Application that uses fuzzy transform for structural break detection in time series data

Elif Ozdemir & Szymon Majorek

30/04/2020

Intoduction

Time series play an important role in data processing area as it is one of the most commonly used structures. Knowing how to deal with this type of series is quite vital in the field of data mining and economics. Besides, F-transform is a very substantial technique for detecting diverse types of breaks. In this report we are going to show the power of those two combined, meaning, we will use F-Transform techniques for detecting strcutural breaks in time series.

Algorithm Description

We followed the implmentation of the algorithm for Detection of Structural Breaks in Time Series Using Fuzzy Techniques by Vilem Novak.

1. Partitioning

We create the fuzzy partitions depending on the value of h and the length of the given time period. h is the distance between the nodes. In our algorithm we decided to keep the value of h between 2 and 10 inclusive. Support of the basic functions equal to 2h.

$$\mathcal{A} = \{A_0, \ldots, A_n\}, \qquad n \geq 2,$$

Figure 1: Fuzzy sets

General formula for the generating function. In the next section we provide more specific formulas.

$$A_k(x) = A\left(\frac{x - x_k}{h}\right), \qquad x \in [c_{k-1}, c_{k+1}]$$

Figure 2: Formula for $A_k(x)$

2. Setting right and left context boundary

For evaluation of the trend we set the context.

$$v_L = 0, v_R = (\pm\sigma)/(2h)$$

Figure 3: Left and right context boundary formula

3. F^1 -transform

After we define the fuzzy partitions we calculate the finite vector $F[f]$ in few steps.

We start by the simplest form of the F-transform which is zero degree F-transform. Later we use the following formula for the calculations of β_k^0 value.

$$F_k[f] = \frac{\int_a^b f(x)A_k(x)\,dx}{\int_a^b A_k(x)\,dx}, \qquad k = 0, \ldots, n.$$

Figure 4: Zero F-transform

For higher degree transformations it is necessary to increase the degree of the polynomial. Applying this rule we are able to calculate β_k^1 for each k.

$$\beta_k^0 = \frac{\int_{c_{k-1}}^{c_{k+1}} f(x)A_k(x)dx}{\int_{c_{k-1}}^{c_{k+1}} A_k(x)dx},$$
$$\beta_k^1 = \frac{\int_{x_{k-1}}^{x_{k+1}} f(x)(x-c_k)A_k(x)dx}{\int_{c_{k-1}}^{c_{k+1}} (x-c_k)^2 A_k(x)dx}.$$

Figure 5: Assumption: boundaires for β_k^1 in the numerator contain c intead of x

These vlaues are coefficients in the equation for finding first degree transform of F.

We do not calculate higher degree transformations as the first degree gives satisfactory results in the later steps. So at the end of the transformation we get the following vector that consists of real numbers.

$$F_k^1[f](x) = \beta_k^0 + \beta_k^1(x - c_k)$$

Figure 6: Formula for the first degree F-transform

$$\mathbf{F}^1[f] = (F_1^1[X], \ldots, F_{n-1}^1[X])$$

Figure 7: F1-transforms computed over the fuzzy partitions

4. Filter β_k^1

We specify the β_k^1 that satisfy the condition for structural break.

5. Locate β_k^1 in the time series

We specify the corresponding fuzzy sets A_k for the β_k^1 that point out on the potential breaks.

Source Code with Explanation

The application is built using Shiny package. The Shiny package for R allows us to build a simple HTML based app without putting much thought into actual web development, hence the code is split in 2 main parts regarding the app itself.

1. ***ui.R***
Code for UI is the much simpler part, here we simply define what will be visible in the application using functions (controlos) from the shiny package, for exmaple

```
names<-colnames(read.csv2("data_all_4.csv"))
selectInput("countryInput","Select country", choices = names[names!="year"])
```

Creates input control responsible for picking country for which we want to display the data. We also define output controls, such as:

```
plotOutput("Plot")
textOutput("Years")
```

so that we can show our calculated results. The actual calculations take place in the 2nd part of the app.

