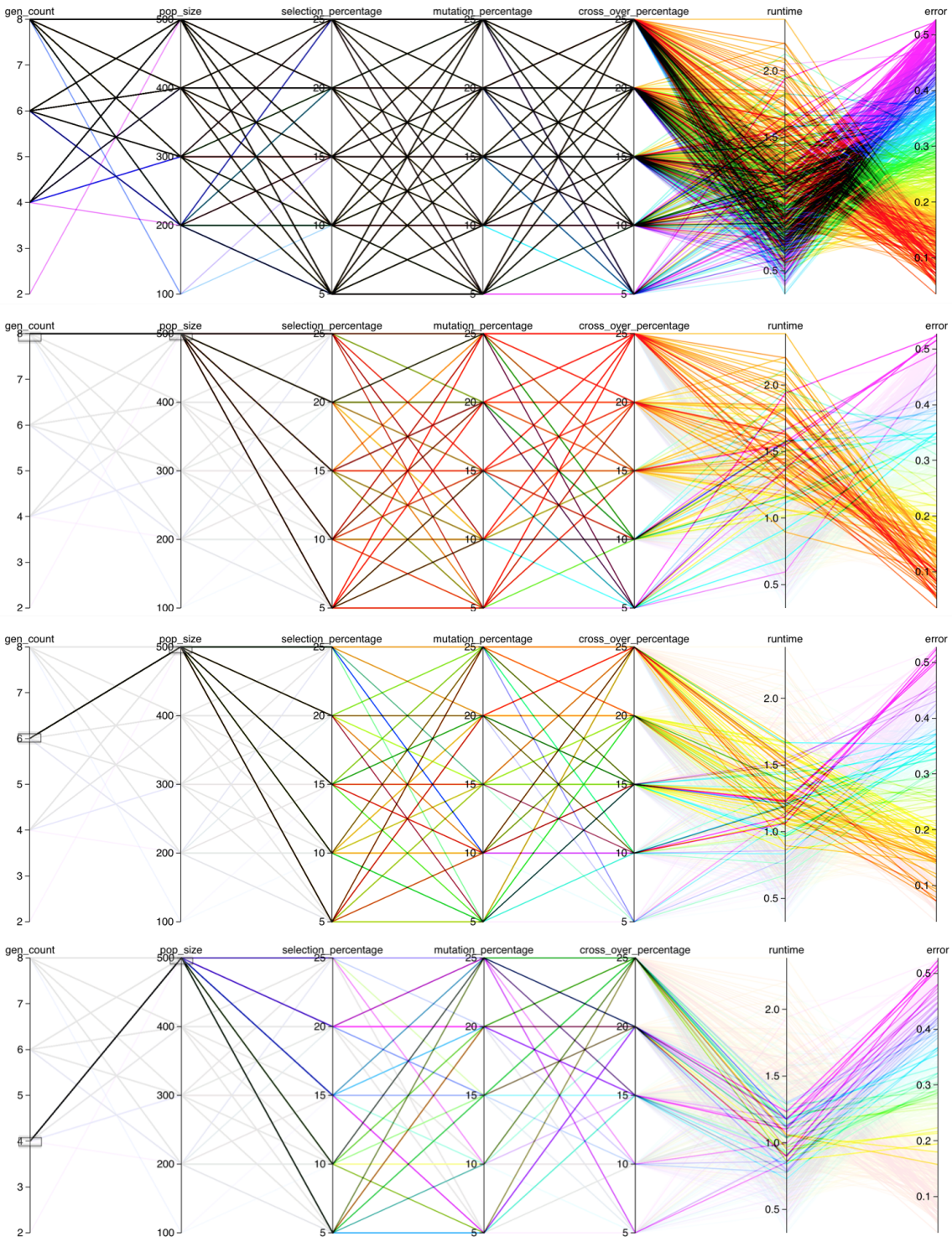


Figure 5.4: The top 1000 designs from the full factorial experiments over the recumbent design space.

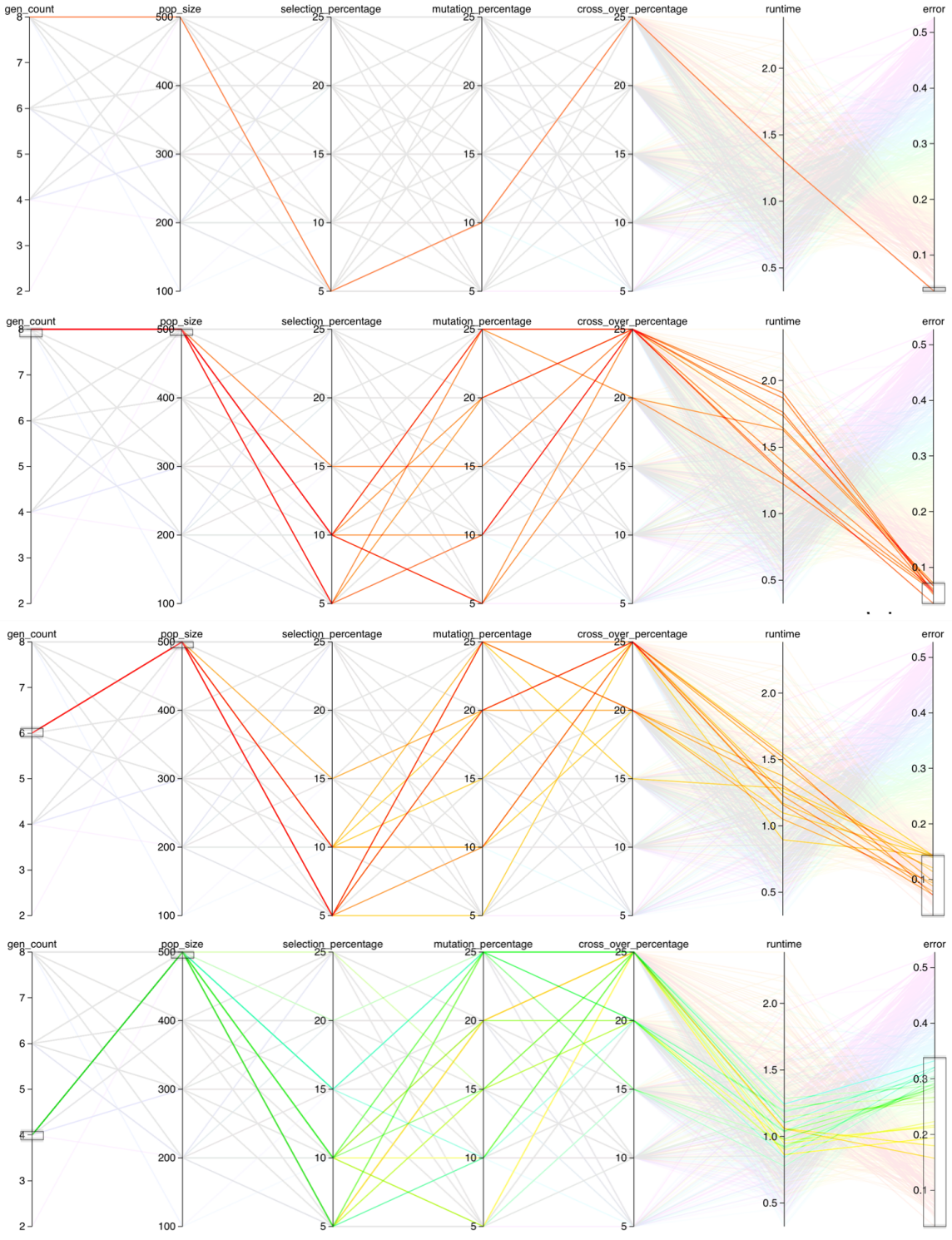


Figure 5.5: The top designs per segment for the full factorial experiments over the recumbent design space.

could produce similar results as the full factorial search, which may allow it to be used in its place in future experiments. As such, the sampling tuner was setup to take the following inputs:

- Generation Count: [1, 2, 4, 6, 8]
- Population Size: [100, 200, 300, 400, 500]
- Selection Percentage: [5, 10, 15, 20, 25]
- Mutation Percentage: [5, 10, 15, 20, 25]
- Cross Over Percentage: [5, 10, 15, 20, 25]

Note that these are the same parameters used in the previous experiment with the addition of a generation count of 1 (this needed to be done so that all parameter vectors had the same length, otherwise they could not form a Greaco-Latin Square in the sampling tuner). Additionally, in order to try and add more confidence to the results of the experiment, the number of trials per configuration were increased from 20 to 100. Since the purpose of this initial experiment is only to determine the validity of this tuner, only the sampling stage was run as described in Section 3.6. The results of these trials are shown below in Figure 5.6, with each attribute being plotted separately to try and make trends more evident.

Looking at all of these plots as a whole, it is evident that there are much larger error ranges for smaller values of each parameter. Reflecting on the trends observed in the full factorial search, this is likely due to the influence of low generation count and population size values more than the result of low selection, mutation of cross-over operators. The plots for generation count and population size support this theory by showing that both the average solution error and the solution spread decrease as their parameter value increases (which directly reflects what was seen in the full factorial search as well). However, while the generation count and population size plots seem to follow the trends observed in the full factorial search, the other genetic operators are not as clear cut. While both the mutation and cross-over percentage plots seem to show a general trend of increasing solution quality with increasing parameter values, this trend seems weak at best. Adding concern to these findings is the selection

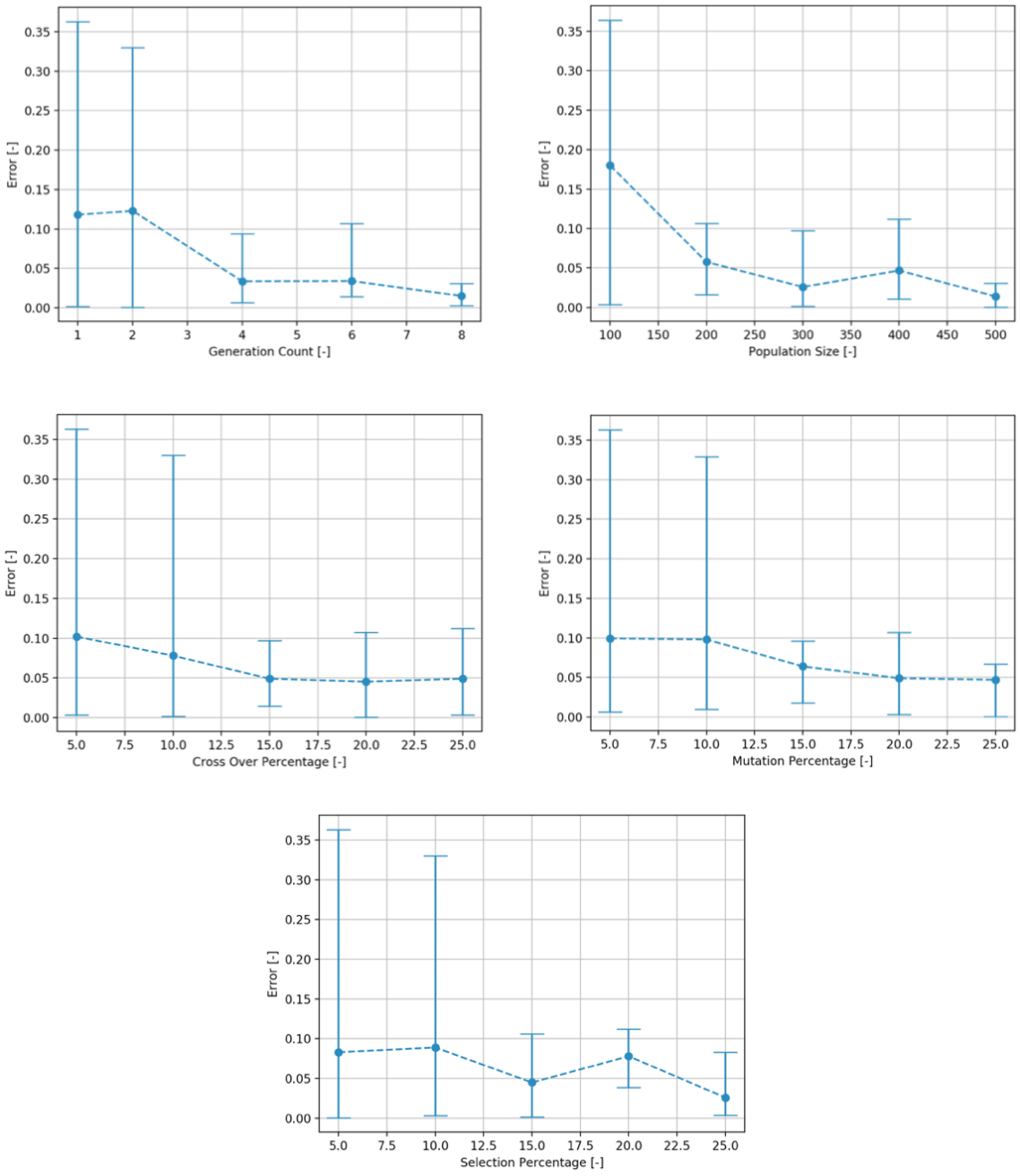


Figure 5.6: Sampling tuner error values for each parameter vector.

percentage plot which has no reasonable trend and instead seems erratic and random in nature. However, this randomness can also imply that the selection parameter has less influence on the overall solution quality than the other genetic operators. Another noteworthy point is that this study found the selection percentage to be weakly coupled to the final solution quality, while the full factorial findings showed

mutation percentage to be erratic in nature. These deviations from the datum coupled with the fact that these final three plots show only muddled trends gives concern to using this method as a tuning solution in future design space searches.

Safety Bike Full Factorial Trial

The full factorial search of the safety bike design space was carried out in a similar fashion to that of the recumbent full factorial experiment, and in line with that, the top 1000 genetic vectors as well as the optimum vector are shown in Figures 5.7 and 5.8. The average runtime for a full factorial pass over the design space was ≈ 1480 seconds, with the optimal genetic operator vector consisting of the values outlined in Table 5.3, which on average produced recumbent designs with an error value of 0.022 and took 1.22 seconds to run. As with the recumbent experiments, Figure 5.7 plots the top 1000 genetic vectors and shows how the performance of the algorithm changes based on changing generation count, and Figure 5.8 shows the top solutions for each of those filtered graphs to make trends easier to spot.

Table 5.3: Optimum Genetic Algorithm Safety Bike Parameters

Variable	Values
Population Size	500
Generation Count	8
Selection Percentage	5
Cross-Over Percentage	25
Mutation Percentage	5

It is noteworthy that the average solution error over the safety bike space was 40% less than that over the recumbent design space. This is likely due to how the MPCM fits riders to bicycles and the constraints that the model enforces. Specifically, the MPCM requires that the bicycle crank's X and Z location be fully specified, along with the height of the rider's seat. This works well in a recumbent design space where the rider's seat is allowed to *slide forwards and backwards*, allowing the model to accommodate a large variety of crank/seat locations. However, in a safety bike

design the rider is usually placed above their cranks at a set distance and uses the *up and down* motion of the seat to adjust for rider height changes. Since the MPCM does not allow this, it can result in a large number of safety bicycle designs being considered invalid. This then leads to a large pruning of the design space, which likely makes it easier to iterate to a stronger final solution. Note that this pruning of the search space is not necessarily a negative feature when fitting a single rider, however it does mean that fitting multiple riders to a safety bike produces fewer valid solutions than fitting multiple riders to a recumbent.

