**What is Efficiency Factor (EF)?**

Essentially, EF is the average distance that you are propelled forward per heart beat. The higher the number, the more efficient you are at running.

To calculate EF, you need your speed and heart rate data. In Joe's post, he gives the following example (using imperial measurements):

For a run with a Normalised Graded Pace of 7.5 min/mile (8 miles per hour), the speed in yards per minute is 234.7 (1760 yards * 8/60). If the average heart rate is 150, the EF is 1.56 (234.7/150).

I haven't used Training Peaks, but it seems EF is given as a single measure for your entire run. It should be possible to calculate EF as a rolling measure so that you could look at any changes in EF over time, or over specific terrain on the run. So this was my other motivation for calculating EF in R.

**Getting the data into R**

I'm using the trackeR package to do the hard work of loading the data from the gpx file.

```
library(trackeR)
library(ggplot2)
library(zoo)
library(hms)
library(gridExtra)

# select a gpx file
filepath <- file.choose()
runDF <- readGPX(file = filepath, timezone = "GMT")
```

This gives us the data from the gpx file in a data frame called runDF. Now we have to calculate the normalised graded speed. This is done by first calculating the point-to-point distances, altitudes and thereby the point-to-point gradients. Point-to-point here refers to the sampling frequency of the gps device. We can extract that information too, so that we can calculate speeds.

Here is the code that gets us to normalised graded speed. The calculation of the dist_adj column is described below.

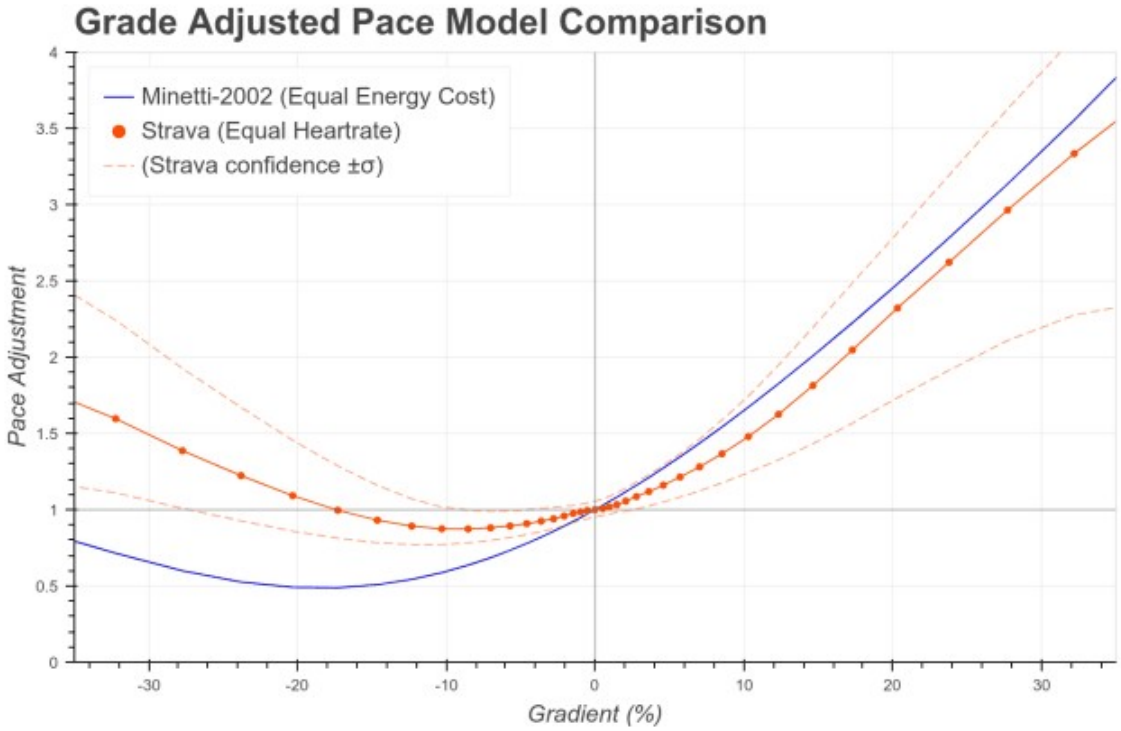
```
# calculate point-to-point distance from the cumulative distance
runDF$dist_point <- c(0,diff(runDF$distance, lag=1))
# calculate the point-to-point gradient as %
runDF$alti_point <- c(0,diff(runDF$altitude, lag=1))
runDF$grad_point <- (runDF$alti_point / runDF$dist_point) * 100
runDF$dist_adj <- runDF$dist_point *
  (0.98462 + (0.030266 * runDF$grad_point) +
     (0.0018814 * runDF$grad_point ^ 2) +
     (-3.3882e-06 * runDF$grad_point ^ 3) +
     (-4.5704e-07 * runDF$grad_point ^ 4))
# time calculations
runDF$time_temp <- strptime(runDF$time, format = "%Y-%m-%d %H:%M:%S")
runDF$time_point <- c(0,diff(as.vector(runDF$time_temp), lag=1))
runDF$time_temp <- NULL
runDF$time_cumulative <- cumsum(runDF$time_point)
runDF$time_hms <- as_hms(runDF$time_cumulative)
# speed in m/s
runDF$speed <- runDF$dist_point / runDF$time_point
# normalised graded speed in m/s
runDF$ngs <- runDF$dist_adj / runDF$time_point
# replace NaNs with 0
runDF[is.na(runDF)] <- 0
```

```
# Efficiency Factor is ngs in yards per minute divided by heart rate
runDF$EF <- (1.0936133 * runDF$ngs * 60) / runDF$heart_rate
```