2. ***server.R***
This is the part of the code where actual calculations take place and plots are created, we can actually split this .R into part that calculates necessary data and then part that uses this data in order to plot obtained results. The *shinyServer()* function is responsible for actual server, in this function we render the plot using other functions written by us.
Initially we have to create our partitions, we've decided to make them uniform so as to ease up our work and we create them using *uniformPartitioning* function, whose main part is:

```
s1 <- seq(0, len-2*h, by = h)
s2<-seq(h, len-h, by= h)
s3<-seq(2*h, len, by = h)
A<-Map(c, s1,s2,s3)
```

Where len is the length of vector of x values and h is fuzzy partition horizon picked by user.
The three functions responsible for calculating function approximation, β^0 and β^1 are respecitvely named *get_F*, *get_beta0* and *get_beta1*. Each of them works with accordance to aforementioned algorithms (report section 2).
Hence in order to get β^0, we calculate weighted averages of the functional values (provided as the data) where weights are the membership degrees. Before diving into *get_beta0* itself, let's take a look at how we calculate the weights. We use this formula;

$$\begin{cases} \dfrac{x - c_{i-1}}{c_i - c_{i-1}} & x \in [c_{i-1}, c_i] \\ \dfrac{c_{i+1} - x}{c_{i+1} - c_i} & x \in [c_i, c_{i+1}] \\ 0 & otherwise \end{cases}$$

Whic translated to R code looks like this:

```
func <- function(Ai, x) {
  ci<-Ai[1]
  ciii<-Ai[length(Ai)]
  cii<-median(Ai)
  ifelse(x>=ci & x<cii, (x-ci)/(cii-ci), ifelse(x>=cii & x<=ciii, (ciii-x)/(ciii-cii), 0))
}
```

Now to calculate β^0 we use discrete version of weighted average:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i \cdot w_i}{\sum_{i=1}^{n} w_i}$$

Where ws are weights. β^0 is calculated by going through the partitions A and for each of them calculating weighted average by first calculating the numerator part of β_i^0, then the denominator and then dividing, it is achieved in a double for loop.

```
for (i in 1:length(A)){
  Ai <- A[[i]] # ith partition
  avg <- 0
  sum1 <- 0
  x2<-seq(Ai[1], Ai[length(Ai)], 1)
  for (j in x2[1]:x2[length(x2)]){
    if(j>length(y)) { # ifs to ensure no exceptions
      break;
    }
    if (j==0){
      next;
    }
    avg <- avg + y[j]*func(Ai, j) # numerator
  }
  sum1 <- sum(func(Ai,x2)) # denominator
  avg <- avg / sum1 # final result for given partition
  B0[i] <- avg
}
```

β^1 works in similar way, however it's numerator as well as denominator are slightly different, we calculate it

as:

$$\beta_k^1 = \frac{\sum_{i=1}^{n} y_i \cdot (i - c_k) \cdot A_{k_i}}{\sum_{i=1}^{n} (i - c_k)^2 \cdot A_{k_i}}$$

and translated to R code it is again a double for loop:

```
for (i in 1:length(A)){
  Ai <- A[[i]] # ith partition
  avg <- 0
  sum1<-0
  x2<-seq(Ai[1], Ai[length(Ai)], 0.000001)
  for (j in x2[1]:x2[length(x2)]){
    if(j>length(y)) { # ifs to ensure no exceptions
      break;
    }
    if (j==0){
      next;
    }
    avg <- avg + y[j]*func(Ai, j)*(j-Ai[2]) # nominator
    sum1 <- sum1 + func(Ai,j)*(j-Ai[2])^2 # denominator
  }
  avg <- avg / sum1 # final result for given partition
  B1[i] <- avg
}
```

Having calculated both β^0 and β^1 we can get our approximation F. To do that we use formula discussed in **algorithms** section.

```
for (i in 1:length(A)){
  for (j in seq(A[[i]][1], A[[i]][3], 0.001)){
    if(j>length(y)) { # ensuring no exception
      break;
    }
    fi[j]<-B0[i]+B1[i]*(j-A[[i]][2])
  }
}
F<-fi
```