5.5.6 Experiment Conclusions

The purpose of these experiments was to determine what the best genetic vector was for both the recumbent and the safety bike design spaces. After running a full factorial search over both spaces, it was found that the top genetic vector from each differed only in the mutation percentage, and even then only by 5%. With that in mind, we can conclude that these two design spaces behave in roughly similar fashions with respect to the genetic framework, and the single genetic vector outlined in Table 5.4 can be used across both spaces with great success. Note that the recumbent genetic vector was chosen to be used across the entire bicycle design space – this choice was made because the PCM (and thus the MPCM) better models recumbent riders than safety bike riders due to the nature of the rider's orientation in the bicycle, and as such it is likely that the MPCM model will be used to model recumbent bicycles more often than safety bicycles. Additionally, this optimum genetic vector will not necessarily work best across all design spaces in the MPCM, rather it works well for the spaces that were investigated in this study. However, given how distinct the recumbent design space is from the safety bike design space, it is likely that the genetic operators listed in Table 5.4 will produce high quality solutions across the entirety of the bicycle design space.

In addition to these full factorial searches, a sampling tuner experiment was conducted over the recumbent design space in an attempt to characterize its performance and correctness. While both generation count and population size showed strong correlations that matched the findings in the full factorial search, the other genetic

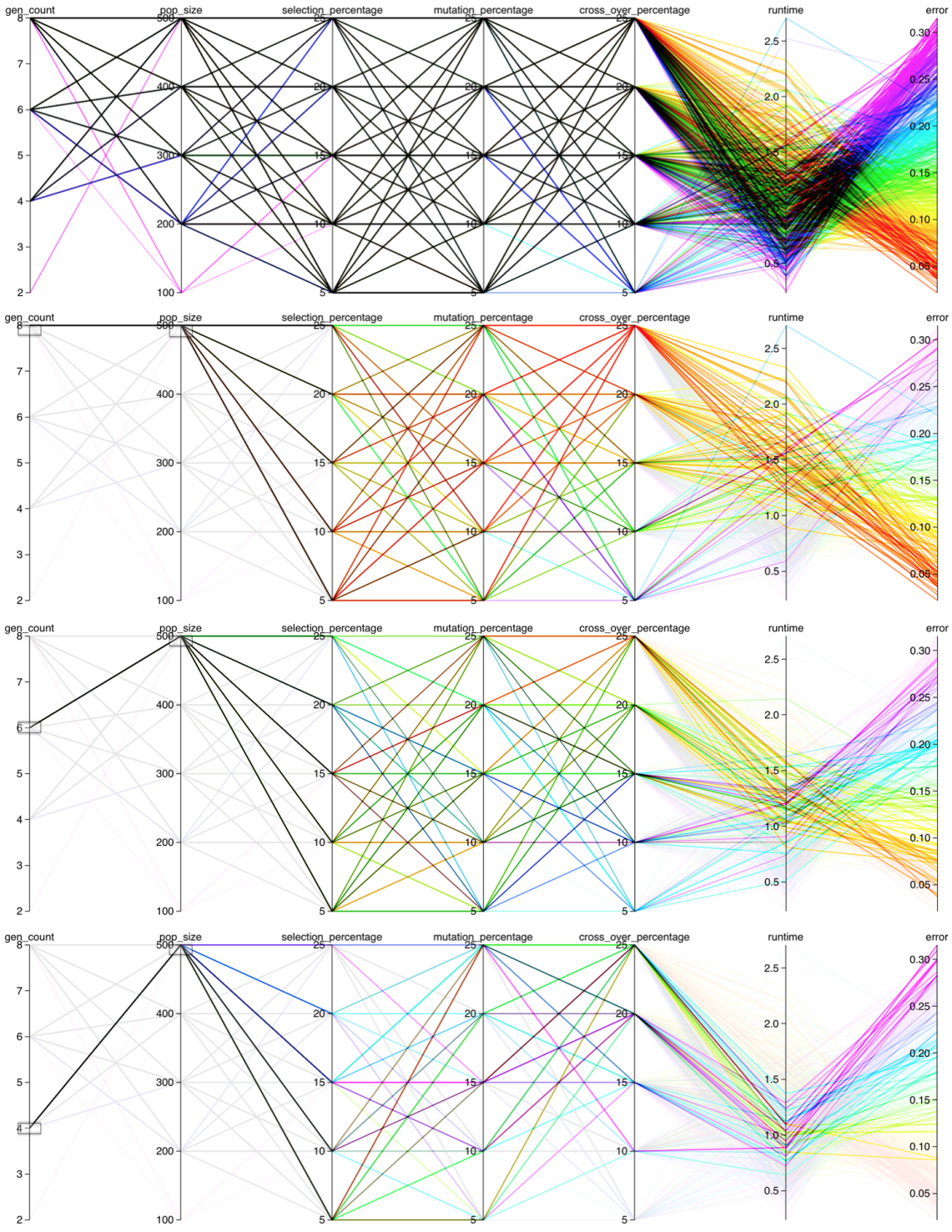


Figure 5.7: The top 1000 designs from the full factorial experiments over the safety bike design space.

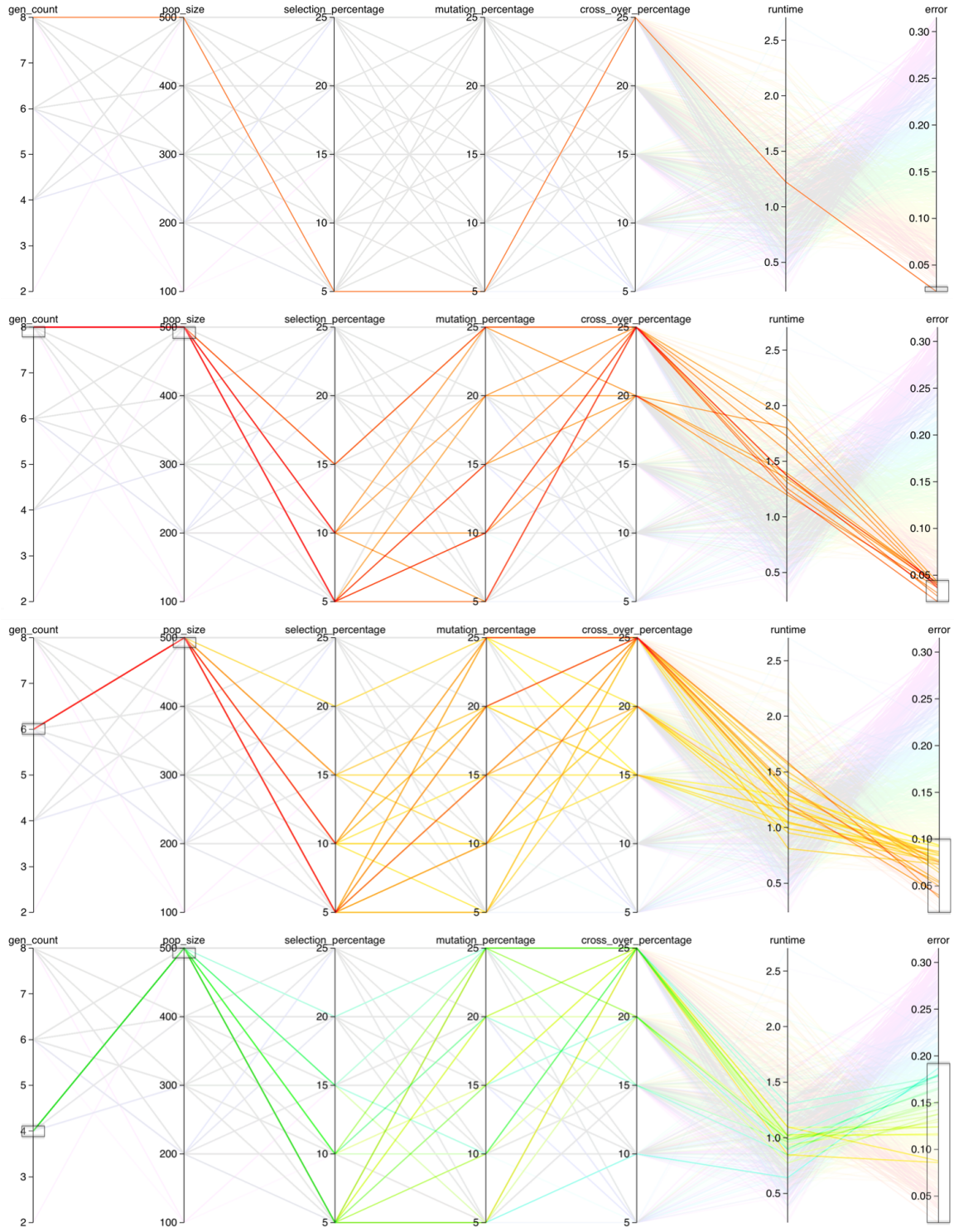


Figure 5.8: The top designs per segment for the full factorial experiments over the safety bike design space.

Table 5.4: Optimum Genetic Algorithm Parameters For Entire Bicycle Space

Variable	Values
Population Size	500
Generation Count	8
Selection Percentage	5
Cross-Over Percentage	25
Mutation Percentage	10

operator findings were either weakly correlated at best or opposite the findings of the full factorial trials. Because of this, the sampling tuner is not recommended for use in characterizing the search spaces examined in these studies.

5.6 Partitioned Genetic Algorithm Experiments

One shortcoming to genetic platforms is their tendency to become overly focused on a single subset of the design space, often missing the optimal solution. One way to circumvent this is through reducing the influence that the top designs have on the iterative process, and partitioning is one way to achieve this. As such, the partitioning algorithm presented in Section 3.7 was applied to the genetic platform described in Section 3.5 as outlined in Section 3.5.2.

5.6.1 Experiment Objectives

These experiments aim to gain insight into how helpful the application of partitioning is in concert with genetic algorithms when applied to the bicycle design space, with the final goal of determining what the best vector of partitioning parameters is. As with previous experiments, *best* here signifies the combination of parameters that routinely yields solutions with the lowest error, where error is measured using the fitness function outlined in Section 3.2.

5.6.2 Process Variables

Since the partitioning algorithm is being run on top of the genetic platform, there are myriad factors that could be considered in the following experiments. However, in order to keep the complexity of the experiments down while focusing on tuning the partitioning algorithm, only the following experimental factors will be examined:

- Partitioning radius
- List of bicycle attributes to partition about

where the partitioning radius is a series of decimal values, and the list of bicycle attributes can be any non-zero combination of attributes outlined in Table 3.4.

The responses that will be recorded include:

- The averaged runtime and errors of all combinations of partition radii and bicycle attributes.
- The vector that produces the optimum solution per partitioning radius.

5.6.3 Experimental Design

While the main focus of this experiment is to determine what the optimum partitioning parameters are, the tests still need a fixed set of bicycle, rider and genetic search spaces to operate within. Since the experiments in Section 5.5 heavily examined subsets of the recumbent and safety bike design spaces, it seemed sensible to use those spaces in this experiment as well. Additionally, since the aim is to find a more optimum solution than the unpartitioned variants, these experiments will use the optimum genetic algorithm configuration that is listed in Table 5.4. With these search spaces fixed, the partitioning parameter space could be expanded and explored with the knowledge that any improvements that are found will be better than the results uncovered in the unpartitioned experiments.

Defining the Partitioning Parameter Space

The partitioning design space has three primary inputs that need to be defined including:

- The partitioning radius
- The list of attributes to partition about
- The threshold to stop considering individuals at

with each having their own set of bounds that they can exist within. For the sake of simplicity, the partitioning algorithm's threshold value was set to 50%, and as such only the top half of each generation's population would be partitioned (and thus available for selection in the next generation). Additionally, while the partitioning radius can technically be any positive decimal value, it makes sense that it would not exceed the maximum distance between two points in the bicycle design space. However, this can still result in a massive range of radii to test, and as such, a more conservative approach was taken in these experiments, with radius integer values between 1 and 5 (inclusive) being tried and analyzed. Additionally, the MPCM has 15 bicycle parameters which could be used in the partitioning distance computation. After some consideration, this list was reduced to the 9 parameters listed in Table 5.5.

The masses of the frame, rider, crank and wheels were omitted because they are generally fairly constant from run to run, and the radius of the crank and the rear wheel were excluded because they have the smallest impact on the underlying Patterson Control Model equations. In order to insure that the optimal radius/attributes configuration would not be overlooked, all combinations of radii and attributes was attempted, with the results being recorded and analyzed below.

5.6.4 Experimental Results

A full factorial search was conducted across the entire design space outlined in Section 5.5.3, resulting in 2560 different parameter combinations that needed to be

Partitioning Attributes List
wheelbase
fork_offset
handlebar_radius
front_wheel_radius
crank_x_offset
crank_z_offset
seat_height
hip_angle
headtube_angle

Table 5.5: List of bicycle attributes to combine while tuning the partitioned genetic search platform.

tried per run. Similar to previous experiments, each trial was repeated 20 times with the average results being reported below. The following sections look to determine what effects partitioning has on the quality of solutions produced by the genetic platform, and what partitioning parameter set yielded the best results. More specifically, the effects of changing the partitioning radius as well as the bike design parameters to partition about is reported and analyzed, with all experiments run on the machine setup described in Section 5.3.2.

Recumbent Full Factorial Trial

This study was conducted with the same input parameters as the recumbent genetic search trials presented in Section 5.5.4, and iterated over the partitioning parameter space as outlined in Section 5.6.3, with each of the 20 runs taking ≈ 8500 seconds on average to complete. The results of these trials are outlined in Figures 5.9 and 5.10. Figure 5.9 shows the averaged results for every trial, with line colors being defined by the final solution error. Figure 5.10 highlights the top configuration for each radius. Note that these graphs are each for a single partitioning radius,

and that the majority of their axes represent the partitioning attributes. These axes represent a binary include/exclude of their respective attribute in each partitioning configuration. For example, the optimum partitioning configuration for a partitioning radius of 1.0 shown in Figure 5.10 partitioned the design space using `wheelbase`, `crank_x_offset` and `seat_height` (since those are the three axes where the configuration passed through them with a value of 1.0). Each graph also includes the minimum, maximum and average errors, as well as the average runtime. Finally, the error and runtime values for the optimum vector per partition radius are outlined in Table 5.6.

Table 5.6: Runtime and error results for the optimum recumbent partitioning vectors.

Partition Radius	**1**	**2**	**3**	**4**	**5**
min_error	2.7e-05	1.3e-27	7.2e-05	4.6e-06	0.0019
max_error	0.0754	0.0789	0.0980	0.0727	0.0757
avg_error	0.0233	0.0244	0.0256	0.0209	0.0229
avg_runtime [sec]	6.65	6.78	6.54	6.89	6.79

Safety Bike Full Factorial Trial

The same experiment was run over the safety bike design space with each of the 20 trials taking ≈ 8400 seconds to complete, producing the results outlined in Figures 5.11 and 5.12. Figure 5.11 shows the averaged results for every trial, with line colors being defined by the final solution error. Figure 5.12 highlights the top configuration for each radius in a similar fashion to the recumbent experiments from the previous section. Finally, the runtimes and errors for the top vector for each partition radius are listed in Table 5.7.

