# Intro

There’s never been a better time to be involved in sports analytics. There is a wealth of open-sourced data and code (not to mention well-researched and public analysis) to digest and use. Both people working for teams and people just doing at as a hobby are publishing new and interesting analyses every day.

Now, I am not one for language wars—and, in fact, I use python often—but I thought it would be awesome to be able to plot Spearman’s pitch control model directly with {ggplot2} and friends. Thus, I set out to convert Laurie’s code to R, attempting to give it a “native” R feel while I was at it.

Most of the process of translating python to R was relatively straightforward (slicing and dicing data frames and arrays/vectors is just part of data cleaning), so I won’t detail them here. However, there was one part that was particularly interesting—the conversion of a python class object. This was actually the key (and most challenging part) of the conversion process.

***For demonstration purposes, I’m going to walk through my steps for converting the python class object as if I were doing it for the first time.***

# Contructor

Below is a stripped down version of Laurie’s code, showing the “essence” of what we need to replicate.2

class player(object):

def init (self,player\_id,frame): self.id = player\_id self.get\_position(frame) self.get\_velocity(frame)

def .get\_position(self,frame):

self.position = np.array(frame[self.player\_id + 'x', self.player\_id +

'y'])

def .get\_velocity(self,frame):

self.velocity = np.array(frame[self.player\_id + 'x\_v', self.player\_id +

'y\_v'])

def tti(self,final\_position): reaction\_time = 0.7 # in s vmax = 5 # in m/s

reaction\_position = self.position + self.velocity \* reaction\_time self.tti = reaction\_time + np.linalg.norm(final\_positon -

reaction\_position)/vmax

def p\_intercept(self,t):

# `tti\_sigma` is the standard deviation of the sigmoid function that quantifies the uncertainty in player arrival time

tti\_sigma = 0.45

den = 1 + np.exp(-np.pi/np.sqrt(3.0)/tti\_sigma \* (t-self.tti))) return 1 / den

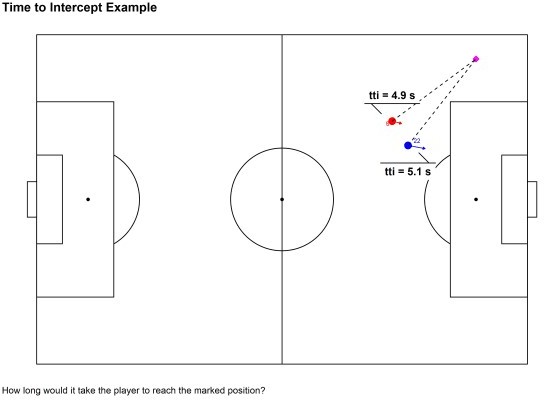
Let’s make some notes and come back to these as we develop our R class.

1. We need a unique identifier: player\_id. This is really just a “best practice” thing for object-oriented programming and makes sense given our context. For a sport like soccer, a unique identifier could just be the player’s name, a combination of the team name and the player jersey number, a league unique identifier, etc.
2. A single-row data frame frame is passed to several of the methods, including the constructor

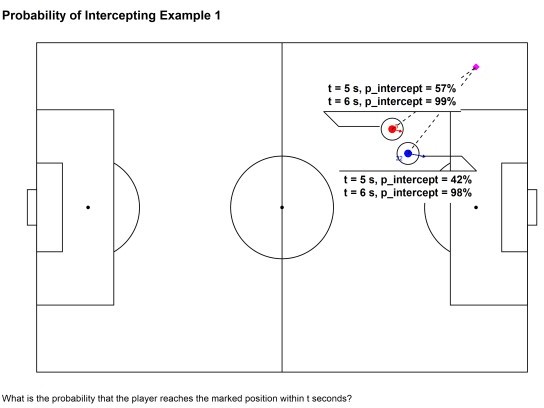
init . This single row data frame is sourced from a much larger tracking data frame, with rows for every 0.04 second time interval (25 frames per second, or one frame per 0.04 seconds) in the game.