Once we have calculated all of those subparts, we use them to
1. Obtain structural breaks.
The process of doing that is rather simple, we use threshold based on standard deviation of input data (discussed in **algorithms** section), namely:

```
right_context <- abs(sd(y)/2/h)
satisfactoryB1 <- B1>right_context |  B1<(-right_context)
```

executing this code returns a logical array of values, where TRUE values indicate structural breaks at given positions. We then use all of this combined in plotting.
2. Plot our results.

```
plotPartitions(A, satisfactoryB1, y, x, maxVal, minVal)
lines(F*100, col="green")
lines(y*100, col="red")
```

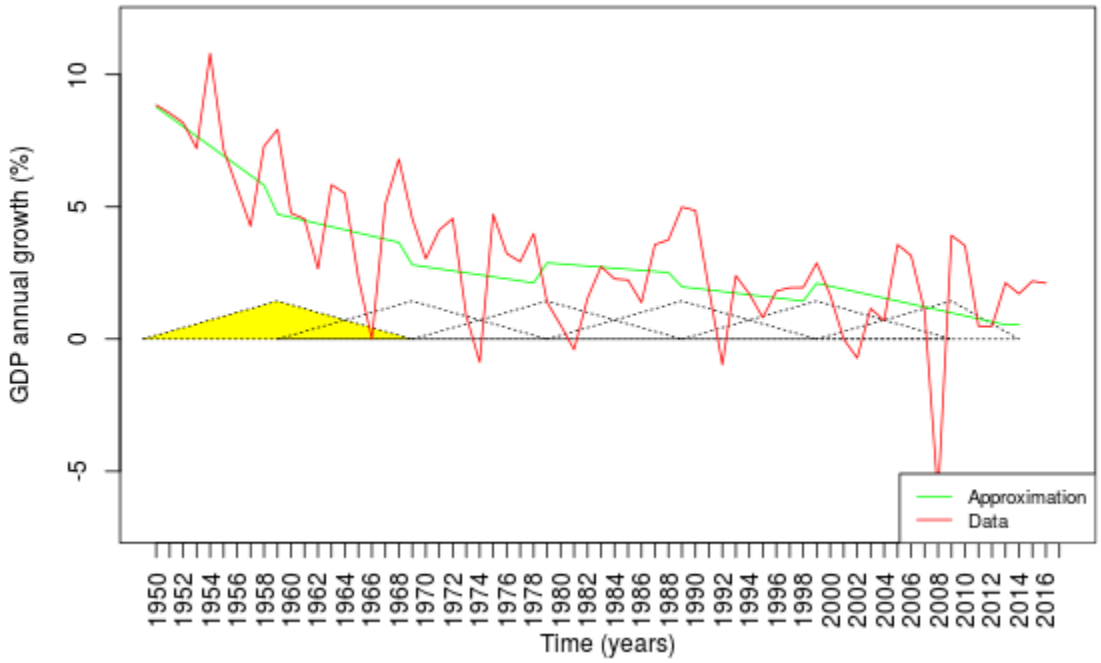
takes care of plotting, where F is approximation, y is actual data and *plotPartitions()* plots the partitions we

obtained earlier, colouring ones with structural breaks.
We first create an empty plot with preset limitations based on x amount of data points to plot, minimal and maximal values to be plotted as well as the the partitions. *polygon()* function creates polygons indicating partitions, which are then scaled based on maximal value so that they don't appear completely ridiculous in some extreme cases.