5.6.5 Experiment Conclusions

Looking at the data from the recumbent and safety bike search trials shown in Figures 5.10 and 5.12, there is no obvious correlation between partitioning attributes

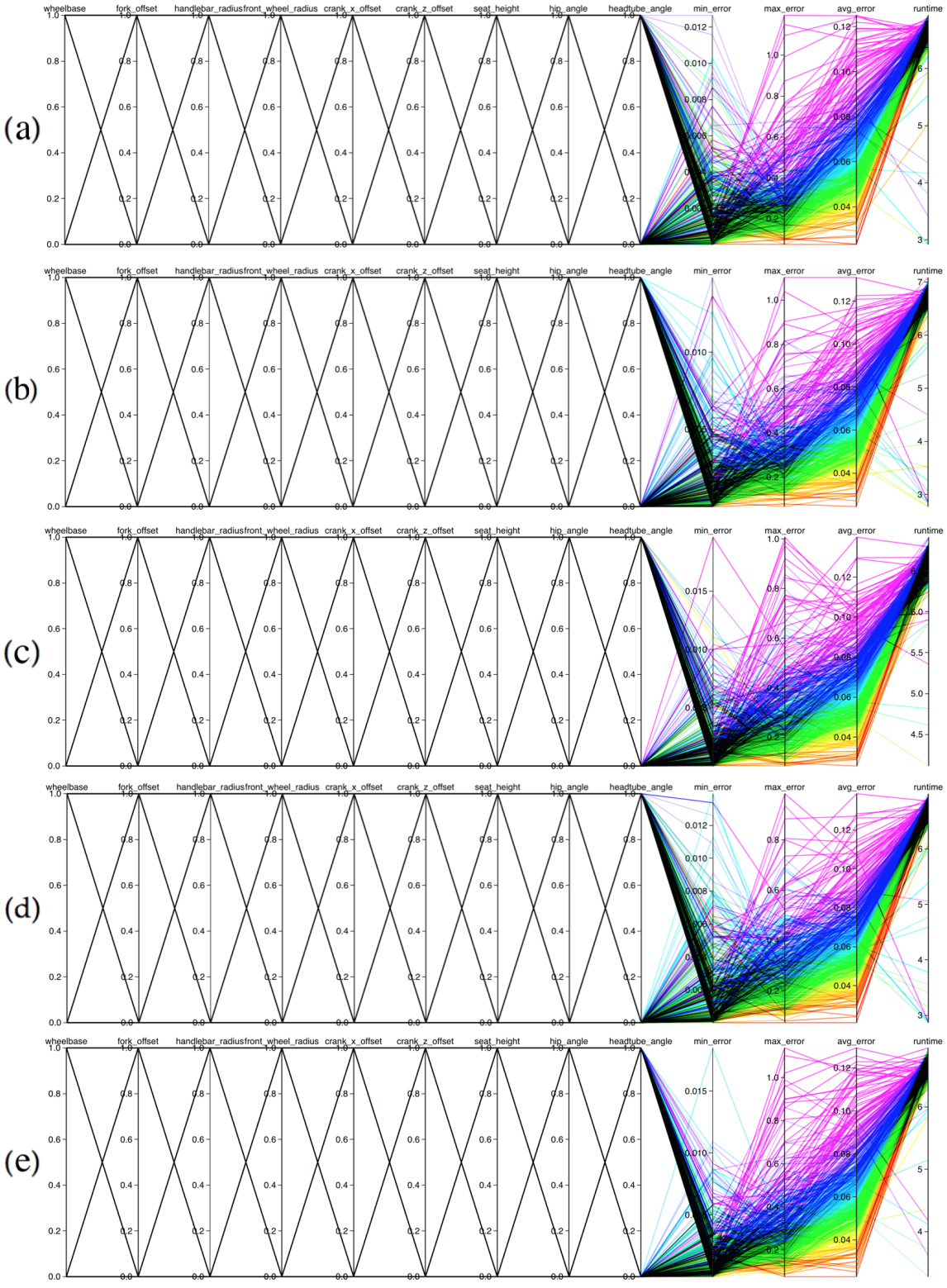


Figure 5.9: All recumbent tuning results with a radius value of a) 1.0 b) 2.0 c) 3.0 d) 4.0 e) 5.0.

Table 5.7: Runtime and error results for the optimum safety bike partitioning vectors.

Partition Radius	1	2	3	4	5
min_error	0.0007	8.0e-05	0.0003	0.0031	0.0004
max_error	0.0696	0.0900	0.0664	0.0731	0.1392
avg_error	0.0190	0.0178	0.0170	0.0230	0.0191
avg_runtime [sec]	6.69	6.39	6.58	6.55	6.79

and quality of solution. This is most likely due to the complex interactions that occur between the partitioning radius and the attributes, with every additional attribute that is considered adding another dimension to the partitioning space. Figure 5.13 shows the top result for each of the runs, and while there is a slight trend in how both lines are curving, in general the deviation is so small that it is hard to consider noteworthy.

However, while there were no direct trends between individual parameter combinations, on average all partitioned trials produced solutions with roughly 80% of the error of their unpartitioned counterparts. More interestingly, even the partition configuration where there were no partitioning attributes performed better than the unpartitioned version. In this case, it is likely due to the partitioning algorithm's threshold step which removes the bottom 50% of all individuals from the population. This would result in there being a generally higher quality population for the random selection to be performed on from generation to generation, which appears to have improved the final solution quality.

Finally, it is important to keep perspective regarding these results. Specifically, while it is true that the partitioned implementation performed better than its unpartitioned variant, the difference between the resulting design scores is incredibly small – small enough to be considered essentially negligible. Thus, the practical conclusion to draw from these experiments is that the user can use either genetic search platform because both will produce the same quality designs. However, the primary benefit to using the partitioned search platform is that it outputs the top design from each of the resulting bike partitions. Thus, the user can specify how they want to parti-

tion their results, making it easier to spot *design groups*, and ultimately see different families of candidates more easily in the final output.

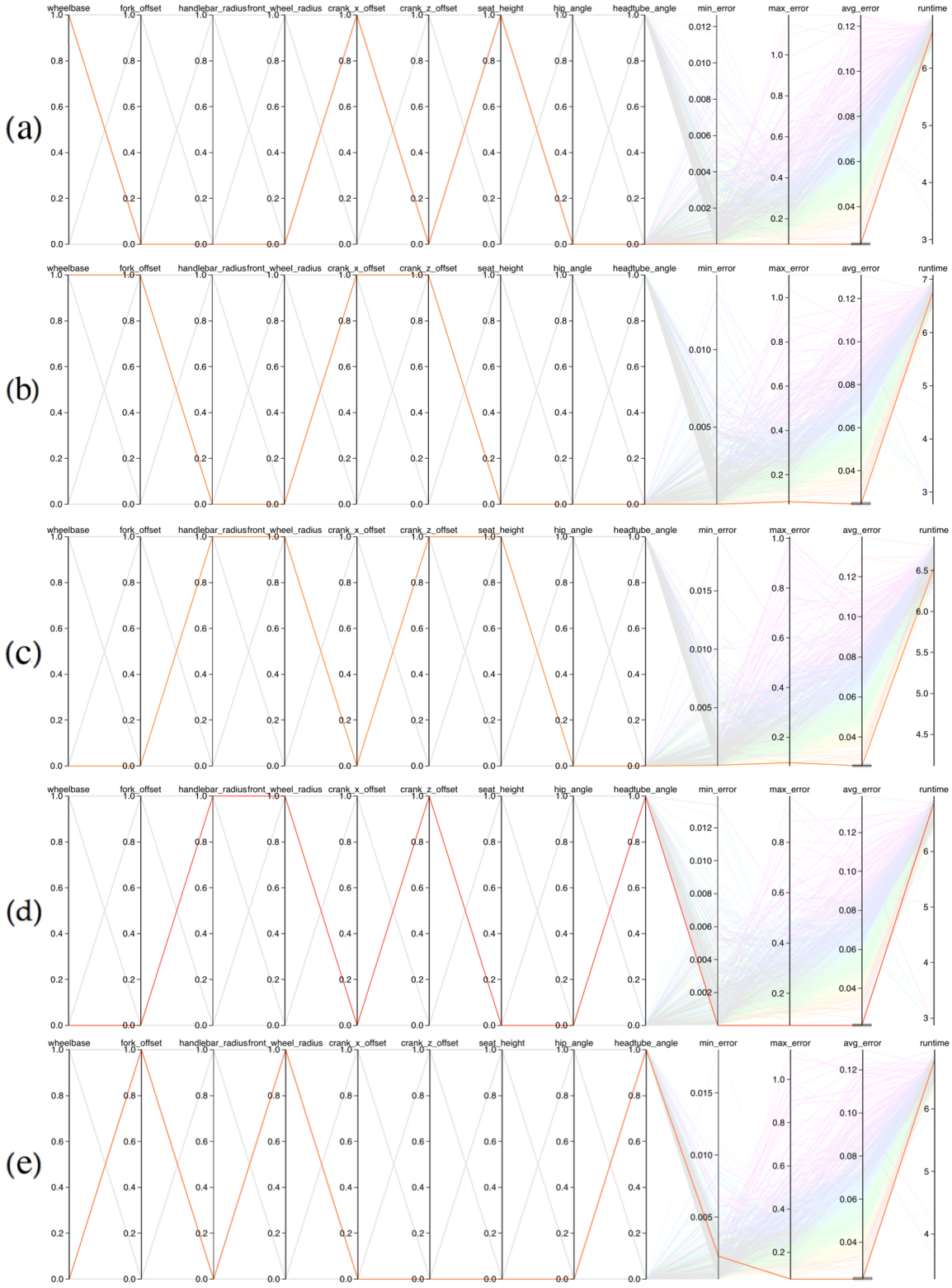


Figure 5.10: Top recumbent tuning results with a radius value of a) 1.0 b) 2.0 c) 3.0 d) 4.0 e) 5.0.

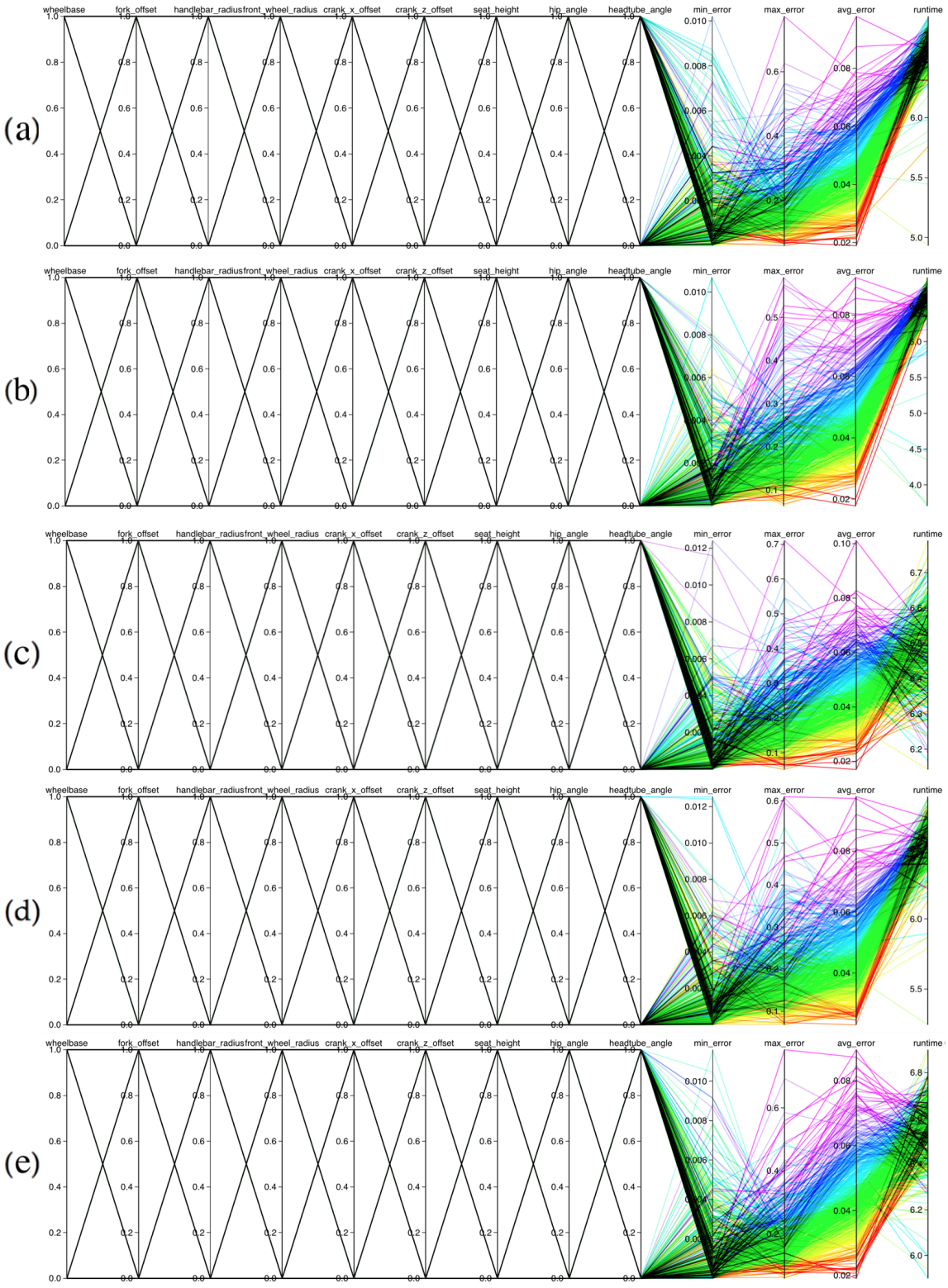


Figure 5.11: All safety bike tuning results with a radius value of a) 1.0 b) 2.0 c) 3.0 d) 4.0 e) 5.0.