1. The python code stores both the player’s position and velocity as 2×1 arrays. This works well with the unpacking that is done in other places in Laurie’s code.
2. tti, short for “time to intercept (a target location)”, uses the player’s position and velocity to define the attribute tti (not to be confused with the method itself). This implies that position and velocity should be defined before tti() is ever called, as they are in init . tti needs the position\_final 2×1 array to calculate tti which is not known upon instantiation; rather, tti can only be properly defined when called to do a specific calculation relating the player’s position and velocity (both defined implicitly in the class, without needing user-specification) with a user- supplied position\_final pair of x and y values.
3. p\_intercept, short for “probability to intercept (a target location)” depends on tti and an additional parameter t, a user-specified value representing how much time is allotted to reach the ball. Like tti, p\_intercept is only “properly” defined when actually doing a calculation on the player’s attributes. Unlike tti, there is no attribute in the player instance that stores this probability; it’s value must be saved in a variable external to the player class if the user wants to use it for something other than an ephemeral calculation.3

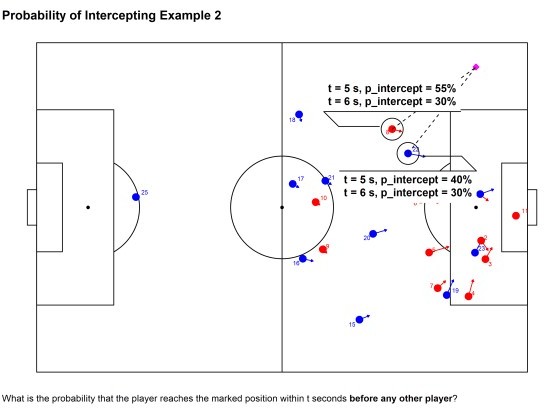
Time to intercept a “target” location (tti) may not be intuitive to comprehend immediately. The plot4 below annotates the tti of a “target” location on the pitch (which does **not** have to be where the ball actually is). tti assumes that the player continues moving at their current speed (annotated by the arrows) for reaction\_time seconds before running at vmax (full speed) to the target position. tti for each player is independent of the tti of all other players, which is a relatively reasonable assumption. 5



The probability of reaching the “target” location (p\_intercept) is directly related to the player’s tti. Uncertainty about how long it will take the player to reach the target location is quantified by the constant tti\_sigma in the calculation. (tti is the mean and tti\_sigma is the standard deviation of the distribution for a player’s time to arrive at the target location.)



Notably, this probability is independent of all other players’ probabilities (which explains how it is possible that both players are shown to have probabilities greater than 50% when t = 8 above). When adjusting for all players’ probabilities (by dividing by the sum of all probabilities), the numbers change. This probability adjustment is key when we calculate pitch control.



Ok, on to the R code. We’ll be using S3 and the {vctrs} package to help create our player class. (As with the python class, I’ve simplified the actual implementation for demonstration purposes.)

First, we start with the constructor new\_player(). Note that there is no direct init equivalent in R. Here we will make a function that is prefixed with new\_ and ends with the name of our class (player).

new\_player <-

function(player\_id = integer(), x = double(),

y = double(), x\_v = double(), y\_v = double()) {

vctrs::new\_rcrd( list(

player\_id = player\_id, x = x,

y = y, x\_v = x\_v, y\_v = y\_v,

tti = -1 # dummy value

),

class = 'player'

)

}

Now let’s reflect upon our prior notes.

1. We have the player\_id in this constructor.
2. We don’t pass the data frame tracking here. We’ll do it in our helper function. We might say that our constructor is “low-level”, not intended for the user to call directly.
3. We split the position and velocity vectors into their individual x and y components, resulting in four total variables instead of two. I don’t think a vector (unnamed or named), list, or matrix are particularly compelling data types to use for an x-y pair of values in R. None natively support unpacking (although

R vectors do have some form of “broadcasting” with their recycling behavior).

