**A white paper on CleanBrain**

**Jan Willem Koten jr.**

**Dear Reader,**

In this paper we are trying to explain how the computer code of CleanBrain works.

[**https://github.com/hinata2305/CleanBrain**](https://github.com/hinata2305/CleanBrain)**.**

In this software, we're trying to improve the test-retest reliability of fMRI. The main focus is on time course reliability. Simply put, it’s about correlating a time series measured at time A with a time series measured at time B. Of course, this only works if you give the same task to the participants at both time points. To boost the test-retest reliability, we’re attempting to build a filter that removes the noise but keeps the cognitive signal intact. Easy, right? Well, maybe not, because the optimal filters are probably dependent on the task you’re measuring in the scanner and the sampling rate of the time series. So, we’re on the hunt for the best possible filter by just trying out all sorts of filter combinations. We need a learning system to figure out which filter is the best. This happens in the script find\_filter.m. Next, we should test whether these filters actually improve the reproducibility of the time series, which is done in the script test\_filter.m. But, of course, this should happen in a different experiment to avoid circularity. In this experiment, the filter was developed on a verbal working memory task and tested on a spatial working memory task. The sections in find\_filter.m and test\_filter.m explain how the main scripts work. We’ll chat a bit about the methodology of fMRI so you can understand why and how things happen in the code.

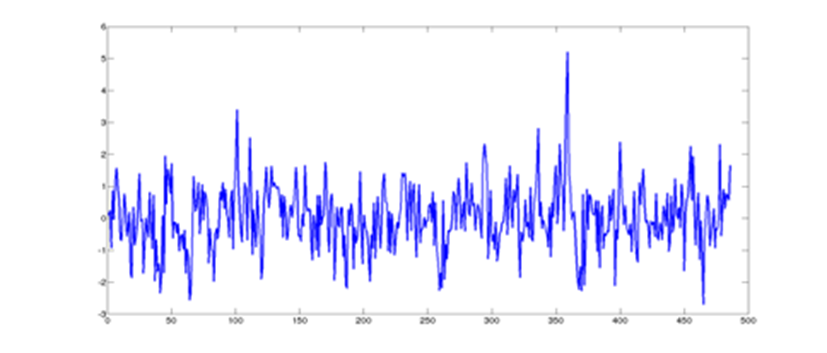
**find\_filter.m**

You’re probably familiar with those slightly cheap-looking TV magazines where the women on the cover are just too beautiful to be real. That artificial vibe comes from the fact that the photos of the made-up actors are smoothed out with special smoothing kernels, removing all the characteristics of their skin to make way for a glassy surface that nature never intended, stripping away any individuality. These Gaussian smooth kernels aren’t just used to “beautify” people; they also have a role in functional imaging, where they serve two purposes. On one hand, they reduce noise at the expense of spatial resolution. On the other hand, they try to fix the pesky spatial overlap between subjects through the kernel. While spatial smoothing kernels are still all the rage in many imaging packages today, temporal smoothing kernels were condemned by the imaging community ages ago. And honestly, the arguments against them were valid. All filters suffer from the same problem: if you overdo it, you throw the baby out with the bathwater. The real question is, what’s the baby? In imaging, it’s the signal components indicated by the cognitive tasks. If these cognitive tasks change relatively quickly, the transition from one cognitive state to the next might get smoothed out by the filter. Another issue is that filters can significantly increase the autocorrelations of the time series, leading to the creation of non-existent correlations between time series that don’t reflect the true relationship between the two.

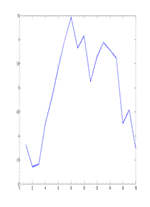
Unlike the photos of famous actors, which are generally taken with the highest quality, fMRI imaging is extremely prone to noise. The time series are somewhat like the highly noisy sounds of old audio recordings from before the war. You can definitely improve these recordings with smart filters, making what’s buried under the noise more audible.

However, with these filters, you always have to be careful not to filter out the signal of interest, like the sound of the violin. Sure, with excessive filtering, it’s possible to make all the noise disappear, but along with the noise, you’ll lose a lot of interesting sound characteristics of the violin. A conservative approach would be to try to preserve the original violin sound at all costs, even if it means you can’t get rid of all the noise. We’ve tried to follow this approach and built a kind of AI that, come what may, keeps the cognitive signal intact, accepting that we can’t eliminate all the noise.