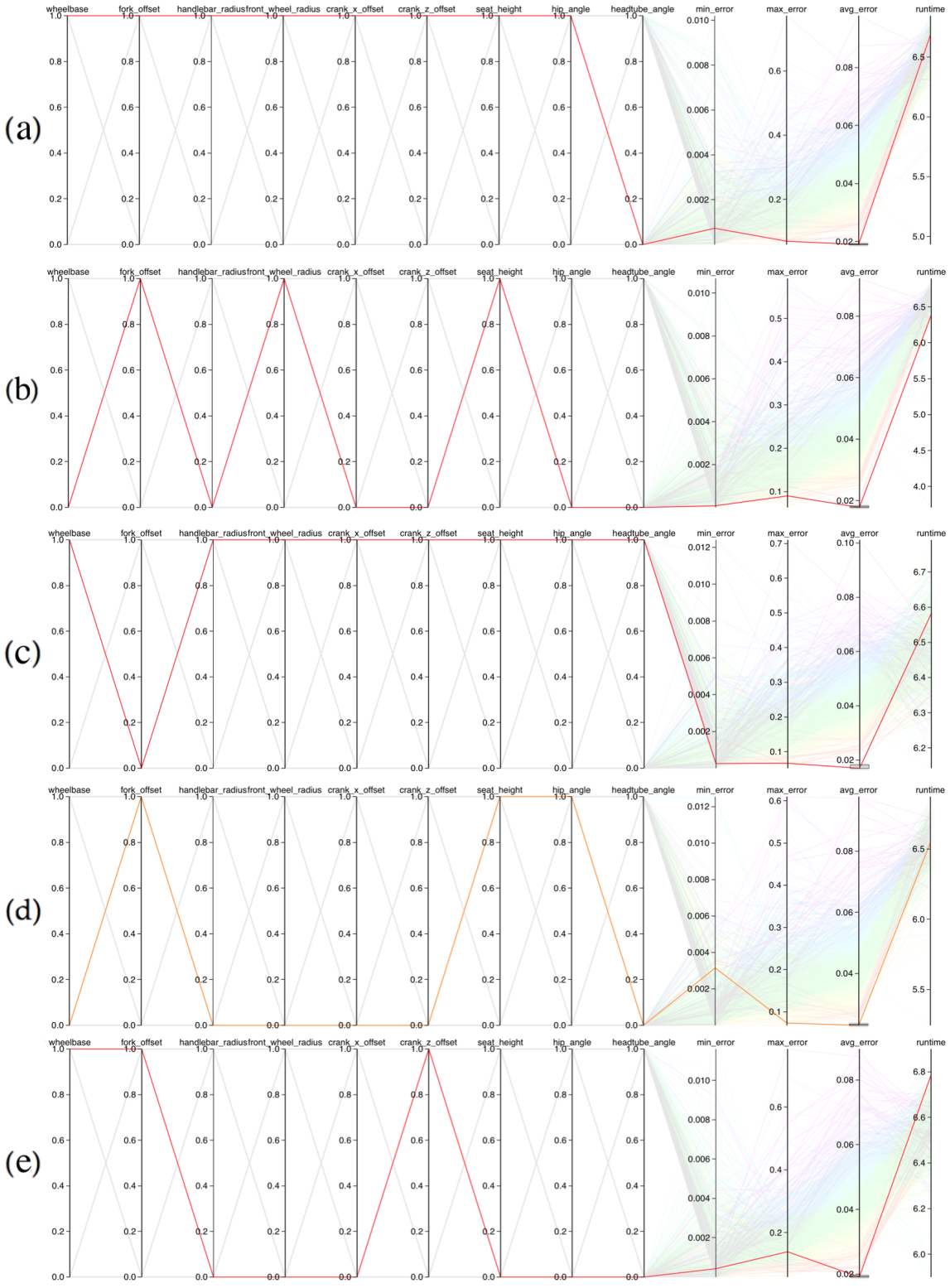


Figure 5.12: Top safety bike tuning results with a radius value of a) 1.0 b) 2.0 c) 3.0 d) 4.0 e) 5.0.

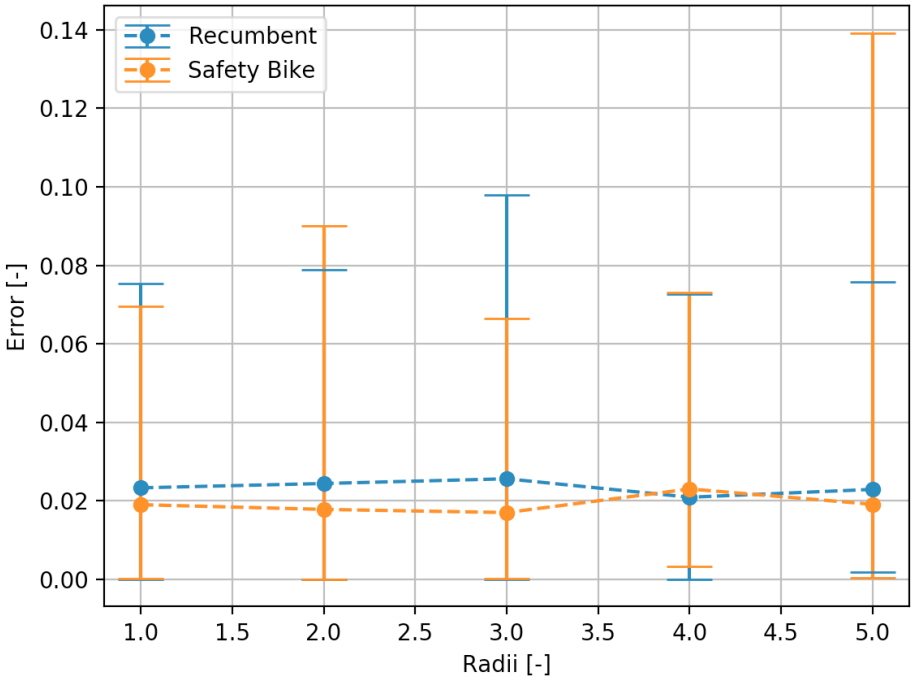


Figure 5.13: Partition radii versus top errors for the recumbent and safety bike design spaces.

Chapter 6

Conclusions

The primary goal of this work was to use the Patterson Control Model to proactively search the bicycle design space and hopefully find rideable bicycle designs that fit a designer's handling needs. This was accomplished by expanding the input set of the Patterson Control Model so that it included more bicycle parameters as well as a 3D body model for the bicycle's rider, which ultimately created a tool which was easier to use than the initial Patterson Control Model itself. From there, this model was integrated into several different search algorithms, all of which had the goal of determining which bicycle design most closely matched the designer's handling goals within a specified bicycle parameter space. While the initial Brute Force implementation was able to reliably locate the optimum bicycle design in any given search space, it was incredibly slow, making it impractical to use in real world design situations. To alleviate this issue, two platforms were created that utilized the strengths of genetic algorithms to search for near optimal bicycle designs in a fraction of the time that the Brute Force implementation required. The base genetic platform was able to find recumbent bicycle designs with an average error of 0.035 in 1.31 seconds of runtime and safety bicycle designs with an average error of 0.022 in 1.22 seconds, making this platform more than practical for a designer to use in the future. The addition of partitioning on top of the genetic search platform halved the solution errors at the cost of increasing the runtime by a factor of 6. However, even with a 6 times increase in runtime the platform still runs in a practical amount of time, making

either genetic implementation a feasible option for designers to choose between, with the benefit of the partitioning solution being that the designer can more easily group their output data so that distinct designs are more apparent. These search platforms were then wrapped in a Jupyter Notebook front-end so that users could more easily interface with their code, making tools which should be both useful and approachable for interested parties from both computing and non-computing backgrounds alike.

Chapter 7

Future Work

Improvements to the work presented in this document fall into two major categories: improvements to the Modified Patterson Control Model (MPCM) and improvements to the genetic search platform itself.

With regards to the MPCM, there are three primary avenues which could be explored. The first would be to investigate using more complex human body models to see what the improvements would be to the final inertial values for the bicycle/rider system. While these are likely small (as mentioned in the previous chapters), it could allow for higher fidelity results which may tweak the overall ranking of some bicycle designs over others. Additionally, many human powered vehicles (HPV's) are wrapped in a *fairing*, which is an aerodynamic shell that is used to reduce wind drag in an attempt to improve the bicycle's speed performance. The current implementation of the MPCM does not factor the fairing into the final inertial computations, which is something that would need to be done if faired bikes were to be accurately modeled in the future (especially since fairings can make up 25+% of some bicycle's overall mass). An initial idea for computing a fairing shape would be to use Convex Hull to get the polygon that encloses the bicycle. This could then be done for varying y-offsets from the bicycle's center in an attempt to capture the 3D shape of a bicycle/rider system. The mass of the 3D fairing could then be computed and used in the overall bicycle inertia calculations. As an extension of the fairing idea, smoothing and aerodynamic rules could also be applied to the final result, allowing the design's aerodynamic

performance to also be factored into the bicycle's fitness computation (which would be useful since aerodynamics is a huge portion of top speed recumbent racing). Finally, the last major addition the MPCM could use is the ability to better fit riders to upright bicycles. Currently, rider's seat heights are fixed by the user's bicycle parameter configuration file with a horizontal sliding seat used to fit riders of varying sizes to the bicycle frame. In order to successfully fit a set of riders to a safety bike, the seat would likely need to be allowed to slide in the vertical direction, which is something the model is not setup to accommodate at this time.

Improving the genetic search platform on the other hand primarily revolves around investigating more design spaces and trying other tuners and parameter configurations. In the studies conducted in this work, the selection, cross-over and mutation operator values were restricted to a ceiling of 25%. This ceiling was arbitrarily set in the beginning of the experiments, and was maintained primarily because the resulting bicycle design solutions and their resulting runtimes were considered worthwhile enough to stop further investigation. However, while the solutions are adequate for a user's needs, more tuning experiments could be done to better characterize the solution landscape of the Patterson Control Model. This practice could also be expanded to other bicycle models (such as the Whipple model) in an attempt to compare and contrast it with the PCM, with the conclusions of this study helping to inform researchers of where to put their resources in future bicycle design related endeavours.

Bibliography

[1] Cycling timeline revised. `https://mjmatthewsdesign.wordpress.com/`. Accessed 22 May 2016.

[2] Jupyter notebook. `https://jupyter.org/`. Accessed 1 Feb 2017.

[3] *NIST/SEMATECH e-Handbook of Statistical Methods*. Accessed 20 January 2017.

[4] A short history of bicycles. Access 1 June 2016.

[5] J. K. A. L. Schwab and J. Nieuwendijk. On the design of a recumbent bicycle with a perspective on handling qualities. *ASME*, 2012.

[6] B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Oper. Res.*, 54(1):99–114, Jan. 2006.

[7] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, 1995.

[8] D. Beasley, D. R. Bull, and R. R. Martin. An overview of genetic algorithms: Part 1, fundamentals, 1993.

[9] C. J. Camacho and D. Gatchell. Successful discrimination of protein interactions. *Proteins*, 40:525537, 2003.

[10] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77–97, 2001.

[11] W. T. Dempster. Space requirements of the seated operator, geometrical, kinematic and mechanical aspects of the body with special reference to the limbs. 1955.

[12] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 2011.

[13] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[14] W. Erdmann. Geometry and inertia of the human body-review of research. *Acta of Bioengineering and Biomechanics*, 1:23–35, 1999.

[15] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Wiley Publishing, 4th edition, 2009.

[16] H. Gannett. *General Summary Showing the Rank of States by Ratios 1880*. 1880.

[17] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[18] J. M. P. A. R. J. D. G. Kooijman, J. P. Meijaard and A. L. Schwab1. A bicycle can be self-stable without gyroscopic or caster effects, April 2011.

[19] M. JK. Human control of a bicycle.

[20] V. S. C. C. Kozakov D, Clodfelter KH. Optimal clustering for detecting near-native conformations in protein docking.

[21] J. Lowell and H. D. McKell. The stability of bicycles. *American Journal of Physics*, 50(12):1106–1112, December 1982.

[22] J. K. M. M. Hubbard, R. Hess and D. L. Peterson. Human control of bicycle dynamics with experimental validation and implications for bike handling and design. *NSF Engineering Research and Innovation Conference*, 2011.

[23] C. C. E. C. T. D. C. J. McConville, J. T. and I. Kaleps. Anthropometric relationships of body and body segment moments of inertia. *Technical Report AFAMRL-TR-80-119, Air Force Aerospace Medical Research Laboratory, Wright-Patterson AFB, OH.*, 1980.

[24] J. P. Meijaard, J. M. Papadopoulos, A. Ruina, and A. L. Schwab. Linearized dynamics equations for the balance and steer of a bicycle: A benchmark and review. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 463(2084):1955–1982, August 2007.

[25] K. J. Moore JK, Hubbard M. A method for estimating physical properties of a combined bicycle and rider. 2009.

[26] R. Myers and E. R. Hancock. Empirical modelling of genetic algorithms. *Evolutionary Computation*, 9(4):461–493, 2001.

[27] M. Obitko. Introduction to genetic algorithms.

[28] W. B. Patterson. *The Lords of the Chainring.* W. B. Patterson, 2004.

[29] A. K. B. S. P. R. Hari Krisnan, V. Devanandh. Estimation of mass moment of inertia of human body, when bending forward, for the design of a self-transfer robotic facility. 2016.

[30] K. L. R. Mukesh and U. Selvakumar. Airfoil shape optimization using non-traditional optimization technique and its validation. Access 5 June 2016.

[31] A. L. Schwab and J. P. Meijaard. A review on bicycle dynamics and rider control. *Vehicle System Dynamics*, 51(7):1059–1090, 2013.

[32] K. Wloch and P. J. Bentley. Optimising the performance of a formula one car using a genetic algorithm. In *In Proceedings of Eighth International Conference on Parallel Problem Solving From Nature*, pages 702–711, 2004.

[33] M. R. Yeadon. The simulation of aerial movement-ii. a mathematical inertia model of the human body. *Journal of Biomechanics*, 23:67–74, 1990.