1. We assign a “dummy” value (-1) to tti when initializing the class instance. We will have a method to update tti based on x and y components.
2. Like tti, we will need a separate p\_intercept method to be used to calculate the probabililty of intercepting a ball given a player’s position, speed, and the final position of the ball (all fed as inputs to tti), as well as the additional user-specified t, representing how much time is allotted to reach the ball.

# Validator

Let’s proceed by creating a validator function to, you guessed it, validate fields in the player class. It is good practice to check the values used to construct the class. The python code did not have any validation like this, but I don’t think it was ever expected to be extremely robust to any user input.

validate\_player <- function(player) { vctrs::vec\_assert(vctrs::field(player, 'player\_id'), integer()) vctrs::vec\_assert(vctrs::field(player, 'x'), double()) vctrs::vec\_assert(vctrs::field(player, 'y'), double()) vctrs::vec\_assert(vctrs::field(player, 'tti'), double()) player

}

Note that we could have simply done this validation in the constructor function, but I think it makes sense to put the validation in its own function so that the constructor is more direct (especially if the validation checks are complex).

# Helper

Finally, we’ll create a helper player() function, which is our “user-facing” function that we expect/want users to use to instantiate objects.

player <-

function(player\_id, frame, tracking) {

player\_id <- as.integer(player\_id) frame <- as.integer(frame)

assertthat::assert\_that(is.data.frame(tracking))

nms\_req <- c('player\_id', 'frame', 'x', 'y', 'x\_v', 'y\_v') assertthat::assert\_that(all(nms\_req %in% names(tracking)))

# `!!` to make sure that we filter using the integer values, not the column itself.

tracking\_filt <- tracking %>% filter(player\_id == !!player\_id, frame ==

!!frame)

assertthat::assert\_that(nrow(tracking\_filt) == 1L)

player <- new\_player(

player\_id = player\_id,

x = tracking\_filt[['x']],

y = tracking\_filt[['y']], x\_v = tracking\_filt[['x\_v']], y\_v = tracking\_filt[['y\_v']]

)

validate\_player(player)

}

Note the following:

We coerce player\_id and frame to integers instead of doubles (particularly since they are expected to be integers in the constructor). This ensures that the new player is instantiated properly by the constructor and passes our validation.

We pass in our entire tracking data frame (that has rows for every 0.04 second interval in the game), as well as the frame to slice out of it. (player\_id is also used to filter tracking.) This makes it convenient for user to instantiate new player objects when operating on the tracking data frame. There is no need to extract the singular initial position and velocity components “manually”; instead, the helper function does it for the user.

# Aside

R’s S3 framework is not a formal OOP framework (not even close really). Note that it does not have a reserved keyword to represent the instance of the class like self in python. Also, it is not actually necessary to much of what is done above (with the constructor, validator, and helper).

For example, we don’t actually have to create a formal-ish constructor prefixed with new\_. We don’t even need a constructor function at all in S3. We could do something like class(var) <- 'player' to create a a player object. Of course, this is prone to errors down the line, so we don’t do that. Likewise with the validator and helper functions. The point of these constructs is to add clarity to our class code. They aren’t strictly necessary.

# Printing

Let’s do one more thing for our player class—create a custom print method. (Writing a custom print method is not required whatsoever, but it can be very helpful for debugging.) If we weren’t using {vctrs} and just S3, we would do this by writing a print.player function. However, {vctrs} provides a “pretty” header for us auto-magically (that looks like ) if we use it to write our print method.

To take advantage of the pretty-printing functionality offered by {vctrs}, we write a format.player() method that will be called by a subclass of the generic vctrs::obj\_print\_data method6, which itself is called whenever we print out an object (whether explicitly with print or just by typing the name of the variable representing our player instance). We’ll add the player’s position and velocity components to the print out.

format.player <- function(player, ...) { if(vctrs::field(player, 'in\_frame')) {

suffix <- sprintf(

'with `position = (%.2f, %.2f)` and `velocity = <%.1f, %.1f>`', vctrs::field(player, 'player\_id'),

vctrs::field(player, 'y'), vctrs::field(player, 'x\_v'), vctrs::field(player, 'y\_v')