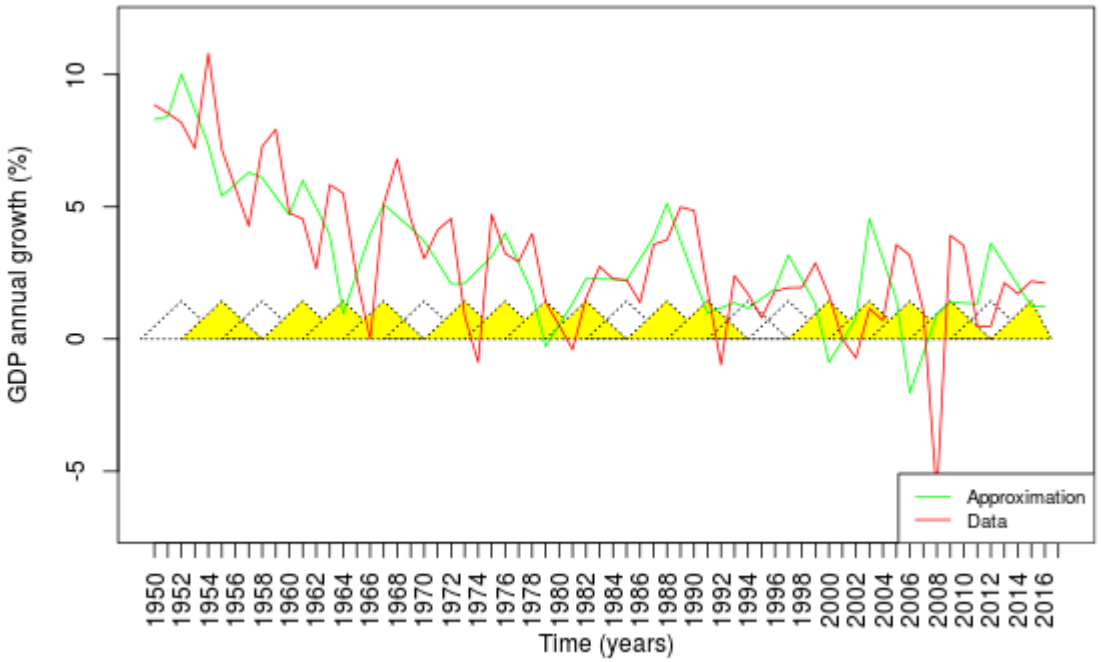
```
plot(NULL, xlim=c(1,length(x_years)), ylim=c(minVal*100-1, maxVal*100+1),
  ylab="GDP annual growth (%)", xlab="Time (years)", xaxt="n")
scale<-maxVal/7.5*100 # maxVal & minVal are the maximum and minimum value to be plotted
for (i in 1:len){
  Ai <- A[[i]]
  x <- seq(Ai[1], Ai[length(Ai)], 0.1)
  if(!D[i]) { # condition used to colour partition if it contains a break
    polygon(c(Ai[1], Ai[2], Ai[3]), c(func(Ai, x)[1], func(Ai, x)[func(Ai, x)==1]*scale,
      func(Ai, x)[length(func(Ai, x))]), col="yellow", lty=3)
  } else {
    polygon(c(Ai[1], Ai[2], Ai[3]), c(func(Ai, x)[1], func(Ai, x)[func(Ai, x)==1]*scale,
      func(Ai, x)[length(func(Ai, x))]), lty=3)
  }
}
# setting xticks as years instead of raw x values
axis(1, at=seq(1,length(x_years)), labels=x_years, las=2)
```

Results

The results and their quality very heavily depend on chosen data as well as the fuzzy partition horizon (h). We can again look at the results in 2 ways, we can focus on the approximation F and at the structural breaks themselves, tho they are sort of inseparable as one depends on the other.
We can take a look at few examples to better illustrate this, if we take Czech Republic into consideration, we can see that the data has a few fairly big jumps, which we suspect could be structural breaks. When using large fuzzy partition horizon, say 10, we get this graph:



Where we can immiedietly tell that even though the approximation doesn't look too exact, the detected structural break is in 1959, which is better than nothing, however it looks as though many of the breaks were omitted, hence we try a different h, say 3.



Structural breaks detected here were 1955, 1961, 1964, 1967, 1973, 1976, 1979, 1982, 1988, 1991, 2000, 2003, 2006, 2009, 2015. Not only we got more of them, they also look accurate and the approximation function looks much closer to the plotted data itself.
Another interesting example could be Aruba, when using $h = 8$, we get 1993-break.