Appendix A

Configuration File Templates

A.1 Bicycle Configuration Template

```
# This is a template for bike parameter inputs. The order
# and capitalization of each parameter is not important,
# but the names of the parameters are.
#
# Additionally, you can specify the value for each variable
# in 1 of 3 ways:
#
# Option 1: Set the parameter equal to a specific value.
#
#  param = value
#
# Option 2: Set the parameter to a range of values between
#           x & y stepping by z.
#
#  param = x to y by z
#
# Option 3: Set the parameter to the values x, y, z.
#
#  param = [x, y, z]
#
# Additionally, you can include comments on their own own
# lines or inline.
#
# Note that the units per line below are just to tell the
```

```
# user what units each parameter values should be in.

# Bike Parameters
wheelbase           = 1.1 to 1.4 by 0.1       # [m]
fork_offset         = -0.075 to 0 by 0.01     # [m]
handlebar_radius    = 0.1 to 0.4 by 0.1       # [m]
front_wheel_radius  = 0.241 to 0.343 by 0.2   # [m]
rear_wheel_radius   = 0.241 to 0.343 by 0.2   # [m]
crank_radius        = [0.150, 0.165, 0.175]   # [m]
crank_x_offset      = 0.2 to 0.4 by 0.05      # [m]
crank_z_offset      = 0.4 to 0.6 by 0.05      # [m]
seat_height         = 0.2 to 0.8 by 0.1       # [m]
hip_angle           = 110 to 135 by 5         # [degrees]
headtube_angle      = 0 to 12 by 1            # [degrees]
frame_mass          = 5 to 20 by 5            # [kg]
crank_mass          = 1                       # [kg]
front_wheel_mass    = 2                       # [kg]
rear_wheel_mass     = 2                       # [kg]
```

A.2 Rider Configuration Template

```
# This is a template for rider parameter inputs. The order
# and capitalization of each parameter is not important,
# but the names of the parameters are.
#
# Additionally, you can specify the value for each variable
# in 1 way:
#
# Set the parameter equal to a specific value:
#
#     param = value
#
# Additionally, you can include comments on their own own lines
# or inline.
#
# Note that the units per line below are just to tell the user
# what units each parameter values should be in.

# Rider Parameters
rider_name    = Chris
rider_mass    = 60    # [kg]
head_diameter = 0.185 # [m]
```

```
torso_length  = 0.48  # [m]
torso_width   = 0.2   # [m]
torso_depth   = 0.2   # [m]
arm_length    = 0.5   # [m]
arm_diameter  = 0.08  # [m]
leg_length    = 1.0   # [m]
leg_diameter  = 0.12  # [m]
```

A.3 Double Rider Configuration File

This configuration file shows how to specify two different rider's for use with the Modified Patterson Control Model. In this example, one rider is *Chris* and the other rider is *Steve*. Note that the parsing of each rider in the file is based around the appearance of the `rider_name` assignment, so its important to keep the blocks separated as shown below.

```
# Rider Parameters for Chris
rider_name    = Chris
rider_mass    = 60      # [kg]
head_diameter = 0.185   # [m]
torso_length  = 0.48    # [m]
torso_depth   = 0.2     # [m]
torso_width   = 0.2     # [m]
arm_length    = 0.5     # [m]
arm_diameter  = 0.08    # [m]
leg_length    = 1.0     # [m]
leg_diameter  = 0.12    # [m]

# Rider Parameters for Steve
rider_name    = Steve
rider_mass    = 90      # [kg]
head_diameter = 0.2     # [m]
torso_length  = 0.58    # [m]
torso_depth   = 0.4     # [m]
torso_width   = 0.4     # [m]
arm_length    = 0.6     # [m]
arm_diameter  = 0.15    # [m]
leg_length    = 0.8     # [m]
leg_diameter  = 0.22    # [m]
```

A.4 Genetic Algorithm Configuration Template

```
# This is a template for a genetic algorithm config.
# The order and capitalization of each parameter is
# not important, but the names of the parameters are.
#
# Additionally, you can specify the value for each
# variable in 1 of 3 ways:
#
# Option 1: Set the parameter equal to a specific value.
#
#  param = value
#
# Option 2: Set the parameter to a range of values between
#           x & y stepping by z.
#
#  param = x to y by z
#
# Option 3: Set the parameter to the values x, y, z.
#
#  param = [x, y, z]
#
# Additionally, you can include comments on their own lines
# or inline.

# Genetic Algorithm Config Parameters

# Number of runs for each trial to average across
num_runs = 10.0

# Number of generations to run
generation_count = 1 to 2 by 1

# Size of each generation's population
population_size = 10 to 20 by 10

# % population to be selected from per iteration
selection_percentage = 5 to 10 by 5

# % population to be crossed-over per iteration
cross_over_percentage = 5 to 10 by 5

# % population to be mutated per iteration
mutation_percentage = 5 to 10 by 5
```

```
# Number of genes to involve during cross-over
cross_over_gene_count = 5

# Number of genes to involve during mutation
mutation_gene_count = 10
```

A.5 Partitioning Configuration Template

This is an example partitioning configuration file. Note that the attributes list specifies only a subset of the bicycle parameters outlined in Table 3.4, however, all may be listed if need be. Note that at least one attribute must be listed, otherwise the partitioning algorithm will not be able to correctly compute the distance between individuals in the bike design space.

```
# This is a template for a partitioning config. The order and
# capitalization of each parameter is not important, but the
# names of the parameters are.
#
# Note that there are 2 parameters that you can set in this config.
# They are the partitioning 'radius' and the set of 'attributes' to
# partition about.
#
# The following describes how you can set the values for each of
# these parameters in this config.
#
# Set the value for the radius by specifying a single value:
#  radius = value
#
# Set the attributes to partition about by specifying a set:
#  param = [x, y, z]
#
#  where x, y and z are valid attributes (note that no quotes are
#  needed here). Additionally, valid attributes include any of
#  those in the Modified Patterson Control Model.
#
# Additionally, you can include comments on their own own lines or inline.

# Partitioning Parameters
radius     = 5.0
attributes = [wheelbase, seat_height, crank_x_offset]
```

Appendix B

Bike Configurations

B.1 Cal Poly's Gemini Bicycle Frame Parameters

```
# Cal Poly's Gemini Bicycle Frame Parameters
wheelbase           = 1.15      # [m]
hip_angle           = 120       # [degrees]
headtube_angle      = 12        # [degrees]
crank_radius        = 0.175     # [m]
crank_x_offset      = 0.2       # [m]
crank_z_offset      = 0.6       # [m]
fork_offset         = -0.075    # [m]
seat_height         = 0.325     # [m]
handlebar_radius    = 0.2       # [m]
front_wheel_radius  = 0.28      # [m]
rear_wheel_radius   = 0.28      # [m]
frame_mass          = 10        # [kg]
crank_mass          = 1         # [kg]
front_wheel_mass    = 2         # [kg]
rear_wheel_mass     = 2         # [kg]
```

B.2 Cervelo R3 Team Bicycle Frame Parameters

```
# Cervelo R3 Team Bicycle Frame Parameters
wheelbase           = 0.94      # [m]
hip_angle           = 120        # [degrees]
```

```
headtube_angle      = 17        # [degrees]
crank_radius        = 0.175     # [m]
crank_x_offset      = -0.58     # [m]
crank_z_offset      = 0.27      # [m]
fork_offset         = 0.0       # [m]
seat_height         = 1.05      # [m]
handlebar_radius    = 0.2       # [m]
front_wheel_radius  = 0.336     # [m] (700c + tire)
rear_wheel_radius   = 0.336     # [m] (700c + tire)
frame_mass          = 5         # [kg]
fairing_mass        = 0         # [kg]
crank_mass          = 1         # [kg]
front_wheel_mass    = 2         # [kg]
rear_wheel_mass     = 2         # [kg]
```

B.3 Parameter Space for Brute Force Search – Gemini Bike Frame

```
# Design space including the Gemini bicycle frame
wheelbase           = 1 to 2 by 0.15            # [m]
hip_angle           = 100 to 130 by 10          # [degrees]
headtube_angle      = 6 to 18 by 3              # [degrees]
crank_radius        = [0.165, 0.170, 0.175]     # [m]
crank_x_offset      = 0 to 0.4 by 0.1           # [m]
crank_z_offset      = 0.2 to 0.8 by 0.2         # [m]
fork_offset         = [0, -0.03, -0.06, -0.075] # [m]
seat_height         = 0.3 to 0.5 by 0.025       # [m]
handlebar_radius    = 0.1 to 0.4 by 0.1         # [m]
front_wheel_radius  = [0.23, 0.28, 0.36]        # [m]
rear_wheel_radius   = [0.23, 0.28, 0.36]        # [m]
frame_mass          = 10                        # [kg]
crank_mass          = 1                         # [kg]
front_wheel_mass    = 2                         # [kg]
rear_wheel_mass     = 2                         # [kg]
```

B.4 Parameter Space for Recumbent Genetic Search

```
# Recumbent design space including the Gemini bicycle frame
```

```
wheelbase           = 1 to 2 by 0.15            # [m]
hip_angle           = 100 to 150 by 10          # [degrees]
headtube_angle      = 0 to 18 by 2              # [degrees]
crank_radius        = [0.165, 0.170, 0.175]     # [m]
crank_x_offset      = -0.5 to 0.5 by 0.05       # [m]
crank_z_offset      = 0.2 to 0.8 by 0.05        # [m]
fork_offset         = [0, -0.03, -0.06, -0.075] # [m]
seat_height         = 0.2 to 0.6 by 0.025       # [m]
handlebar_radius    = 0.1 to 0.4 by 0.05        # [m]
front_wheel_radius  = [0.23, 0.28, 0.36]        # [m]
rear_wheel_radius   = [0.23, 0.28, 0.36]        # [m]
frame_mass          = 10                        # [kg]
fairing_mass        = 0                         # [kg]
crank_mass          = 1                         # [kg]
front_wheel_mass    = 2                         # [kg]
rear_wheel_mass     = 2                         # [kg]
```

B.5 Parameter Space for Safety Bike Genetic Search

```
# Recumbent design space including the Cervelo R3 safety bicycle frame
wheelbase           = 1 to 2 by 0.15            # [m]
hip_angle           = 100 to 150 by 10          # [degrees]
headtube_angle      = 0 to 20 by 2              # [degrees]
crank_radius        = [0.165, 0.170, 0.175]     # [m]
crank_x_offset      = -1.0 to 1.0 by 0.05       # [m]
crank_z_offset      = 0.2 to 0.8 by 0.05        # [m]
fork_offset         = [0, -0.03, -0.06, -0.075] # [m]
seat_height         = 0.2 to 1.2 by 0.025       # [m]
handlebar_radius    = 0.1 to 0.4 by 0.05        # [m]
front_wheel_radius  = [0.23, 0.28, 0.336]       # [m]
rear_wheel_radius   = [0.23, 0.28, 0.336]       # [m]
frame_mass          = 5                         # [kg]
fairing_mass        = 0                         # [kg]
crank_mass          = 1                         # [kg]
front_wheel_mass    = 2                         # [kg]
rear_wheel_mass     = 2                         # [kg]
```

Appendix C

Rider Configuration Trials

C.1 Chris Hunt's Rider Parameters

This configuration file outlines all of the rider parameters used to model Chris Hunt.

```
# Chris Hunt's Rider Parameters
rider_name    = Chris
rider_mass    = 60      # [kg]
head_diameter = 0.185   # [m]
torso_length  = 0.48    # [m]
torso_depth   = 0.2     # [m]
torso_width   = 0.2     # [m]
arm_length    = 0.5     # [m]
arm_diameter  = 0.08    # [m]
leg_length    = 1.0     # [m]
leg_diameter  = 0.12    # [m]
```

Appendix D

Genetic Operator Configurations

D.1 Base Genetic Search Configuration

The following is the base genetic algorithm configuration file used in the main validation experiments for the genetic search platform.

```
# Number of runs for each trial to average across
num_runs = 20.0

# Number of generations to run
generation_count = 2 to 8 by 2

# Size of each generation's population
population_size = 100 to 500 by 100

# % population to be selected from per iteration
selection_percentage = 5 to 25 by 5

# % population to be crossed-over per iteration
cross_over_percentage = 5 to 25 by 5

# % population to be mutated per iteration
mutation_percentage = 5 to 25 by 5

# Number of genes to involve during cross-over
cross_over_gene_count = 8
```

```
# Number of genes to involve during mutation
mutation_gene_count = 8
```

Appendix E

Control Sensitivity Trials

E.1 Control Sensitivity Values for Chris Hunt Riding Cal Poly's Gemini Frame

Table E.1 enumerates the control sensitivity values for the rider specified in Appendix C.1 riding Cal Poly's Gemini frame (see Appendix B.1) from 0 to 25 m/s.

E.2 Control Sensitivity Values for Chris Hunt Riding Cervelo's 2012 55cm R3 Team Frame

Table E.2 enumerates the control sensitivity values for the rider specified in Appendix C.1 riding Cervelo's 2012 R3 Team frame (see Appendix B.2) from 0 to 25 m/s.

Speed [m/s]	Control Sensitivity
0	0.0
1	5.2874126319688814
2	9.354120785527682
3	11.76725941081217
4	12.798612526935305
5	12.93402876510182
6	12.576652422290532
7	11.9857701616695
8	11.308517943958789
9	10.62209029950619
10	9.964085088347053
11	9.350782989946877
12	8.787386266179801
13	8.273564580645214
14	7.806404030623976
15	7.381950634303657
16	6.995999896484056
17	6.6444840070832525
18	6.3236457485945685
19	6.030100891080484
20	5.760843907430813
21	5.513226483676934
22	5.284924533503191
23	5.073901927187384
24	4.878375059368508
25	4.69678016551565

Table E.1: Control Sensitivity for Chris Hunt on Cal Poly's Gemini Frame.

Speed [m/s]	Control Sensitivity
0	0.0
1	2.317841464743265
2	4.304454705511157
3	5.769600764672873
4	6.695333078648078
5	7.173311173885147
6	7.328172920907239
7	7.2718171888559135
8	7.088329936036018
9	6.835044164166083
10	6.548771056446919
11	6.25214096975315
12	5.958519031826883
13	5.675413804032869
14	5.4067138819541
15	5.154112774961068
16	4.91800169083479
17	4.698023105053452
18	4.493411116204026
19	4.303198721807412
20	4.126342326175626
21	3.9617948983646185
22	3.8085473726739907
23	3.665650515910766
24	3.5322248961374347
25	3.4074637233886675

Table E.2: Control Sensitivity for Chris Hunt on Cervelo's 2012 R3 Team Frame.

Appendix F

Computing the Rider's Geometry

This chapter explains how the rider's body is fit to the bicycle frame, including how each of their body segment's is located and rotated. This section assumes that the bicycle is operating in the same XZ-Plane as used by the Modified Patterson Control Model, and the equations listed will utilize all of the parameter inputs defined in Table 3.1 and Figure 3.1. Finally, the body model being positioned is the same model as described in Section G.1, and as such, all of that model's assumptions apply here as well. With all that in mind, the following sections break down how each segment of the rider's body is positioned relative to the bicycle.