)

} else {

suffix <- 'is not on the pitch'

}

prefix <- sprintf('`player\_id = %s` ', vctrs::field(player, 'player\_id')) msg <- sprintf('%s%s', prefix, suffix)

paste(msg, sep = '\n')

}

obj\_print\_data.player <- function(player) { cat(format(player), sep = '\n')

}

# Basic Usage

Ok, so that is all fine and dandy, but how would we go about instantiating players in a normal workflow? Let’s say that we want to calculate the pitch control for a single frame in the tracking data (called

tracking\_start below).7 Note that the pitch dimensions have been translated to a 100x100 unit-less grid. (A real pitch is something like 105x68 m).

tracking\_start

## # A tibble: 26 x 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ## | frame | ball\_x | ball\_y | side | player\_id | x | y | x\_v | y\_v |
| ## | <int> | <dbl> | <dbl> | <chr> | <int> | <dbl> | <dbl> | <dbl> | <dbl> |
| ## | 1 53027 | 89.25 | 36.11 | home | 1 | 86.40 | 57.90 | 5.625 | -5.861 |
| ## | 2 53027 | 89.25 | 36.11 | home | 2 | 90.57 | 39.92 | 1.429 | -2.975 |
| ## | 3 53027 | 89.25 | 36.11 | home | 3 | 91.44 | 34.29 | 1.35 | 3.521 |
| ## | 4 53027 | 89.25 | 36.11 | home | 4 | 87.99 | 23.00 | 0.957 | 5.107 |
| ## | 5 53027 | 89.25 | 36.11 | home | 5 | 79.96 | 36.31 | 4.036 | 1.764 |
| ## | 6 53027 | 89.25 | 36.11 | home | 6 | 78.28 | 52.40 | 3.707 | -0.911 |
| ## | 7 53027 | 89.25 | 36.11 | home | 7 | 81.71 | 25.49 | 1.621 | 2.239 |
| ## | 8 53027 | 89.25 | 36.11 | home | 8 | 72.44 | 73.77 | 1.921 | -0.725 |
| ## | 9 53027 | 89.25 | 36.11 | home | 9 | 58.30 | 37.28 | 0.821 | -1.132 |
| ## | 10 53027 | 89.25 | 36.11 | home | 10 | 56.85 | 51.61 | 0.857 | -0.843 |

## # ... with 16 more rows

Let’s convert players with id’s 10 through 12 (on the home team) to player instances and see how they look when printed out.

10L:12L %>% map(~player(player\_id = .x, frame = 53027L, tracking = tracking\_start))

## [[1]]

## <player[1]>

## `player\_id = 10` with `position = (10.00, 51.61)` and `velocity = <0.9,

-0.8>` ##

## [[2]]

## <player[1]>

## `player\_id = 11` with `position = (11.00, 47.48)` and `velocity = <-0.3, 0.9>`

##

## [[3]]

## <player[1]>

## `player\_id = 12` is not on the pitch

# Pseudo-Encapsulation

We still need to implement analogues for the tti and p\_intercept methods in the python player class. Starting with tti, let’s use some pseudo-encapsulation (with getters and setters) for a player’s tti value.

# Frobenious norm

euclidean\_norm <- function(x1, x2, y1, y2) { m <- matrix(c(x1, y1)) - matrix(c(x2, y2)) sqrt(sum(m^2))

}

.get\_tti.player <- function(player, x2, y2) { ri <- 0.7 # in s

vmax <- 5 # in m/s

x1 <- vctrs::field(player, 'x') + vctrs::field(player, 'x\_v') \* ri y1 <- vctrs::field(player, 'y') + vctrs::field(player, 'y\_v') \* ri ri + euclidean\_norm(x1, x2, y1, y2) / vmax

}

.msg\_cls\_err <- function(player, f) { cls <- class(player)[1]

sprintf('`%s()` doesn\'t know how to handle class `%s`!', f, cls)