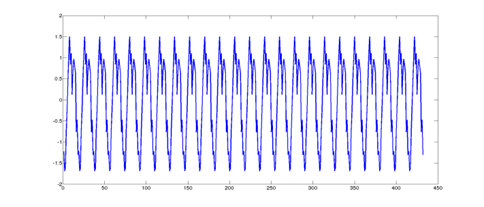
When building a filter, the question naturally arises: what exactly is the signal and what is noise? The best-known method for getting rid of noise is to average multiple observations of a phenomenon. The averaged signal represents, in a way, the best possible state of the signal, also known as the ground truth. Let’s say you have 24 repetitions of a task; you can average those 24 repetitions. The averaged signal shows how the signal associated with a cognitive task develops over time without noise. In our paper, we call this the “task-related signal average.” To understand how this signal emerges from the time series, it might be wise to take a look at the figure below:



The figure above shows a time series where a memory task was repeated 24 times. However, thanks to the high noise level in the signal, it’s pretty hard to tell where the task starts and ends. To tackle this, we mentioned averaging the parts of the signal that are related to the task. The averaged signal, of course, is much shorter and is displayed down here.



The averaged signal is noise-free and symbolizes what the cognitive-related signal fluctuations would look like if there were absolutely no noise. You could then repeat the signal shown above 24 times in a row, and you’d get a very idealized idea of what the signal for the 24 tasks would look like if there were no noise and a person performed the tasks consistently every time. This whole thing is displayed down here.

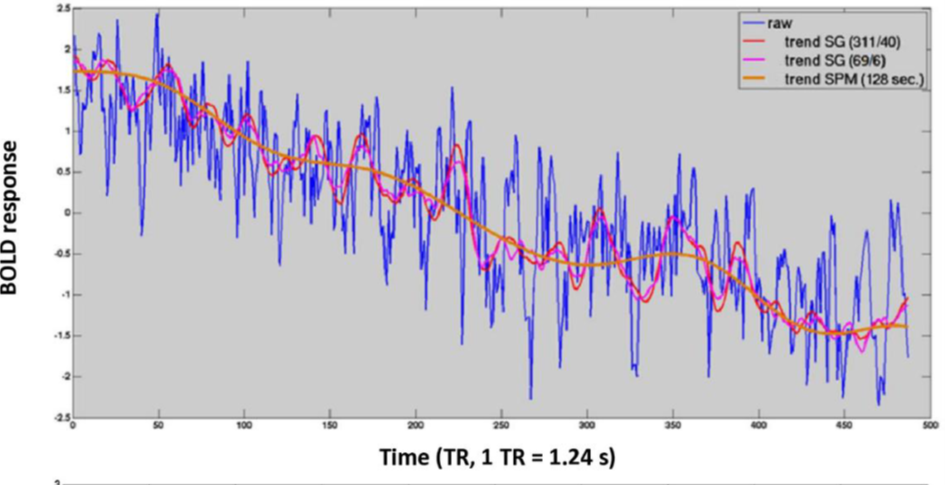


The signal shown up there is generally referred to as the empirical predictor function. It’s clear that the signal up there has a very high degree of smoothness due to the averaging. Furthermore, the signal is quite self-similar. This means that if you were to shift the signal by one unit and correlate it with the unshifted signal, the correlation would be pretty high. This relationship is called the autocorrelation at lag 1. Of course, you can shift the signal by 2 time points and correlate it, and so on. Now, we assume that a filtered signal should never have a higher autocorrelation than the correlation of the predictor function. If the signal shows higher autocorrelations than the predictor function, you’ve gone overboard and, in a way, created an actor who’s too beautiful to be real. It should be clear that the filtered signal shouldn’t be the same fMRI signal as the one from which the predictor function is derived, because that would be circular. You’d need to repeat the same experiment again. So, in short: you create a predictor function from a test run. In the next step, you repeat the experiment (retest run) and filter the signal. Then, in another step, you correlate the filtered signal with the predictor function. If the correlation between the filtered signal and the predictor function is increased by the filter, then the filter is a success, provided that the autocorrelation of the filtered signal isn’t higher than that of the predictor function.

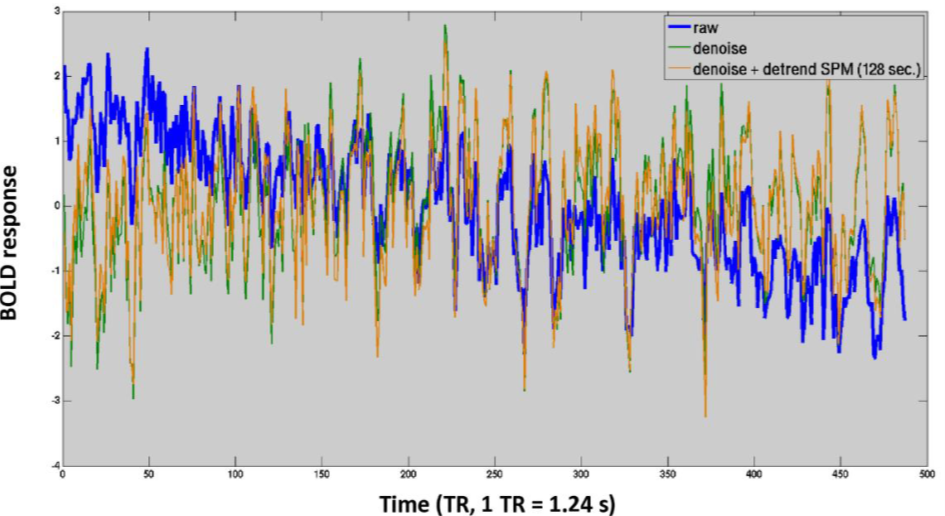
The problem, however, is that most filters have various parameters that determine the extent of filtering or noise suppression. It’s not exactly easy to know the parameter sizes that suppress noise without removing the