F.1 Accommodating Rider Body Thickness

This model begins locating the rider on the bicycle by positioning them such that the center of their legs goes through the center of the bicycle's cranks, and so that the rider is sitting in a seat that is behind those cranks at the user specified seat height. However, since the rider has a thickness specified by the user (both a torso and a leg thickness), placing the rider in their seat is not completely straightforward. Instead, this model generally works by computing where the rider would be if they had zero body thickness, and then adds in the offsets to accommodate for their actual body thickness. This manifests itself as an offset of the rider's hip from the bicycle seat,

with many equations utilizing the following two variables:

$$hip\_center\_delta_x = \left(\frac{torso\_depth}{2}\right) Cos(\alpha - 90° + \theta)$$

$$hip\_center\_delta_z = \left(\frac{torso\_depth}{2}\right) Sin(\alpha - 90° + \theta) + Cos(\theta)\left(\frac{leg\_diameter}{2}\right)$$

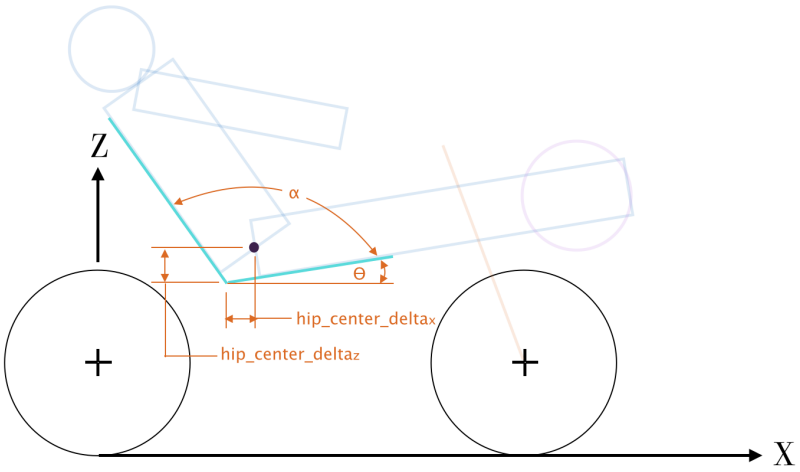


Figure F.1: Calculating the rider's hip center.

which specify the offset of the body's hips in the X and Z locations as shown in Figure F.1. Additionally, computing the angle between the X axis and the bottom of the seat is important in many calculations. This angle, known as θ is complex to calculate because the rider's leg length and seat height are fixed, and as a result the rider's seat must be rotated to accommodate for their leg's thickness as shown in Figure F.2. In that figure, θ' is defined as if the seat bottom were of infinite length and went through the crank center, while θ is defined as the proper angle between the seat and the X axis in order to make the rider's leg center-line go through the crank center, as defined in the following equations:

$$\theta' = ArcSin\left(\frac{C_z - H_z}{leg\_length - C_r}\right)$$

$$\theta = ArcSin\left(\frac{C_z - (\frac{leg\_diameter}{2})Cos(\theta') - H_z}{leg\_length - C_r}\right)$$

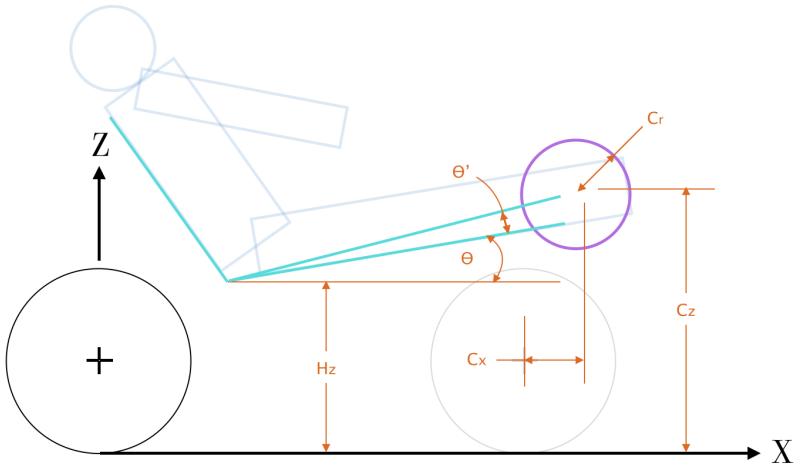


Figure F.2: Calculating the seat angle while compensating for rider body thickness.

F.2 Locating the Rider's Head

The rider's head is located atop the rider's torso and as such is simple to locate, as shown in Figure F.3, with the equations of the rider's head's center of mass being given by:

$$\phi = 180^\circ - \alpha - \theta$$

$$x' = \left(torso\_length + \frac{head\_diameter}{2}\right) Cos(\phi) - hip\_center\_delta_x$$

$$z' = \left(torso\_length + \frac{head\_diameter}{2}\right) Sin(\phi) + hip\_center\_delta_z$$

$$head\_cg_x = H_x - x'$$

$$head\_cg_z = H_z + z'$$

F.3 Locating the Rider's Torso

The rider's torso is assumed to be a cuboid as shown in Figure F.4, with the equations specifying its center of mass being given by:

$$\phi = 180^\circ - \alpha - \theta$$

$$torso\_cg_x = H_x + hip\_center\_delta_x - \left(\frac{torso\_length}{2}\right) Cos(\phi)$$

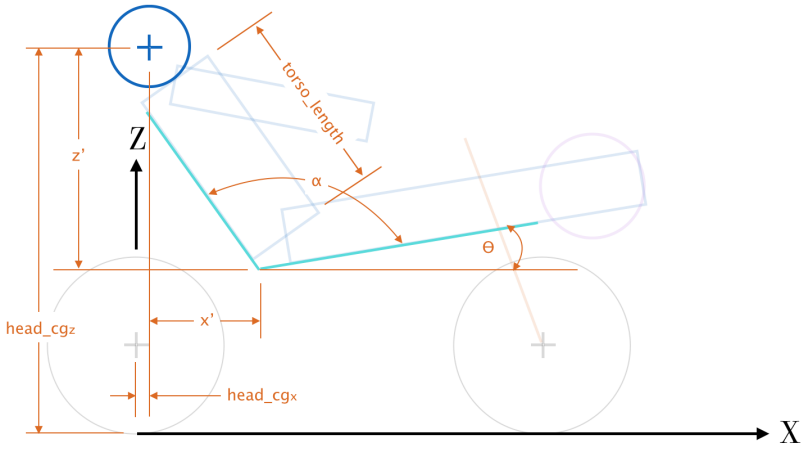


Figure F.3: Calculating the location of the rider's head's center of gravity.

$$torso\_cg_z = H_z + hip\_center\_delta_z + \left(\frac{torso\_length}{2}\right) Sin(\phi)$$

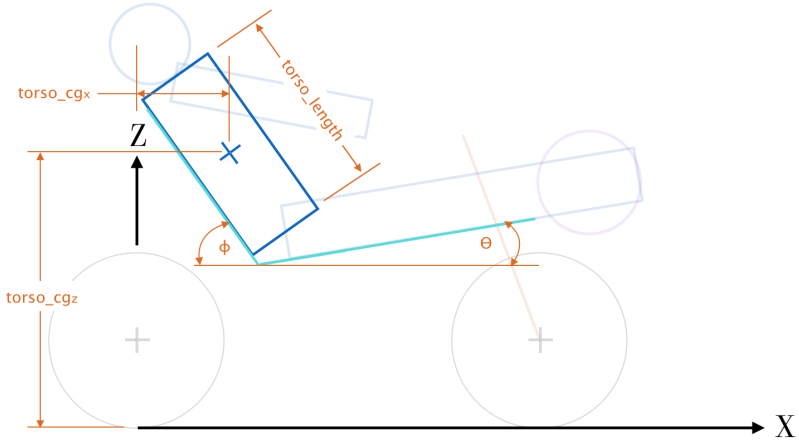


Figure F.4: Calculating the location of the rider's torso's center of gravity.

F.4 Locating the Rider's Legs

The rider's legs are assumed to each be a single cylinder, with both going through the crank center with the legs stopping at the farthest point away from the rider on the crank circle. Additionally, the rider's legs are assumed to start at their hip center as shown in Figure F.5, with the equations specifying each leg's center of mass being given by:

$$leg\_start_x = H_x + hip\_center\_delta_x$$

$$leg\_start_z = H_z + hip\_center\_delta_z$$

$$leg\_end_x = H_x + leg\_length * Cos(\theta) + hip\_center\_delta_x$$

$$leg\_end_z = H_z + leg\_length * Sin(\theta) + hip\_center\_delta_z$$

$$leg\_cg_x = leg\_start_x - \frac{leg\_start_x - leg\_end_x}{2}$$

$$leg\_cg_z = leg\_start_z - \frac{leg\_start_z - leg\_end_z}{2}$$

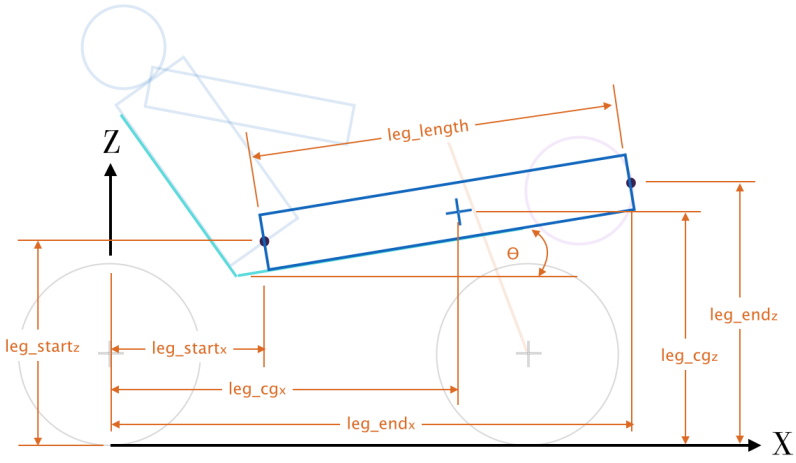


Figure F.5: Calculating the location of the rider's leg's center of gravity.

F.5 Locating the Rider's Arms

Since the rider's arms are the last component to be oriented, they are able to utilize many of the variables which had been computed to locate the other body segments. As such, many of the variables listed in Figure F.6 and the subsequent equations are computed in previous sections.

$$arm\_start_x = H_x + hip\_center\_delta_x - torso\_length * Cos(\phi)$$

$$arm\_start_z = H_z + hip\_center\_delta_z + torso\_length * Sin(\phi)$$

$$\zeta = ArcTan\left(\frac{arm\_start_z - fork\_top_z}{fork\_top_x - arm\_start_x}\right)$$

$$arm\_end_x = arm\_start_x + arm\_length * Cos(\zeta)$$

$$arm\_end_z = arm\_start_z - arm\_length * Sin(\zeta)$$

$$arm\_cg_x = Cos(\zeta) * \frac{arm\_length}{2} + arm\_start_x$$

$$arm\_cg_z = -Sin(\zeta) * \frac{arm\_length}{2} + arm\_start_z$$

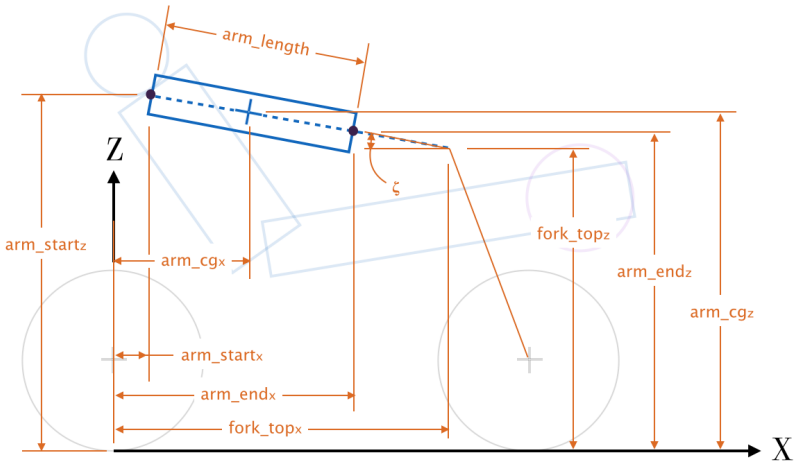


Figure F.6: Calculating the location of the rider's arm's center of gravity.

Appendix G

Rider Inertia Equations

This chapter assumes that the rider is being modeled using the body model explained in Section 3.1.3 and that the center of gravity locations and masses for each of the rider's segments have been computed already. Each segment's inertia is computed about the global X axis used in the Patterson Control Model (the axis that passes through the wheel's contact points with the ground plane), and the process of computing these inertia values follows the steps outlined in Section G.1.

G.1 Computing the Inertia of a Body about an Axis

This section outlines the general steps required to compute the Moment of Inertia (and subsequently the Radius of Gyration) of a body about an arbitrary axis. Note that this example focuses on computing the Moment of Inertia I about the X axis, and this is represented by I_{xx}. Additionally, the Radius of Gyration will also be about the X axis, and is represented by K_{xx}.

1) Establish the Global Coordinate System and call it XYZ.

2) Establish a local coordinate system for each member located at the member's

mass center and call it $x_i y_i z_i$.

3) Compute the total body's center of mass location relative to the desired axis:

$$r_{body} = \frac{\sum m_i r_i}{m_{body}}$$

where r_i is the position vector to the centroid of each segment that makes up the body, m_i is the mass of each segment being analyzed, m_{body} is the total mass of the body and r_{body} is the position vector to the center of mass of the body.

4) Compute each segment's local moment of inertia using the ideal definitions of inertia for the segment about its local mass center. Note that this assumes that the segment's mass moment of inertia is already cataloged due to it being a common shape such as a sphere, cylinder or cube. If this is not the case then you will need to compute the inertia value manually about the body's center of mass. Mass moment of inertia equations for common shapes fixed at their mass centers include:

$$Cuboid \Rightarrow \frac{m}{12}\begin{bmatrix} y^2+z^2 & 0 & 0 \\ 0 & x^2+z^2 & 0 \\ 0 & 0 & x^2+y^2 \end{bmatrix}$$

$$Cylinder \Rightarrow I_{xx} = I_{yy} = \frac{1}{4}m*r^2 + \frac{1}{12}m*l^2, I_{zz} = \frac{1}{2}m*r^2$$

$$Sphere \Rightarrow I_{xx} = I_{yy} = I_{zz} = \frac{2}{5}m*r^2$$

Finally, define each segment's local inertia matrix using the symbol J.

5) Define a local set of unit vectors to be used later for rotation of each segment.

- Let $\hat{z}$ be along the length of the member.
- Let $\hat{y}$ be parallel to Y.
- Let $\hat{z}$ be defined by the right-hand-rule.

6) Define a rotation matrix for each segment by dotting its unit vector with the Global Coordinate System:

$$R_i = \begin{bmatrix} X*\hat{x_i} & X*\hat{y_i} & X*\hat{z_i} \\ Y*\hat{x_i} & Y*\hat{y_i} & Y*\hat{z_i} \\ Z*\hat{x_i} & Z*\hat{y_i} & Z*\hat{z_i} \end{bmatrix}$$

where R_i represents a rotation matrix for each individual body segment.

7) Now, rotate each segment's inertia matrix to align with the Global Coordinate System:

$$I_i = R_i * J_i * R_i^T$$

where R_i is the segment's rotation matrix, J_i is the segment's local inertia matrix and I_i is the resulting rotated local inertia matrix for each segment that is now aligned with the Global Coordinate System.

8) Translate each segment's rotated moment of inertia so that it is referencing the rider's center of mass using the parallel-axis theorem.

$$I_i^* = I_i + m_i \begin{bmatrix} d_y^2 + d_z^2 & -d_x d_y & -d_x d_z \\ -d_x d_y & d_z^2 + d_x^2 & -d_y d_z \\ -d_x d_z & -d_y d_z & d_x^2 + d_y^2 \end{bmatrix}$$

where:

- I_i = the segment's rotated moment of inertia.
- m_i = the mass of the segment.
- d_x = the x distance from the segment's center of mass to the body's center of mass.
- d_y = the y distance from the segment's center of mass to the body's center of mass.
- d_z = the z distance from the segment's center of mass to the body's center of mass.

9) Sum all segment's moments of inertia to get the body's moment of inertia.

$$I_{xx} = \sum I_i^*$$

This moment of inertia about a given axis can then be converted into the body's Radius of Gyration about that axis via:

$$K_{xx} = \sqrt{\frac{I_{xx}}{m_{body}}}$$

This process was completed for each of the components of the rider's body with each body segment using the unit vectors shown in Figure G.1, with the resulting radius of gyration being used as an input into the core Patterson Control Model equations.