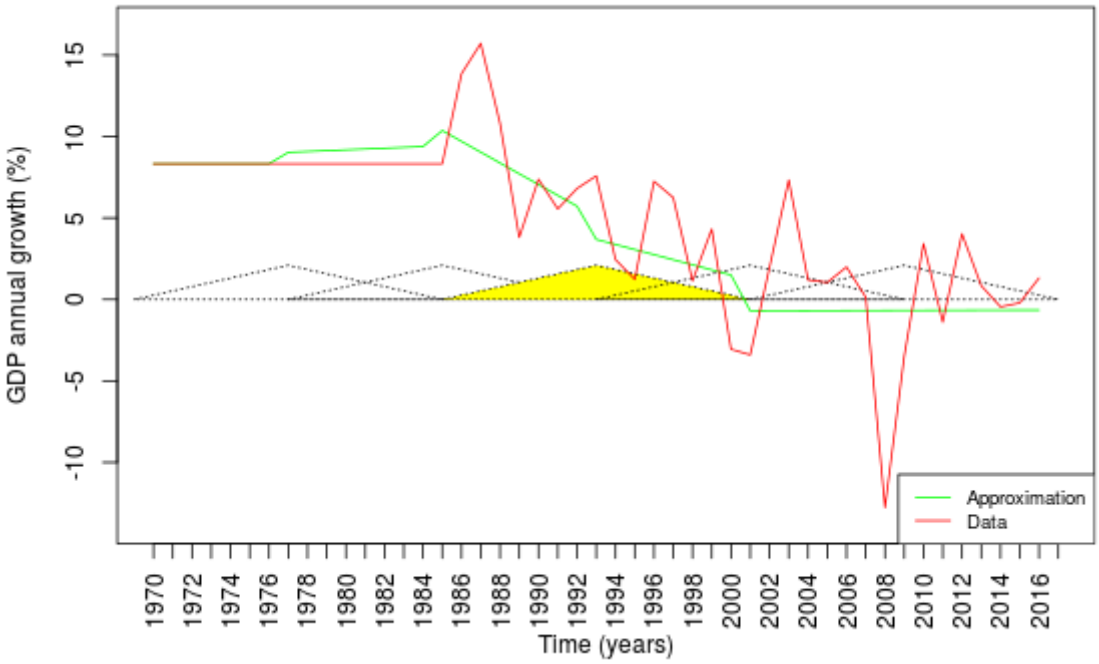
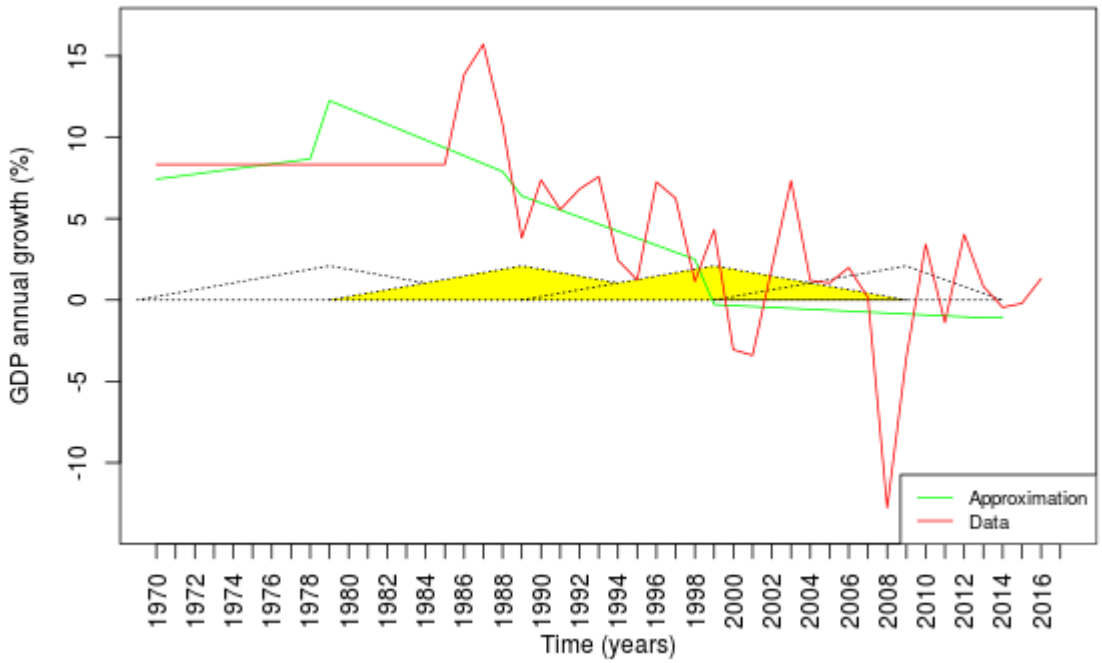
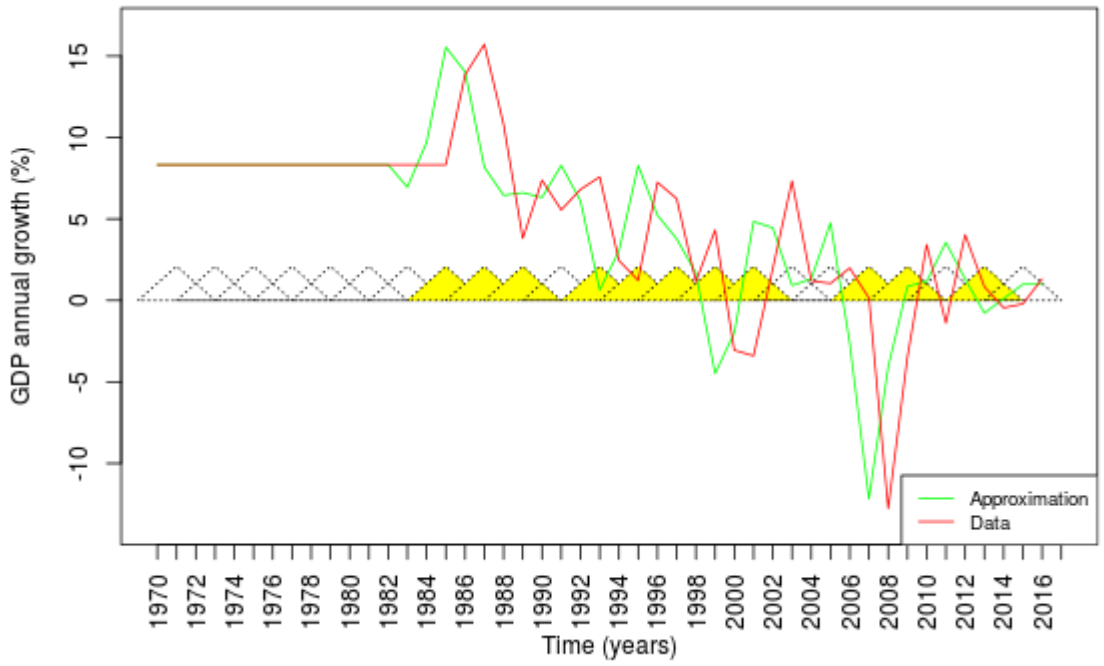


Figure 8: Aruba;h=8

However, in this case when pick a bigger h, like $h = 10$, we get more structural breaks that happened in 1989, 1999.



Nevertheless, the best results are obtained for smaller h. The next graph displays the results for $h = 2$



We get the breaks in 1985, 1987, 1989, 1993, 1995, 1997, 1999, 2001, 2007, 2009, 2013. They appear to be more accurate again, same goes for the approximation function.

Looking at those examples we can say that the results tend to be hit or miss, depending on fuzzy partition horizon choice, one of the better choices would be such that $h|length(x)$, but frankly, the best way to find accurate h is to use trail and error method.

Conclusion

We shared our implementation of the fuzzy transform technique on structural break detection in time series. As expected, the algorithm gives more precise results when the chosen h value is small. However, after testing bigger h values we see that the output is still correct in some cases. Also the efficiency of the algorithm and low computational complexity is worth mentioning.

Resources

1. http://www.ijfis.org/journal/view.html?uid=810&&vmd=Full
2. https://data.worldbank.org/indicator/NY.GDP.MKTP.KD.ZG?
3. https://stackoverflow.com/