signal. We opted for a Savitzky-Golay filter because this filter is particularly good at preserving the original characteristics of the signal. Essentially, a window is slid over the time series, and a polynomial is fitted within that window. The value of the time point in the middle of the window is replaced by the fitted point, and then the window is shifted one point further to repeat the process. The problem is that the size of the window and the degree of the polynomial are unknown. To figure that out, it’s best to just try out all the filter combinations, which, of course, represents a huge parameter space that demands quite a bit of computational power. That’s the basic idea behind our pipeline. A child can do the laundry, as long as the washing machine is big enough.

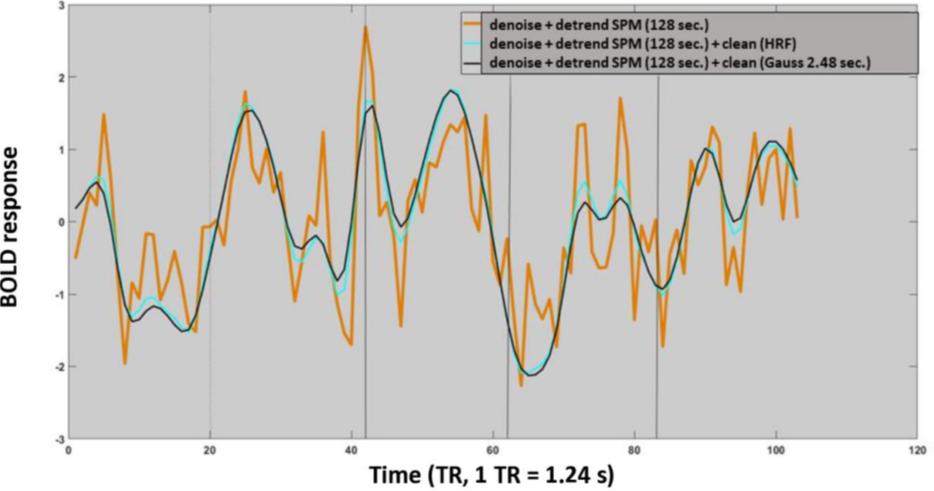
Okay, sounds simple, but there’s a little problem. There’s this difference between high-pass and low-pass filters. Hopefully the bell rings, but what was that again? Well, when you look at an fMRI signal, it can sometimes show a trend, like in the figure down here.



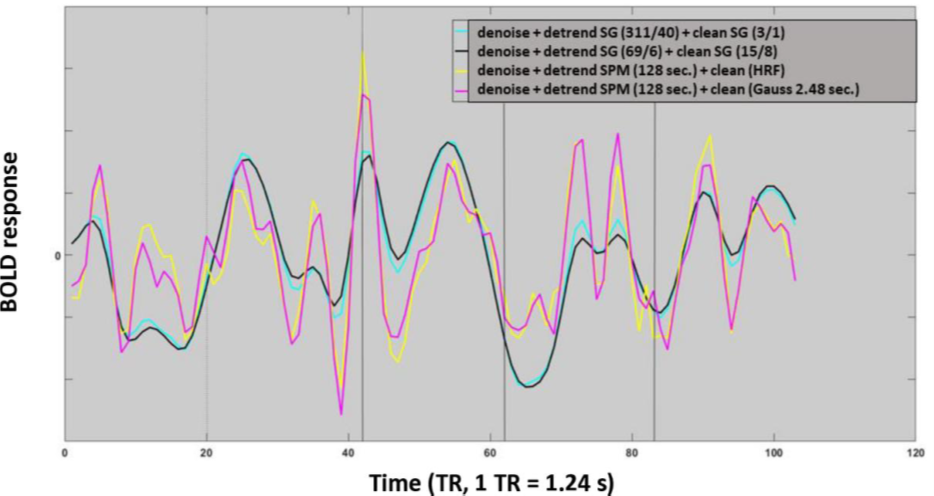
You can see that the signal shown above has a clear downward trend. However, this has nothing to do with the cognition we’re measuring. It’s just some junk that doesn’t belong there, but no one knows why. So, the whole curve needs to be straightened out to make the signal behave somewhat decently. To straighten it out, you first need to figure out exactly how the trend is behaving. You can use a filter to try to follow the trend. There are different filters you can use for this. The brown one is the usual SPM filter (DCT=128 sec), and the “red” and “pink” filters are the SG filters we’re suggesting. So, when we subtract the brown signal or the red or pink signal from the blue signal, everything gets straightened. This process is called detrending. Whether the trick actually works is shown down here.