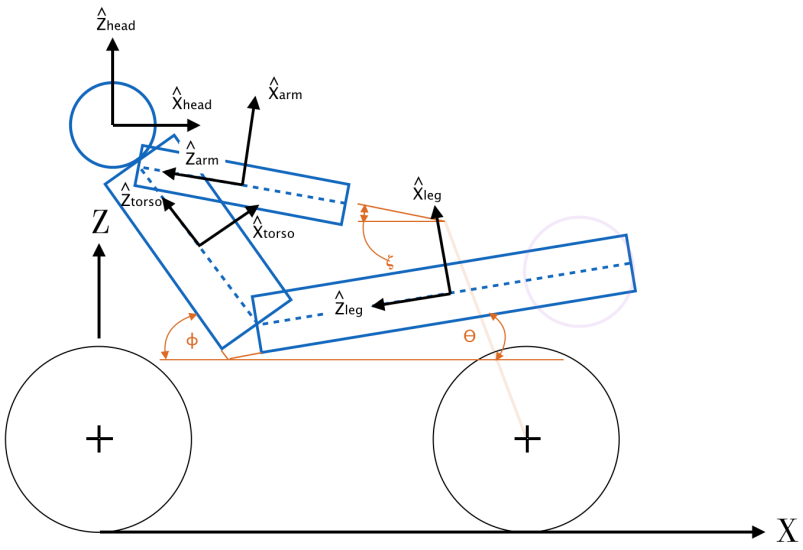


Figure G.1: Unit vectors for each of the rider's body segments.

G.2 Computing the Inertia of the Rider's Head

Since the rider's head is a sphere of constant density, its moment of inertia values are the same in all 3 global axes, namely:

$$I_{xx} = I_{yy} = I_{zz} = \frac{2}{5} m * r^2$$

As such, this inertia need only be computed and then translated to be about the wheel contact patch using the parallel-axis theorem:

$$I^*_{xx}\_head = I_{xx} + m * dx^2$$
$$I^*_{xx}\_head = \frac{2}{5} m_{head} * r^2_{head} + m_{head} * head\_cg^2_z$$

G.3 Computing the Inertia of the Rider's Torso

The rider's torso is represented as a cuboid which has the following principle moments of inertia:

$$J = \frac{m}{12}\begin{bmatrix} a^2+l^2 & 0 & 0 \\ 0 & b^2+l^2 & 0 \\ 0 & 0 & a^2+b^2 \end{bmatrix}$$

where $a = torso\_width$, $b = torso\_depth$ and $l = torso\_length$. Next, these inertias must be rotated using a rotation matrix given by the following equation:

$$R_{torso} = \begin{bmatrix} X * \hat{x_{torso}} & X * \hat{y_{torso}} & X * \hat{z_{torso}} \\ Y * \hat{x_{torso}} & Y * \hat{y_{torso}} & Y * \hat{z_{torso}} \\ Z * \hat{x_{torso}} & Z * \hat{y_{torso}} & Z * \hat{z_{torso}} \end{bmatrix}$$

$$X = (1,0,0) \qquad \hat{x_{torso}} = (Cos(90^\circ - \phi), 0, Sin(90^\circ - \phi))$$
$$Y = (0,1,0) \qquad \hat{y_{torso}} = (0,1,0)$$
$$Z = (0,0,1) \qquad \hat{z_{torso}} = (-Cos(\phi), 0, Sin(\phi))$$

resulting in the following rotational matrix:

$$R_{torso} = \begin{bmatrix} Sin(\phi) & 0 & -Cos(\phi) \\ 0 & 1 & 0 \\ Cos(\phi) & 0 & Sin(\phi) \end{bmatrix}$$

Applying the rotation to the local inertial tensor results in:

$$I_{torso} = R_{torso} * J_{torso} * R_{torso}^T$$

$$\text{I`torso} = \begin{bmatrix} Sin(\phi) & 0 & -Cos(\phi) \\ 0 & 1 & 0 \\ Cos(\phi) & 0 & Sin(\phi) \end{bmatrix} \frac{m_{torso}}{12} \begin{bmatrix} a^2+l^2 & 0 & 0 \\ 0 & b^2+l^2 & 0 \\ 0 & 0 & a^2+b^2 \end{bmatrix} \begin{bmatrix} Sin(\phi) & 0 & Cos(\phi) \\ 0 & 1 & 0 \\ -Cos(\phi) & 0 & Sin(\phi) \end{bmatrix}$$

which results in the final, rotated but unshifted intertial tensor:

$$I_{torso} = \frac{m_{torso}}{12} \begin{bmatrix} (a^2+l^2)Sin(\phi)^2 & 0 & (a^2+l^2)Sin(\phi)Cos(\phi) - (a^2+b^2)Sin(\phi)Cos(\phi) \\ 0 & (b^2+l^2) & 0 \\ (a^2+l^2)Sin(\phi)Cos(\phi) - (a^2+b^2)Sin(\phi)Cos(\phi) & 0 & (a^2+l^2)Cos(\phi)^2 + (a^2+b^2)Sin(\phi)^2 \end{bmatrix}$$

This inertia now must be shifted to be about the X-axis via the parallel-axis theorem:

$$I^*_{torso} = I_{torso} + m_{torso} \begin{bmatrix} d_y^2 + d_z^2 & -d_x d_y & -d_x d_z \\ -d_x d_y & d_z^2 + d_x^2 & -d_y d_z \\ -d_x d_z & -d_y d_z & d_x^2 + d_y^2 \end{bmatrix}$$

$$dx = torso\_cg_x$$

$$dy = 0$$

$$dz = torso\_cg_z$$

$$I^*_{torso} = I_{torso} + m_{torso} \begin{bmatrix} (torso\_cg_z)^2 & 0 & -torso\_cg_x * torso\_cg_z \\ 0 & (torso\_cg_x)^2 + (torso\_cg_z)^2 & 0 \\ -torso\_cg_x * torso\_cg_z & 0 & (torso\_cg_x)^2 \end{bmatrix}$$

However, since we only are concerned with I˙xx, we can simplify this to:

$$I^*_{torso\_xx} = I_{torso\_xx} + m_{torso}(torso\_cg_z)^2$$

G.4 Computing the Inertia of the Rider's Leg

The rider's legs are represented as cylinders which have the following principle moments of inertia:

$$J_{xx} = J_{yy} = \frac{1}{4} m * r^2 + \frac{1}{12} m * l^2$$

$$J_{zz} = \frac{1}{2} m * r^2$$

$$r = \frac{leg\_diameter}{2}$$

$$R_{leg} = \begin{bmatrix} X * \hat{x_{leg}} & X * \hat{y_{leg}} & X * \hat{z_{leg}} \\ Y * \hat{x_{leg}} & Y * \hat{y_{leg}} & Y * \hat{z_{leg}} \\ Z * \hat{x_{leg}} & Z * \hat{y_{leg}} & Z * \hat{z_{leg}} \end{bmatrix}$$

$$X = (1, 0, 0) \qquad \hat{x_{leg}} = (-Sin(\theta), 0, Cos(\theta))$$

$$Y = (0, 1, 0) \qquad \hat{y_{leg}} = (0, 1, 0)$$

$$Z = (0, 0, 1) \qquad \hat{z_{leg}} = (-Cos(\theta), 0, -Sin(\theta))$$

resulting in the following rotational matrix:

$$R_{leg} = \begin{bmatrix} -Sin(\theta) & 0 & -Cos(\theta) \\ 0 & 1 & 0 \\ Cos(\theta) & 0 & -Sin(\theta) \end{bmatrix}$$

Applying the rotation to the local inertial tensor results in:

$$I_{leg} = R_{leg} * J_{leg} * R_{leg}^T$$

$$\text{I'leg} = \begin{bmatrix} -Sin(\theta) & 0 & -Cos(\theta) \\ 0 & 1 & 0 \\ Cos(\theta) & 0 & -Sin(\theta) \end{bmatrix} \begin{bmatrix} 14m_{leg} * r^2 + \frac{1}{12}m_{leg} * l^2 & 0 & 0 \\ 0 & \frac{1}{4}m_{leg} * r^2 + \frac{1}{12}m_{leg} * l^2 & 0 \\ 0 & 0 & \frac{1}{2}m_{leg} * r^2 \end{bmatrix} \begin{bmatrix} -Sin(\theta) & 0 & Cos(\theta) \\ 0 & 1 & 0 \\ -Cos(\theta) & 0 & -Sin(\theta) \end{bmatrix}$$

which results in the final, rotated but unshifted intertial tensor:

$$I_{leg} = \frac{m_{leg}}{12} \begin{bmatrix} (3r^2 + l^2)Sin(\theta)^2 + 6r^2Cos(\theta)^2 & 0 & (3r^2 - l^2)Sin(\theta)Cos(\theta) \\ 0 & 3r^2 + l^2 & 0 \\ (3r^2 - l^2)Sin(\theta)Cos(\theta) & 0 & (3r^2 + l^2)Cos(\theta)^2 + 6r^2Sin(\theta)^2 \end{bmatrix}$$

This inertia now must be shifted to be about the X-axis via the parallel-axis theorem:

$$I_{leg}^* = I_{leg} + m_{leg} \begin{bmatrix} d_y^2 + d_z^2 & -d_x d_y & -d_x d_z \\ -d_x d_y & d_z^2 + d_x^2 & -d_y d_z \\ -d_x d_z & -d_y d_z & d_x^2 + d_y^2 \end{bmatrix}$$

$$dx = leg\_cg_x$$

$$dy = \frac{torso\_width}{2} - \frac{leg\_diameter}{2}$$

$$dz = leg\_cg_z$$

Note that dy is not zero in this case because the two legs are offset from the rider's sagittal plane, and as such each leg exhibits extra inertia about the global X axis. In this model we are assuming that the outside of the rider's leg is aligned with the outside edge of their torso, and we are then finding the center-line of that leg relative to the rider's sagittal plane. Additionally, since we only are concerned with I_{xx}, we can simplify this equation to:

$$I^*_{leg\_xx} = I_{leg\_xx} + m_{leg}\left[\left(\frac{torso\_width}{2} - \frac{leg\_diameter}{2}\right)^2 + leg\_cg_z^2\right]$$

G.5 Computing the Inertia of the Rider's Arm

The rider's arms are represented as cylinders which have the following principle moments of inertia:

$$J_{xx} = J_{yy} = \frac{1}{4}m * r^2 + \frac{1}{12}m * l^2$$

$$J_{zz} = \frac{1}{2}m * r^2$$

$$r = \frac{arm\_diameter}{2}$$

$$R_{arm} = \begin{bmatrix} X * \hat{x_{arm}} & X * \hat{y_{arm}} & X * \hat{z_{arm}} \\ Y * \hat{x_{arm}} & Y * \hat{y_{arm}} & Y * \hat{z_{arm}} \\ Z * \hat{x_{arm}} & Z * \hat{y_{arm}} & Z * \hat{z_{arm}} \end{bmatrix}$$

$$X = (1,0,0) \qquad \hat{x_{arm}} = (Sin(\eta), 0, Cos(\eta))$$

$$Y = (0,1,0) \qquad \hat{y_{arm}} = (0,1,0)$$

$$Z = (0,0,1) \qquad \hat{z_{arm}} = (Cos(\eta), 0, Sin(\eta))$$

resulting in the following rotational matrix:

$$R_{arm} = \begin{bmatrix} Sin(\eta) & 0 & Cos(\eta) \\ 0 & 1 & 0 \\ Cos(\eta) & 0 & Sin(\eta) \end{bmatrix}$$

Applying the rotation to the local inertial tensor results in:

$$I_{arm} = R_{arm} * J_{arm} * R_{arm}^T$$

$$\text{I`arm} = \begin{bmatrix} Sin(\eta) & 0 & Cos(\eta) \\ 0 & 1 & 0 \\ Cos(\eta) & 0 & Sin(\eta) \end{bmatrix} \frac{m}{12} \begin{bmatrix} 3r^2 + l^2 & 0 & 0 \\ 0 & 3r^2 + l^2 & 0 \\ 0 & 0 & 6r^2 \end{bmatrix} \begin{bmatrix} Sin(\eta) & 0 & Cos(\eta) \\ 0 & 1 & 0 \\ Cos(\eta) & 0 & Sin(\eta) \end{bmatrix}$$

which results in the final, rotated but unshifted intertial tensor:

$$I_{arm} = \frac{m_{arm}}{12}\begin{bmatrix} (3r^2+l^2)Sin(\eta)^2 + 6r^2Cos(\eta)^2 & 0 & (9r^2-l^2)Sin(\eta)Cos(\eta) \\ 0 & 3r^2+l^2 & 0 \\ (3r^2-l^2)Sin(\eta)Cos(\eta) & 0 & (3r^2+l^2)Cos(\eta)^2 + 6r^2Sin(\eta)^2 \end{bmatrix}$$

This inertia now must be shifted to be about the X-axis via the parallel-axis theorem:

$$I^*_{arm} = I_{arm} + m_{arm}\begin{bmatrix} d_y^2+d_z^2 & -d_xd_y & -d_xd_z \\ -d_xd_y & d_z^2+d_x^2 & -d_yd_z \\ -d_xd_z & -d_yd_z & d_x^2+d_y^2 \end{bmatrix}$$

$$dx = arm\_cg_x$$

$$dy = \frac{torso\_width}{2} - \frac{arm\_diameter}{2}$$

$$dz = arm\_cg_z$$

Note that dy is not zero in this case because the two arms are offset from the rider's sagittal plane, and as such each arm exhibits extra inertia about the global X axis. In this model we are assuming that the outside of the rider's arm is aligned with the outside edge of their torso, and we are then finding the center-line of that arm relative to the rider's sagittal plane. Additionally, since we only are concerned with I_{xx}, we can simplify this equation to:

$$I^*_{arm\_xx} = I_{arm\_xx} + m_{arm}\left[\left(\frac{torso\_width}{2} - \frac{arm\_diameter}{2}\right)^2 + arm\_cg_z^2\right]$$

G.6 Computing the Overall Radius of Gyration

Once the moments of inertia of each of the rider's body segments, and each of the bike frame's tubes has been computed the total radius of gyration of the bicycle can be calculated using the following equation:

This moment of inertia about a given axis can then be converted into the body's Radius of Gyration about that axis via:

$$K_{xx} = \sqrt{\frac{\sum I_{i\_xx}}{m_{total}}}$$

Additionally, this is where the observation that some rider limbs and frame tubes are doubled (two legs, 2 seat stays, etc.), and their inertias should be doubled in this summation as well to take that into account. The resulting radius of gyration can then be fed directly into the Patterson Control Model (which the Modified Patterson Control Model wraps) in order to compute the Control Spring and Control Sensitivity curves for a specific bicycle/rider configuration.

Appendix H

Computing the Frame

Note that this section uses all of the variables expressed in the Modified Patterson Control Model (reference Table 3.1 and Figure 3.1). As with the MPCM, the bicycle calculations will lie in the XZ-Plane, with Y being orthogonal to the other two axes in accordance with the right-hand-rule. Finally, this section assumes that the rider's location has already been computed, and as such we know the seat's location in the X and Z axes (expressed as H_x and H_z, respectively).