}

.get\_tti.default <- function(player, ...) { stop(.msg\_cls\_err(player, '.get\_tti'), call. = FALSE)

}

.get\_tti <- function(player, ...) { UseMethod('.get\_tti')

}

`.set\_tti<-.player` <- function(player, value) { vctrs::field(player, 'tti') <- value

player

}

`.set\_tti<-.default` <- function(player, ...) { stop(.msg\_cls\_err(player, '.set\_tti'), call. = FALSE)

}

`.set\_tti<-` <- function(player, ...) { UseMethod('.set\_tti<-')

}

There’s a couple of things going on here:

The .get\_tti and .set\_tti functions that call UseMethod are true S3 generics that perform method dispatch, i.e. find the correct method for the object passed to the generic (based on the class of the object). The .get\_tti.player and .set\_tti.player with the .player “suffix” so that they only work in their defined manners when passed in a player instance. (They won’t be called with an object that is not of the player class.)

The ellipses (...) in the S3 generic function signatures may be a bit mysterious since they aren’t passed explicitly to UseMethod. Any non-player arguments are captured in these ellipses and passed to whatever method that is called from the generic (e.g. .get\_tti.player method called from the .get\_tti generic). For .get\_tti, the ellipses is intended to capture x2 and y2, and for

.set\_tti, it captures value.

We must use the “strange” syntax .set\_tti<-.player (instead of just .set\_tti.player, which may seem more “natural”) in order to update an attribute in an already instantiated class. 8

We define the function euclidean\_norm() outside of .get\_tti.player simply because it is not something that is specific to the time to intercept calculation for a player; it can work with any two pairs of x and y coordinates.9

ri and vmax, representing a player’s reaction time and a player’s maximum velocity respectively, are constants defined in the Spearman paper. We could change these if we wanted to, or even make them dynamic (i.e. configurable via other function parameters, or even at instantiation time).

To really complete our getter and setter methods for tti, we should write methods to handle the case when a non-player object is passed to them. The generic .get\_tti and .set\_tti methods will dispatch to these functions if the object passed to them (the first argument named player) doesn’t actually inherit from the player class.

.get\_tti.default <- function(player, ...) { stop(.msg\_cls\_err(player, '.get\_tti'), call. = FALSE)

}

.set\_tti.default <- function(player, ...) { stop(.msg\_cls\_err(player, '.get\_tti'), call. = FALSE)

}

Let’s see how our pseudo-encapsulation looks in action.

players <- 8L:10L %>% map(~player(player\_id = .x, frame = 53027L, tracking = tracking\_start))

map(players, ~vctrs::field(.x, 'tti')

|  |  |
| --- | --- |
| ## | [[1]] |
| ## | [1] -1 |
| ## |  |
| ## | [[2]] |
| ## | [1] -1 |
| ## |  |
| ## | [[3]] |
| ## | [1] -1 |

target\_x <- 89

target\_y <- 36

for(i in seq\_along(players)) {

value <- .get\_tti(players[[i]], x2 = target\_x, y2 = target\_y)

.set\_tti(players[[i]]) <- value

}

map(players, ~vctrs::field(.x, 'tti')

|  |  |
| --- | --- |
| ## | [[1]] |
| ## | [1] 8.749699 |
| ## |  |
| ## | [[2]] |
| ## | [1] 6.72504 |
| ## |  |
| ## | [[3]] |
| ## | [1] 7.689027 |

Note how the player tti values changed after we defined them for a specified target\_x and target\_y.

Our approach to p\_intercept is very similar to that for tti, so I don’t show most of it here. As before, we define getters and setters, as well as generics for the class (the intended target of method dispatch), as well as a default class to handle unexpected inputs. Probably the only interesting part is the calculation itself, as shown below. If you compare it to the p\_intercept method in the python object definition, you’ll see it’s basically identical.