The blue signal is the one that contains the nasty trend, while the brown signal is the same signal with the trend subtracted. You can see it’s nice and straight now. (What the green signal is will be revealed later.) So, to detrend, you need a high-pass filter. That’s a filter that lets high frequencies through but blocks the low ones. There’s also the opposite, which is the forbidden low-pass filter, but we’re trying to rehabilitate that one. As mentioned above, a low-pass filter can really backfire if you overdo it, as you can see down here.

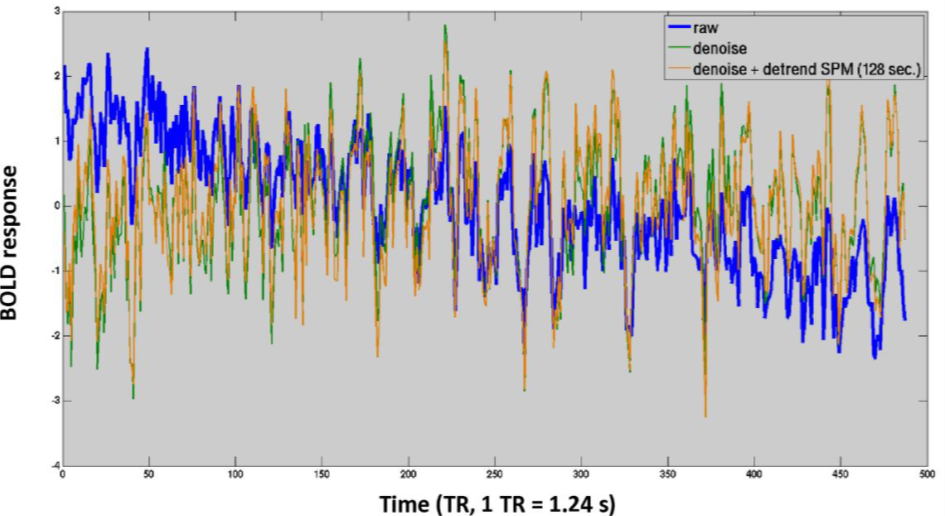


The brown signal is the original. The blue and black signals are what’s left after filtering the original with a Gaussian or HRF filter, as was commonly done in the past. A lot of the signal is gone, and that can’t be good. So, if not like this—then how? That’s the question, and we believe a Savitzky-Golay filter could do a better job.



And indeed, when you compare the SG-filtered signals here in pink and yellow with the blue and black signals, it’s immediately clear what we’re talking about. The yellow-blue signals are surprisingly similar to the original and thus contain the desirable cognitive treasure, while the Gaussian and HRF filters have actually thrown out the precious baby with the bathwater.

Now let’s turn our attention to the green signal mentioned above, where we tactically wrapped the curtain of silence around: What is that supposed to be? Well, that’s detrending.



To understand what detrending is, let’s bring back the previous image and display it above. So, denoising is another element that’s often used in the fMRI circus. However, it doesn’t really relate to the filters we’ve mentioned so far. We previously asked ourselves, in the context of the predictor function, what the true brain signal might look like. We focused on signals coming from the gray matter, which is the outer layer of the brain, also known as the cortex. But you could also ask: What is real noise? To explore that, we’ll look at tissue parts in the brain that we’re pretty sure contain no measurable brain activity. There are two categories here: the white matter, which are the "telephone cables" connecting the brain centers, and the brain holes filled with brain water. Of course, brain activity runs through these brain cables, but we can’t measure that activity with fMRI. The signals found in the white matter are dubious signals that can easily be considered real noise. The same goes for the holes in the brain filled with some kind of fluid (by the way, they’re called ventricles). The signal in these holes definitely contains no brain activity and is pure garbage. Generally, the weight of the noise time series (garbage time series) is estimated using a linear model. It goes like this:

Cognitive Signal = b1 \* White matter signal + b2 \* Ventricle signal + b3 \* Signal trend + error

The noise signals and the trend are then weighted and subtracted from the cognitive signal, so in theory, a noise-free signal remains (a nice dream). The effect is visible in the time series shown above.