H.1 Computing Seat Stay Location

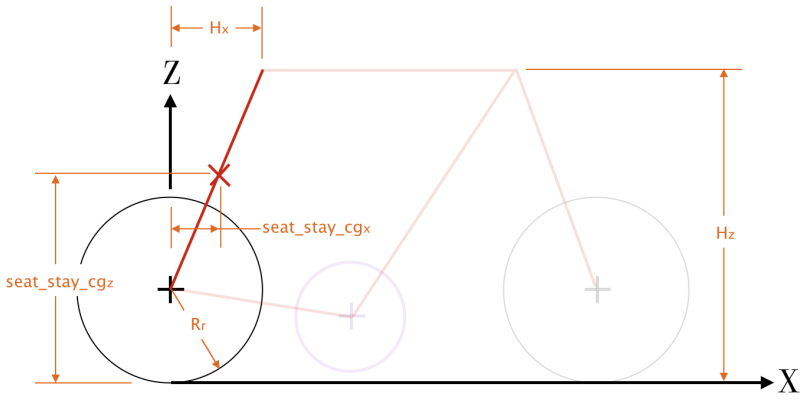


Figure H.1: Calculating the location of the frame's seat stay center of gravity.

$$seat\_stay\_cg_x = \frac{H_x}{2}$$

$$seat\_stay\_cg_z = \frac{(H_z - R_r)}{2} + R_r$$

$$seat\_stay_{len} = \sqrt{(H_z - R_r)^2 + H_x^2}$$

H.2 Computing the Chain Stay Location

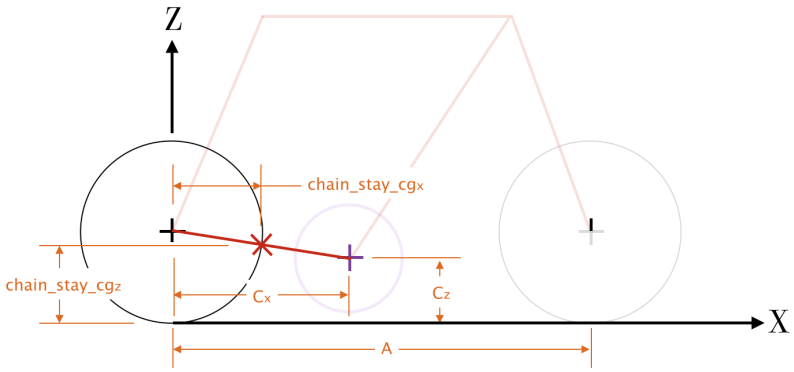


Figure H.2: Calculating the location of the frame's chain stay center of gravity.

$$chain\_stay\_cg_x = \frac{(A + C_x)}{2}$$

$$chain\_stay\_cg_z = \frac{(C_z - R_r)}{2} + R_r$$

$$chain\_stay_{len} = \sqrt{(A + C_x)^2 + (C_z - R_r)^2}$$

H.3 Computing the Fork Location

To begin, we will compute the bottom of the fork since it is known based on the model inputs.

$$fork\_slope = tan(90° + \beta)$$

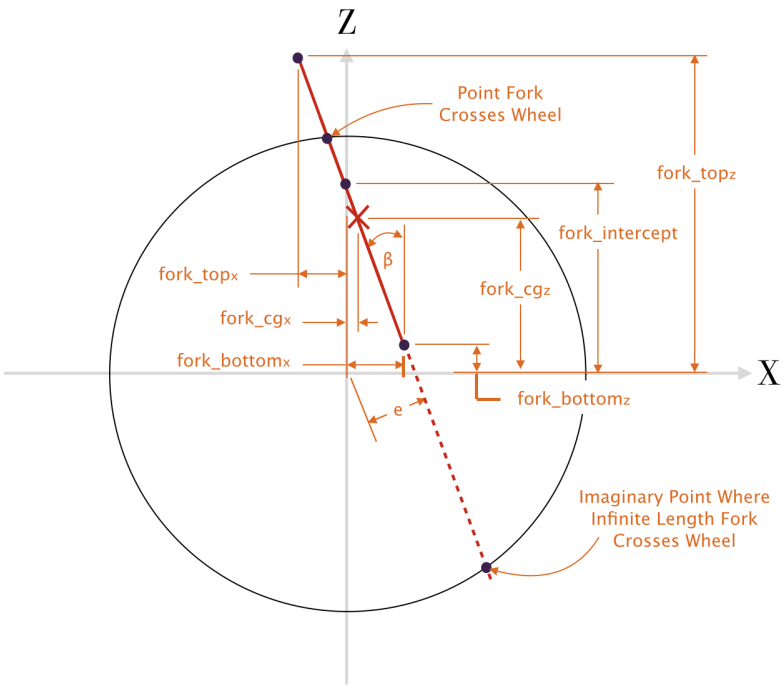


Figure H.3: Calculating the location of the frame's fork center of gravity.

$$fork\_bottom_x = A + e * Cos(\beta)$$

$$fork\_bottom_z = R_f + e * Sin(\beta)$$

$$fork\_intercept = fork\_bottom_z - fork\_slope * fork\_bottom_x$$

Now that we know where the bottom of the fork is, we need to determine where it crosses the front wheel so that we can specify the top of the fork as well. Note that we assume here that the fork will cross the front wheel (and thus that the fork_offset (e) isn't so massive that it is larger than the radius of the front wheel). Since we don't know the headtube angle up front, we need to compute both locations that the fork line would cross the front wheel (assuming it had infinite length), and then choose the one that is at the top of the wheel. To do this, we will use a modified version of the equation of a circle:

$$(x - h)^2 + (z + k)^2 = r^2$$

assuming that $h = A$ and $k = R_f$ we get:

$$(x - A)^2 + (z + R_f)^2 = r^2$$

now, since we know that the fork circle is centered at the temporary origin, we can use the equation of a line and plug that in for z to get:

$$(x - A)^2 + (m * x + (b - R_f))^2 = r^2$$

substituting $fork\_slope$ for m and $fork\_intercept$ for b results in:

$$(x - A)^2 + (fork\_slope * x + (fork\_intercept - R_f))^2 = r^2$$

now, multiplying everything out and combining like terms results in:

$$(fork\_slope^2 + 1)x^2 + (2 * (fork\_intercept - R_f) * fork\_slope - 2A)x + (A^2 + (fork\_intercept - R_f)^2 - r^2) = 0$$

now, using the quadratic equation we can compute the two locations (x_1, z_1) and (x_2, z_2) that the fork would cross the front wheel assuming infinite length.

$$a = fork\_slope^2 + 1$$

$$b = 2 * (fork\_intercept - R_f) * fork\_slope - 2A$$

$$c = A^2 + (fork\_intercept - R_f)^2 - r^2$$

$$x = -b \pm \frac{\sqrt{b^2 - 4 * a * c}}{2 * a}$$

Now we can compare the location of the rider's seat to know if we should extend the fork to just the wheel's radius, or past it to the height of the seat's height.

```
if Hz > 2Rf:
    fork_top_x = (Hz - fork_intercept) / fork_slope
    fork_top_z = Hz
else:
    if z1 > z2:
        fork_top_x = x1
        fork_top_z = z1
    else:
        fork_top_x = x2
        fork_top_z = z2
```

Finally, with the fork's bottom and top location's computed, we can calculate the location of its center of gravity and its total length:

$$fork\_cg_x = \frac{(fork\_top_x - fork\_bottom_x)}{2} + fork\_bottom_x$$

$$fork\_cg_z = \frac{(fork\_top_z - fork\_bottom_z)}{2} + fork\_bottom_z$$
$$fork_{len} = \sqrt{fork\_bottom_x - fork\_top_x)^2 + (fork\_top_z - fork\_bottom_z)^2}$$

H.4 Computing the Top Tube Location

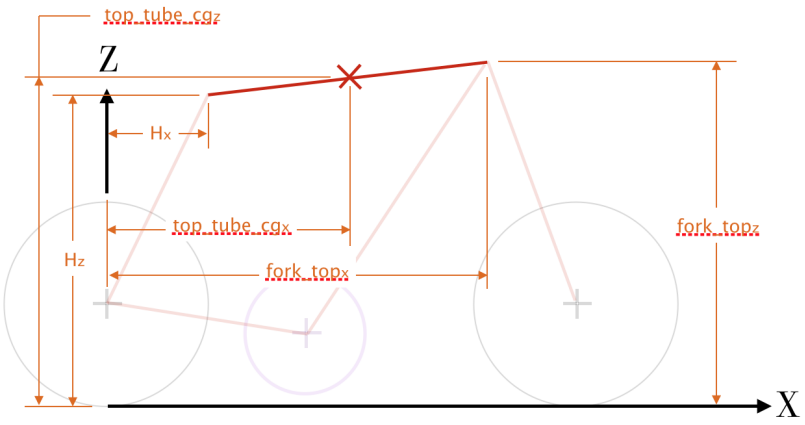


Figure H.4: Calculating the location of the frame's top tube center of gravity.

$$top\_tube\_cg_x = \frac{(fork\_top_x - H_x)}{2} + H_x$$
$$top\_tube\_cg_z = \frac{(fork\_top_z - H_z)}{2} + H_z$$
$$top\_tube_{len} = \sqrt{(fork\_top_x - H_x)^2 + (fork\_top_z - H_z)^2}$$

H.5 Computing the Down Tube Location

Since the crank location can move around relative to the top of the fork, there needs to be a set of if-else logic to determine the proper equation to use when computing the down tube.

If the crank is ahead of the front wheel then the tube's X center of gravity location is:

$$down\_tube\_cg_x = \frac{(A + C_x) - fork\_top_x}{2} + fork\_top_x$$

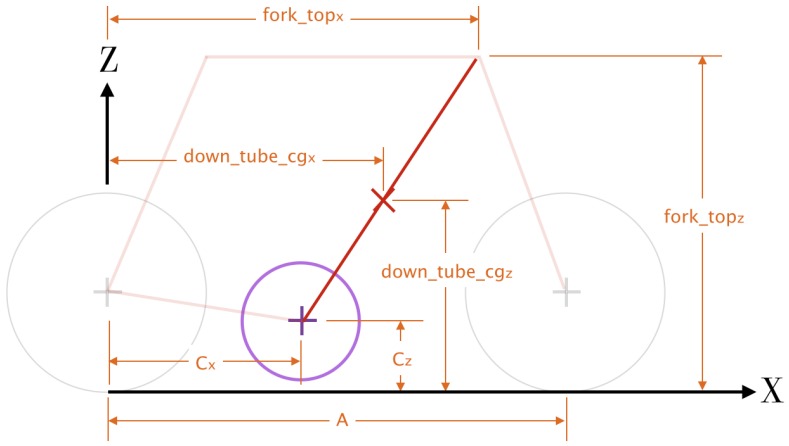


Figure H.5: Calculating the location of the frame's down tube center of gravity.

otherwise, the tube's X center of gravity location is:

$$down\_tube\_cg_x = \frac{fork\_top_x - (C_x + A)}{2} + C_x + A$$

Likewise, if the crank is above the top of the front wheel then the tube's Z center of gravity location is:

$$down\_tube\_cg_z = \frac{(C_z - fork\_top_z)}{2} + fork\_top_z$$

otherwise, the tube's Z center of gravity is:

$$down\_tube\_cg_z = \frac{(fork\_top_z - C_z)}{2} + C_z$$

with the final tube length being equation to:

$$down\_tube_{len} = \sqrt{fork\_top_z - down\_tube_z)^2 + (fork\_top_x - down\_tube_x)^2}$$

Appendix I

Bloopers

This appendix exists just to provide a visual for the range of bicycle designs that were generated in this work. The top designs represent some of the best designs that the model generated, followed by a slow degradation of handling quality as you read down the page. Note that while some designs are obviously impractical (namely the ones at the bottom of the page), there are a few which look novel enough to be worth investigating (excluding the top row which has already been built).

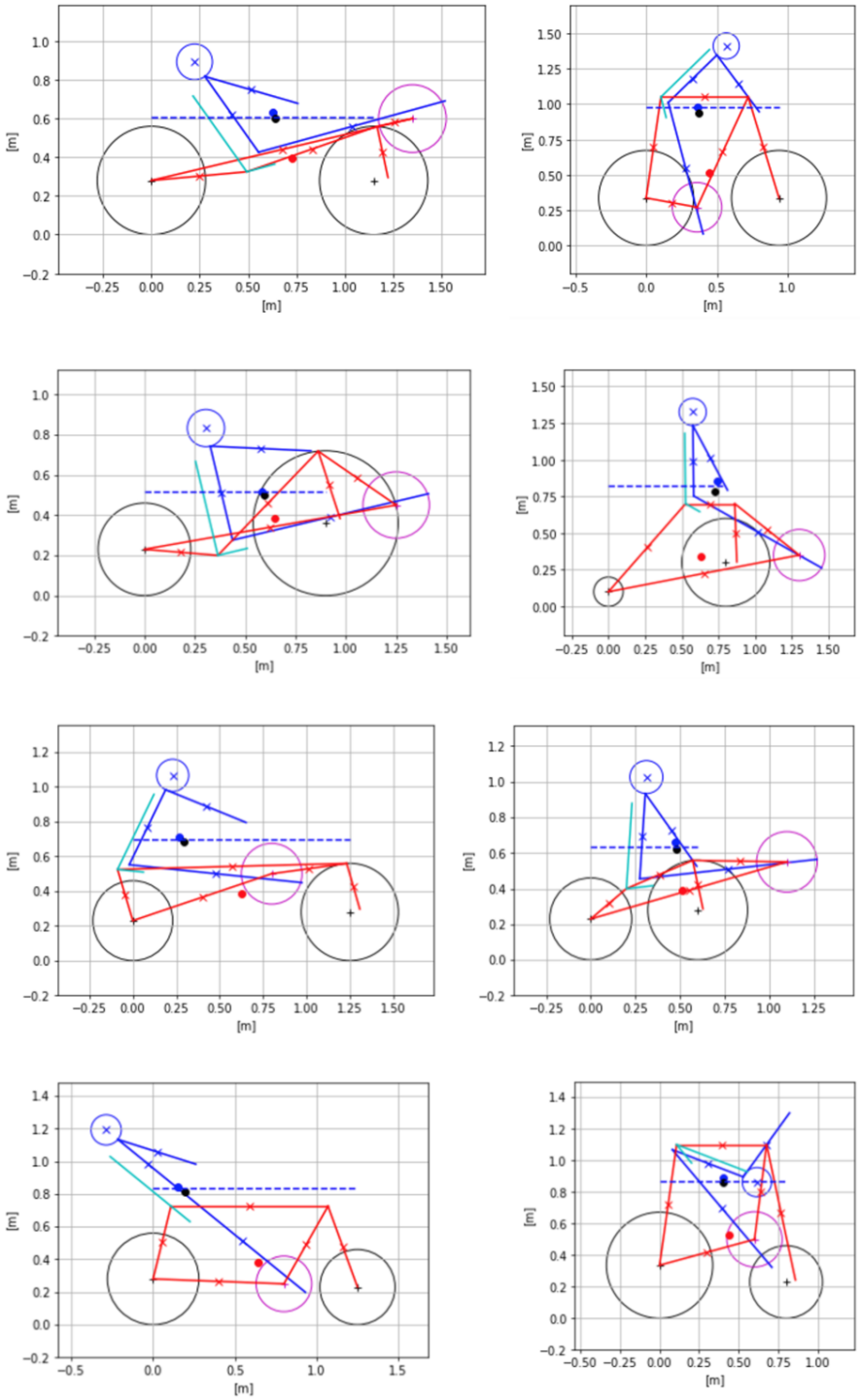


Figure I.1: A sampling of generated bicycle designs.