.get\_p\_intercept.player <- function(player, t) { tti\_sigma <- 0.45

den <- 1 + exp((-base::pi / sqrt(3) / tti\_sigma) \* (t - vctrs::field(player, 'tti')))

1 / den

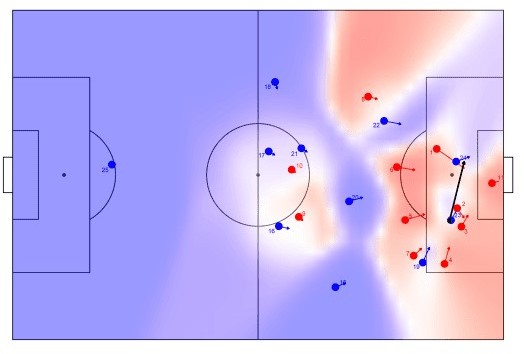
}

There is certainly more to show, especially for what is needed to calculate pitch control. (We need to integrate probabilities across all players over time, and do it for the entire pitch.) Nonetheless, the player class and the pseudo-encapsulation that we’ve implemented with S3 and {vctrs} is really the key component underlying the whole pitch control calculation.

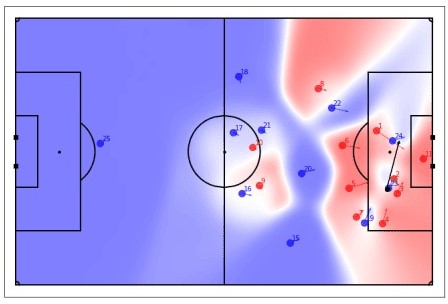
# Advanced Usage

To really motivate the reader, let’s see what this implementation allows us to do.

First, let’s emulate the pitch control plot of event 823, which is a pass by the away (blue) team in the home (red) team’s penalty area preceding a successful shot.

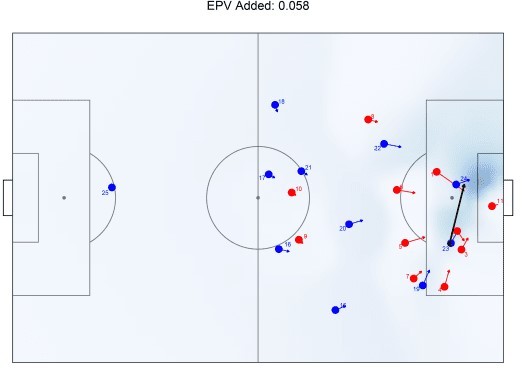


Compare this to the python version.

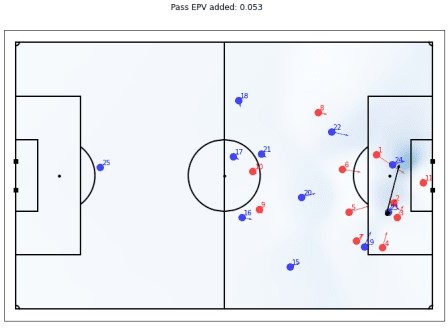


By no means is it a perfect replication—some of the contour edges are at different angles—but I think it’s a pretty good emulation overall.

Second, let’s replicate the expected possession value (EPV) plot of the same event, including the EPV added by the pass.



Again, we can compare this plot to the python equivalent.



Cool, it seems to be pretty close. We do have a small discrepancy in the EPV added calculation. (This EPV is actually an “expected” EPV calculation that uses pitch control to weight the pre-learned EPV grid). I believe this is probably due to discrepancies in the integration done in the pitch control calculation and not due to a significant a code issue.

The code to prepare the data for these plots gets more complex, which is why I have excluded it here.10 However, none of it is unreasonably difficult to understand or implement once we have a properly defined player object.

# Conclusion

Thus, we see that there is a huge payoff to creating a sound and robust player object—we can calculate

pitch control and EPV, and feed them into pretty visualizations that can provide insight. I believe that the code here could be easily adapted to fit whatever one would like to study. For example, the valuation grid used here could be easily swapped out with expected threat (xT), which is a more modern and probably a better valuation framework than the one used here. Furthermore, one could calculate EPV across the entire game. The possibilities for analyses really open up.