We’ve essentially subtracted the junk from the blue lines up there, and what’s left is the green time series, which, by the way, looks strikingly similar to the brown time series from which both the trend and the junk were removed. So, how do we explain that? Well, it’s not too hard. The junk also contains a significant downward trend, so this trend disappears from the brain signals when you subtract the junk. The filter, by the way, is a filter from the SPM package, a discrete cosine transformation with a length of 128 seconds. But that thing doesn’t seem to do much. Does a Savitzky-Golay filter work better?

So, that’s the whole story, broadly speaking. But, as is often the case in life, the devil is in the very boring details, as the slightly tortured reader surely knows from experience. The actual goal wasn’t necessarily to provide an introduction to signal processing, but rather to explain just the bare essentials to understand my computer code.

Let’s start with the most important part: the code for creating the predictor function. This happens in the function extract\_section\_single.m. Now, the tasks in fMRI are usually not presented at regular intervals. This is done to capture the signal at different times, because the temporal resolution of fMRI is quite poor. So, you need a file that mentions the start times of the tasks, which is determined here with the variable *encode1*. The good news is that all tasks are always timed the same way, so it’s enough to add a certain time measure to the start times, referred to here as "*part*," to capture the cognitive signal. Of course, you also need the fMRI signal, because you can’t do without it.

In line 18, a loop begins where the signal sections are averaged. In line 31, the loop starts where the averaged signals are concatenated to create the actual predictor function. They are collected in the variable *toi.pred*. By the way, this function can handle multiple time series at once, which is why it’s set up to be multidimensional. We want to emphasize again that we create a predictor function for each region. In this case, we have 34 regions, so 34 functions are created per person.

As we’ve come to realize, the self-similarity of the time series, captured through autocorrelation, is another important element of the pipeline. The autocorrelation of the time series is also estimated per region. We use the function autocortimeAll.m for this. We limit ourselves to lag 1 to lag 4 autocorrelations.

As the last element, we need a function that frees the data from the noise within a linear model. This happens in a function with the not-so-surprising name remove\_noise.m. This function can handle a variable number of input arguments. In its simplest form, it only subtracts the noise time courses from the cognitive time series. For that, it only needs 2 input arguments. In the next step, both the noise and the slow trend in the signal are passed to the function. Last but not least, you can also include the high-frequency noise in the model, which means you’d need 4 input arguments. The high-frequency noise is, of course, the noise isolated by the low-pass filter. However, it’s best to keep your fingers away of that last option, as it has been shown that low-pass filters included in linear models have a disastrous effect on the cognitive components of the signal. That was one of the many attempts that didn’t work. We’ll keep the code discussion to a model with 3 input variables. In lines 31 to 33, the linear model is created. The important beta weights are then generated in line 36. In line 38, the actual cognitive time series, referred to here as y, is freed from the noise.

Last but not least, we obviously need the function detrend\_gol.m, which contains the actual SG filter. As discussed, the SG filter essentially consists of a moving window, where the average data point of the window is replaced by a point determined by a higher-order polynomial. With a moving window, there are, of course, edge issues. There are two ways to handle this: Either you accept that the SG filter can’t smooth the beginning of the signal, or alternatively, you take half the number of data points at the end of the time series that corresponds to the window size and mirror that section to slap it at the beginning of the time series before the starting edge. We opted for the latter, although the function can also do the former. The aforementioned magic happens in lines 51 to 56. In line 58, the actual SG filter is applied using the function smooth.m, and we’d like to mention that this filter is a standard MATLAB function.

**Now comes the absolute crux in two variations**. We want to optimize both a high-pass filter and a low-pass filter. However, it turns out that the two should be optimized a bit differently. The high-pass filter used for detrending should be developed together with the noise time courses within the aforementioned linear model. In contrast, the low-pass filter is optimized after the denoising and detrending have been done. In other words, the low-pass filter is not optimized within a linear model.

Let’s start with the high-pass filter. What do we need? We need:

The SG-filtered signal (trend) from a test run.

The noise time series from a test run.

The predictor function from a retest run.

We’d like to note here that we also consider the inverse case, which means determining what’s mentioned in 1 and 2 from a retest run and 3 from a test run. Furthermore, we want to determine the scope of the parameter space for the SG filter. We want to combine all window sizes with all possible polynomials. However, it obviously doesn’t make sense to use a window size that is smaller than the duration of the cognitive task, which in our case is 17 data points. Thus, the 17 data points represent the minimum window size.