### How do I calculate normalised graded pace?

If you are running uphill, your pace will be slower when running on the flat (and vice versa). Can we normalise for gradient to see how fast the runner would be travelling if the whole course was flat.

There is a paper from 2002 looking at the changes in pace over different gradients (Minetti et al. (2002)). However, this was a small study of 10 people. Nowadays we have much more data thanks to consumer-grade GPS-enabled devices. Strava used their user's data to make a new model to calculate GAP (Grade Adjusted Pace) for display on their site. Their findings plotted with those of Minetti et al. are shown here:

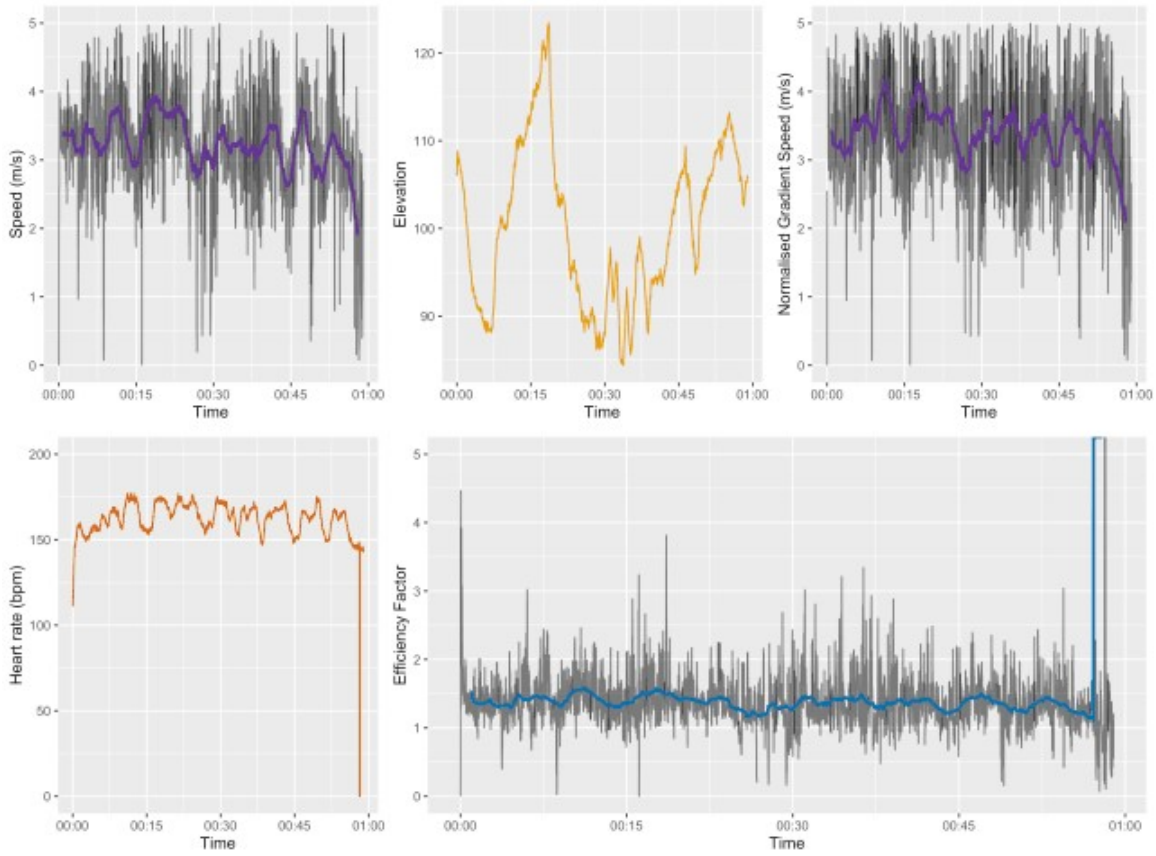



Reproduced from medium post by Drew Robb

I took the points from this graph using IgorThief and fitted a polynomial equation to this data. The co-efficients of this fit allowed me to find what kind of speed adjustment is required for a given gradient. This information is used the line in the code above:

```
runDF$dist_adj <- runDF$dist_point *
  (0.98462 + (0.030266 * runDF$grad_point) +
     (0.0018814 * runDF$grad_point ^ 2) +
     (-3.3882e-06 * runDF$grad_point ^ 3) +
     (-4.5704e-07 * runDF$grad_point ^ 4))
```

Let's look at our graphical output before seeing how it was generated.




Example of the output (EF is shown in blue, bottom right)

The output shows speed, elevation, normalised graded speed, heart rate and EF. The example shown above has some intervals (note the peaks in the heart rate trace). EF is pretty constant, but varies a bit during the interval/rest cycles.

**The remaining code**

The plots above were generated using this code.

```
# format ticks on plots to be hh:mm
format_hm <- function(sec) stringr::str_sub(format(sec), end = -4L)

# make the plots
p1 <- ggplot(runDF, aes(time_hms, speed)) +