All this information is combined in the function opti\_gol\_trend\_inter\_2d.m, which contains a double loop that covers the entire parameter search space. In line 24, the outer loop for the window space is initialized. Naturally, it starts with the smallest window and ends with the largest. In this context, it’s clear that a window can never be larger than the time series being examined. In each iteration, the window size is increased by one. In line 28, an inner loop is initialized, which covers the entire polynomial space, where the polynomial degree can never be greater than the window size minus 1. In line 32, the signal trend is determined using an SG filter. For this, the function detrend\_gol.m is used. In line 32, the cognitive time series is freed from the noise and the trend using the function remove\_noise.m. Then, in line 38, the actual core of the function takes place. Here, the cleaned time series is correlated with the predictor time course. It’s important to mention that the predictor function was determined outside the routine and is passed as the input variable ‘*pred*’ into the routine.

So now, let’s talk about the low-pass filter. In this case, we only need:

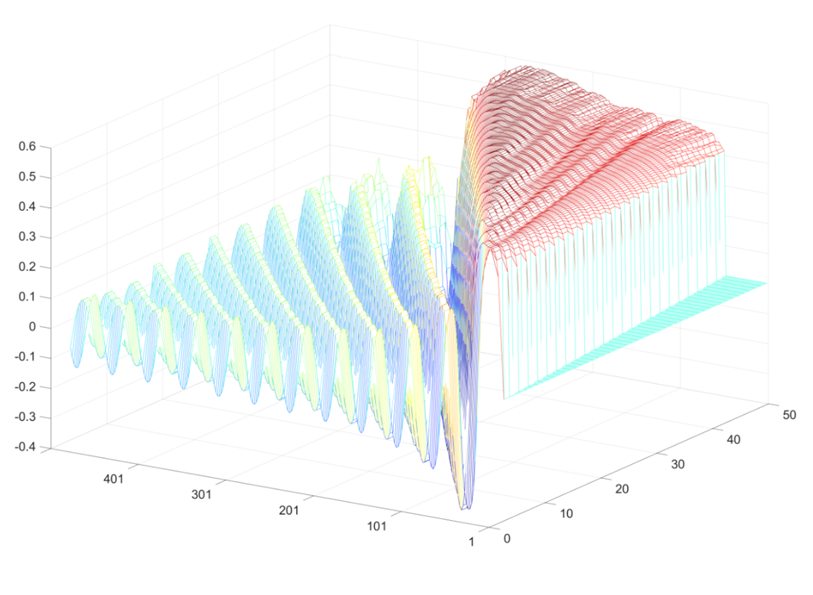
The SG-filtered signal from a test run.

The predictor function from a retest run.

As mentioned above, this needs to be determined for the entire window and polynomial space, and we use the function opti\_gol\_autonorm.m for that. As noted in the introduction, low-pass filters can increase the autocorrelations of the time series, which is why it’s important to determine the autocorrelation in this case. Let’s take a closer look at this function. As explained earlier, this function also contains a double loop to cover the entire window and polynomial space. Additionally, the autocorrelation of the filtered time series is determined in line 55. We won’t dwell too long on this function, as the attentive reader will recognize the further structure within it.

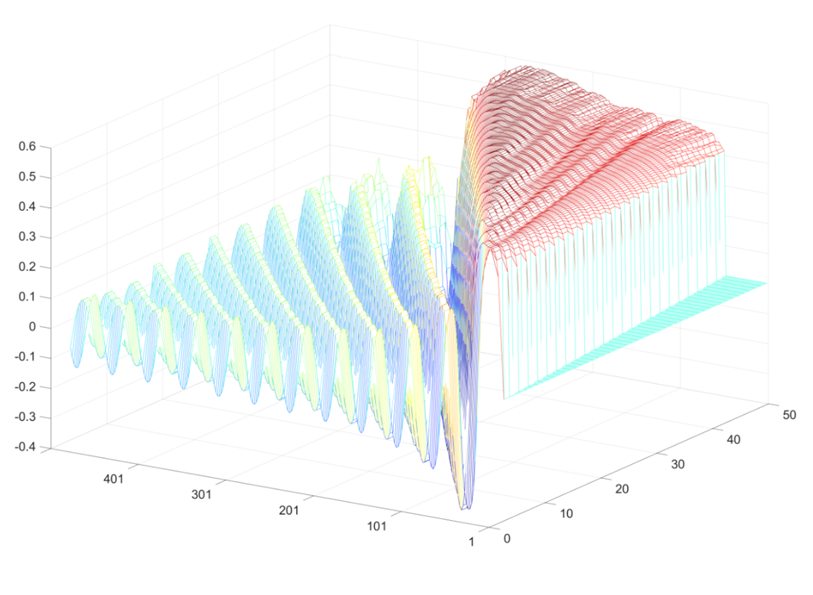
"So far, we’ve explained how to optimize the filters on a single subject basis, but of course, you need to gather the information from all individuals. To ensure this, the functions opti\_gol\_trend\_inter\_2d.m and opti\_gol\_autonorm.m are integrated into the function find\_filter.m. You can determine the version of this function in line 31. Version 1 controls the low-pass filter, while Version 2 controls the high-pass filter. Let’s start with the latter. In line 147, the loop begins that gathers the information for n individuals. As mentioned, you can obtain the cognitive time series from a test run and the predictor function from a retest run, and vice versa. In the variables *opti1* and *opti2*, the data for both variants are collected and finally stored in line 156. For the function opti\_gol\_trend\_inter\_2d.m to work properly, it obviously needs the cognitive time series, which are obtained in lines 39 and 94, as well as the noise time series, which are collected in lines 103 and 104. The latter includes the first 5 components of the white matter, the first 5 components of the ventricles, and 2 head movement components derived from the head movement time series. In all cases, a Principal Component Analysis (PCA) is applied to the corresponding time series to reduce the data set, so you’re not working with thousands of regressors, which obviously doesn’t make sense.

When you average the entire output of opti\_gol\_trend\_inter\_2d.m, you can create a graph that shows how well the filtered (detrended) and denoised signal correlates with the predictor time series. We’ve illustrated that here.

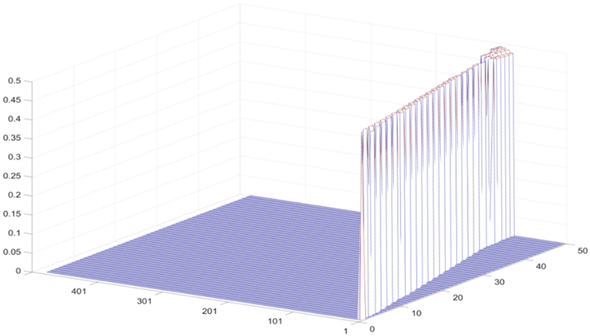


The Z-axis represents the correlation between the observed time series and the predictor time series. The left axis shows the size of the window used for the SG filter, while the right axis represents the order of the polynomial. You can see that there are quite a few filters that could potentially work, but if the filter settings are off, it can really backfire like a mouse in cat belly.

The low-pass filter is managed in a very similar way by find\_filter.m as the high-pass filter. Here, the person loop starts in line 127. However, this loop requires not only the denoised signal but also the detrended and denoised signal. This means you use the SG filter that you found in the previous step to detrend the signal. This happens in lines 67 and 77. In this case, we can also graphically represent the entire output from the routine.



The result is actually even more dramatic than in the previous case. If the filters are set incorrectly, you can even end up with negative correlations between the filtered time series and the predictor function. Meanwhile, the correct filters can significantly improve the correlation. However, we can only use filters that have autocorrelations that are not greater than the autocorrelations of the predictor function. Taking this into account, only a very narrow family of filters remains.



So that’s it for the first part of the program.

**test\_filter.m**

So, in the previous script, find\_filter.m, we were on a mission to figure out what the optimal parameters are for an SG high- and low-pass filter. As you most likely remember, we did this by correlating a filtered signal with the predictor signal, using the filter that maximizes the correlation while making sure that the autocorrelation of the filtered signal doesn’t exceed that of the predictor function. Now, in this section, we’re diving into whether our SG filter actually outperforms the SPM filters that were once part of the SPM 99 package. One should mention that the SPM low-pass filters got the boot because they were causing all sorts of problems, and of course, we’re left wondering if our SG filters are any better. To find out, we’re comparing our SG preprocessing pipeline with the good ol’ SPM 99 pipelines. We’re borrowing a few functions from the SPM 99 package for this little experiment: spm\_filter.m, spm\_Gpdf.m, spm\_hrf.m, and spm\_dctmtx.m.

There are two key criteria here: First, what’s the deal with the autocorrelations? And second, what does the time series reliability look like? After that, we’ll also check the power spectra of the different pipelines because they’ll give us a hint about whether any task-relevant frequencies have been filtered out of the signals.

The heart of this pipeline is the function estimate\_ICC\_simplified3fft.m. This function does a whole bunch of stuff, but we’re just going to focus on the important bits and skip over a ton of unnecessary details that would just confuse everyone. The key thing to remember is that you need both test and retest data to use this function. It’s crucial to keep in mind that n time series were collected per person, so this function can handle n time series at once. Lines 34 to 42 are where the Fast Fourier Transform happens. In lines 62 to 64, we’re calculating connectivity. Then, in lines 75 to 85, we’re crunching the Lag-1 to Lag-4 autocorrelations. The loop from line 90 to 132 is where we calculate the Connectivity Upper Bound and the detectable connectivity, which we’ve detailed in previous white papers. As mentioned, the function above calculates the relevant statistics for each person. To get the data for all individuals, we need to build a loop that gathers data from everyone.