  geom_line(aes(alpha = 0.2)) +
  ylim(0,5) +
  geom_line(aes(y = rollmean(speed, 120, na.pad = TRUE)), colour = "#663399", size =
1) +
  scale_x_time(labels = format_hm) +
  labs(x = "Time", y = "Speed (m/s)") +
  theme(legend.position="none")
p1
p2 <- ggplot(runDF, aes(time_hms, altitude)) +
  geom_line(colour = "#E69F00") +
  scale_x_time(labels = format_hm) +
  labs(x = "Time", y = "Elevation") +
  theme(legend.position="none")
p2
p3 <- ggplot(runDF, aes(time_hms, ngs)) +
  geom_line(aes(alpha = 0.2)) +
  ylim(0,5) +
  geom_line(aes(y = rollmean(ngs, 120, na.pad = TRUE)), colour = "#663399", size = 1)
+
```

```
  scale_x_time(labels = format_hm) +
  labs(x = "Time", y = "Normalised Gradient Speed (m/s)") +
  theme(legend.position="none")
p3
p4 <- ggplot(runDF, aes(time_hms,heart_rate)) +
  geom_line(colour = "#D55E00") +
  ylim(0,200) +
  scale_x_time(labels = format_hm) +
  labs(x = "Time", y = "Heart rate (bpm)") +
  theme(legend.position="none")
p4
p5 <- ggplot(runDF, aes(time_hms,EF)) +
  geom_line(aes(alpha = 0.2)) +
  ylim(0,5) +
  geom_line(aes(y = rollmean(EF, 120, na.pad = TRUE)), colour = "#0072B2", size = 1)
+
  scale_x_time(labels = format_hm) +
  labs(x = "Time", y = "Efficiency Factor") +
  theme(legend.position="none")
p5

# arrange plots into a quilt and save
q1 <- grid.arrange(p1,p2,p3,p4,p5, layout_matrix = rbind(c(1,2,3),c(4,5,5)))
saveName <- paste("./Output/Plots/",sub(".gpx",".png",basename(filepath)),sep = "")
ggsave(saveName, q1)
```

Most parts of that code are standard ggplot stuff. Two exceptions:

- To make the rolling average, I used the zoo package and calculated a 2 minute (120 s) window with rollmean
- Suppressing the seconds on the x-axis was done using format_hm. This is specified in the code and changes an hh:mm:ss time to hh:mm