This is happening in the script test\_filter.m. The actual person loop kicks off at line 57 and ends at line 229. Let’s take a closer look at this function. In lines 60 to 63, we’re loading in the data for each person. Lines 65 to 66 are where we pull in the log files for each individual. These log files tell us when a cognitive stimulus was presented to a person and how they reacted. If the scanner wasn’t perfectly in sync with the stimulus software (Presentation), we can still figure out the correct start time of the experiment here. I’ll probably ditch this function in the future since the Presentation software package that creates these log files isn’t as dominant as it used to be. In lines 68 to 73, we’re figuring out the reaction times, but honestly, that’s not the main focus. Line 75 runs a check, and any time series that are too short get tossed aside. Lines 77 and 78 determine the start and end times for the time series we’re analyzing. In line 85, we calculate various statistics for head movements. The following lines are all about executing the important processing steps and gathering the relevant statistics. In line 88, we collect the key FFT, reliability, and connectivity statistics derived from the raw time series. In lines 91 to 96, we’re trying to capture the signal trends of the gray matter using various SG filters. Finally, in lines 98 to 100, we load in the noise time series, which include 5 white matter, 5 ventricle, and 2 head movement components captured via PCA by FreeSurfer. In lines 102 to 104, we’re de-noising the time series in the linear model.

The relevant FFT, connectivity, and reliability statistics for the de-noised data are calculated in line 107. Essentially, this pattern repeats itself, with alternative filters being thrown into the pipeline every now and then. The comments in the code should be enough to give you a grasp of which filters were applied to the data and what exactly was used to de-noise it. However, we definitely want to take a closer look at the end of the loop because an important trick happens here. We’ll illustrate this trick with line 145, although it’s done similarly in other places with different filters.

Here’s the command: noise1c = cleanCon\_1 - clean\_Confilter\_1;

cleanCon\_1 is a time series that has been both de-noised and detrended. clean\_Confilter\_1 is the time series that has also had a low-pass filter applied to it. When we take the difference between the two, what’s left is the high-frequency noise. It’s basically the residue that remains. We subject these junk time series to an FFT analysis to see if they contain signals in cognitive frequency ranges. If they do, then the filter for fMRI is definitely not up to snuff. The FFT of the pure noise time series (junk) is captured in line 149. We handle the other filter in exactly the same way. Lines 222 and 223 are also important because here we determine the empirical BOLD response, also known as the predictor function. For this, we use a special function called extract\_section\_single\_withJitter.m, which takes the jitter of the time series into account.

Alright, enough about that. The key takeaway is to keep in mind that within the loop, core statistics from various pipelines are being collected. The pipelines use different types of filters, with their parameters set in lines 28 to 50. For the SG pipeline, we’re only using SG filters, while the SPM pipeline gets a bit more complicated. For detrending, we use a discrete cosine transform with a length of 128 seconds. For the low-pass filter, we either use a Gaussian filter (2.48 seconds) or an HRF filter. We also looked at pipelines with conventional FFT filters, like those used in resting state fMRI. However, we got a bit embarrassed, so we’re going to self-censor and not report on that, lest the whole resting state crowd comes after us.

The cell formats collected in the pipeline aren’t equally suitable for all applications. That’s why they’re reshaped into other formats in lines 235 to 261. The formats ending with the suffix ‘\_bild’ are, for example, suitable for visualization as a connectome.

As promised, let’s keep it brief and focus on the key results. The ultimate test of whether a low-pass filter is fit for purpose comes when the FFT of pure noise signals shows no power in task-relevant frequency ranges. To figure that out, we first need to identify which frequency ranges are task-relevant. The best thing to do is to capture the average empirical BOLD response, referred to here as the predictor function. This happens in line 272. Next, we calculate the FFT of the predictor function in line 277. Now we want to compare the FFTs of the pure noise with the FFT of the predictor function. If there are overlaps in the task-relevant frequency ranges, that’s pretty suspicious and suggests that the low-pass filters aren’t doing their job. The mean FFTs of the pure noise time series are calculated in lines 284 to 297. Then, in lines 303 and 304, we create the plots. The whole story is neatly summarized in Figure 8 of the main paper. The main result is that the SG low-pass filter with a window size of 15 and an 8th-degree polynomial is clearly superior to all other filters. In lines 312 and 313, we calculate the autocorrelations of the various pipelines. This is reported in Table 1 of the main paper. At the top of Table 1, you’ll find the autocorrelations of the different pipelines. Below, the autocorrelations of the predictor functions are displayed. It’s clear that our favored SG filter (15/8) has autocorrelations at Lag 2 to Lag 4 that are significantly lower than those of the predictor function. So, we can say that the signals aren’t any prettier than the original. The same can’t be said for the classic Gaussian and HRF filters, which show significantly higher autocorrelations in some areas.