A Comparison of Linear Time-Invariant System Identification Techniques

Alexander Murray

alexander.murray@students.fhnw.ch

alex.murray@gmx.ch

November 17, 2016

Abstract

System identification is an important topic in control theory. Small errors in the derivation of a real world system's transfer function can lead to large deviations later on. Worse, it can lead to unstable control loops. It is therefore crucial to create an accurate model.

In this report, two system identification techniques proposed by L. Sani and P. Hudzovic will be analysed and compared to one another. It will be shown that a combination of L. Sani's characterisation and P. Hudzovic's transfer function formula produces the most accurate results. Furthermore, an extension to both methods will be proposed using a least square fit, allowing the obtained system to be further refined.

Contents

1 Introduction

The need to identify the dynamic behaviour of an unknown, arbitrary system or *plant*, as it is referred to in control theory, is a common problem. The plant's behaviour dictates how the *controller* must behave in order to create a stable *control loop*. The ability to accurately and – preferrably automatically – deduce this behaviour is crucial. Small errors in the identification process can lead to large deviations later on. Worse, it can lead to unstable control loops.

The focus of this report will be the identification of time-invariant (LTI) systems that don't exhibit overshoot or undershoot. specifically, two fairly similar methods of system identification proposed by L. Sani[1] and P. Hudzovic[2] will be analysed and compared to one another using MATLAB.

Furthermore, an extension to both methods involving a least squares fit will be proposed, allowing the results of both methods to be further refined.

2 Theory

The effect of any invariant linear system (LTI) on an arbitrary input signal is obtained by convolution of the input signal with the system's impulse response function. In a LTI system, the output of the system $y(t)$ for an input $x(t)$ can be obtained by the convolution integral:

$$y(t) = g(t) * x(t) = \int_0^t g(t-\tau)x(\tau)\,d\tau \quad (2.1)$$

where $g(t)$ is the *impulse response* of the system. That is, $g(t)$ is the output of the system with an input $x(t) = \delta(t)$, where $\delta(t)$ is the Dirac delta. The impulse response completely characterizes the dynamic behaviour of the system.

Applying the Laplace transform to the convolution integral (equation 2.1) we obtain equation 2.3:

$$\mathscr{L}[y(t)] = \mathscr{L}[g(t)x(t)] = \mathscr{L}[g(t)]\,\mathscr{L}[x(t)] \quad (2.2)$$

or in simple expression:

$$Y(s) = G(s)X(s) \quad (2.3)$$

where $Y(s)$, $G(s)$ and $X(s)$ are the Laplace transforms of $y(t)$, $g(t)$ and $x(t)$ respectively. A Transfer Function (TF) is the mathematical representation of the relation between the input and output of a system. In a LTI system, TF can be expressed as the ratio of the Laplace transform of the output and the input, and corresponds to the Laplace transform of the impulse response $G(s)$.

$$G(s) = \frac{Y(s)}{X(s)} \quad (2.4)$$

In order to obtain $G(s)$ from an unknown system, a signal in the form of the Dirac delta function must be applied to the input of the system and its output must be measured. Unfortunately, it is very hard to do this in most practical cases, due to:

1. Difficulty of generating a Dirac delta function (infinite amplitude, zero time).
2. Any finite approximation of the Dirac delta function will cause an extremely small and hard to measure response signal.

A far more practical method is to measure the *step response* instead. A step function is easier to create in the physical world. The derivative of the resulting measured signal will very closely resemble its theoretical impulse response.

2.1 The Model

The model used by both Hudzovic[2] and Sani[1] approximate the step response of a plant by using a series of PT1 elements multiplied together with varying time constants T_k to form a PTn element, $G(s)$. This is defined as:

$$G_n(s,r) = y_0 + K_s \prod_{k=1}^{n} \frac{1}{1 + s \cdot T_k(r)} \quad (2.5)$$

where the scale factor, K_s, is defined as:

$$K_s = \frac{xa(\infty)}{xeo} \quad (2.6)$$

The transfer function in equation 2.5 serves as a basis to model the step response of many systems.

Rather than individually having to find the time constants $T_1 \ldots T_n$ – the effort of which would greatly increase with the order n – the two methods of P. Hudzovic and L. Sani instead calculate these constants using a common function $T_k(r)$.

The approach proposed by P. Hudzovic[2] for $T_k(r)$ is:

$$T_k(r) = \frac{T}{1 - (k-1)r} \tag{2.7}$$

where the constant r must be confined to the interval $0 \leq r \leq \frac{1}{n-1}$.

The approach proposed by L. Sani[1] for $T_k(r)$ is:

$$T_k(r) = T \cdot r^k \tag{2.8}$$

where the constant r must be confined to the interval $0 \leq r \leq 1$.

As can be seen, in both cases, the problem has been reduced to finding appropriate values for n, T and r such that the step response of $G_n(s, r)$ approximates the data acquired from the plant as closely as possible.

2.2 Step Response Characterisation

The first step to calculating T, r, and n is to understand how the order n influences the shape of the transfer function's step response.

It is important to realise that no matter how a step response function is scaled or offset (defined by the parameters K_s for amplitude, T for time scale, x_0 and y_0 for offset) the actual *shape* always remains the same. Thus, only its shape tells us something about its complexity.

Therein lies the key. A method needs to be devised for determining how "simple" or how "complex" a step response is – independent of scale and offset – before it is possible to start modelling and fitting a system to it.

This "complexity" is **directly related** to the required order n of the transfer function.

P. Hudzovic's Approach

P. Hudzovic proposed the following method (see figure 1):

- Find the point of inflection of the step function. This typically involves calculating the derivative and searching for a maximum.
- Place a tangent in said point and find the intersections with the minimum and maximum horizontal lines.
- The distance between the two intersections is referred to as T_g, and the distance between the minimum intersection point and the beginning of the signal is referred to as T_u.

The "complexity" of the step response is defined by the ratio of T_u and T_g and is written as:

$$\text{plant}_{Tu/Tg} = \frac{T_u}{T_g} \tag{2.9}$$

L. Sani's Approach

L. Sani proposed a different method (see figure 2): Determine the times t_{10}, t_{50} and t_{90} required for reaching the values at $10\,\%$, $50\,\%$ and $90\,\text{percent}$ respectively.

The "complexity" of the step response is defined by the ratio of $t_{90} - t_{10}$ and t_{50}, otherwise referred to as λ:

$$\text{plant}_\lambda = \frac{t_{90} - t_{10}}{t_{50}} \tag{2.10}$$

Short Visual Explanation

Visually, one can see how decreasing T_g in figure 1 causes the step response to become steeper (i.e. it becomes more "complex" and thus requires a higher order n) and the value of plant_{T_u/T_g} in equation 2.9 increases. Similarly, decreasing the difference $t_{90} - t_{10}$ in figure 2 also causes the step response to become steeper and causes the value of plant_λ increases.

On the other hand, one can also see how increasing T_u and t_{50} increases the delay time of the step response, which similarly leads to a higher "complexity", and thus, a higher order n.

How n is calculated will become clear in the next section.

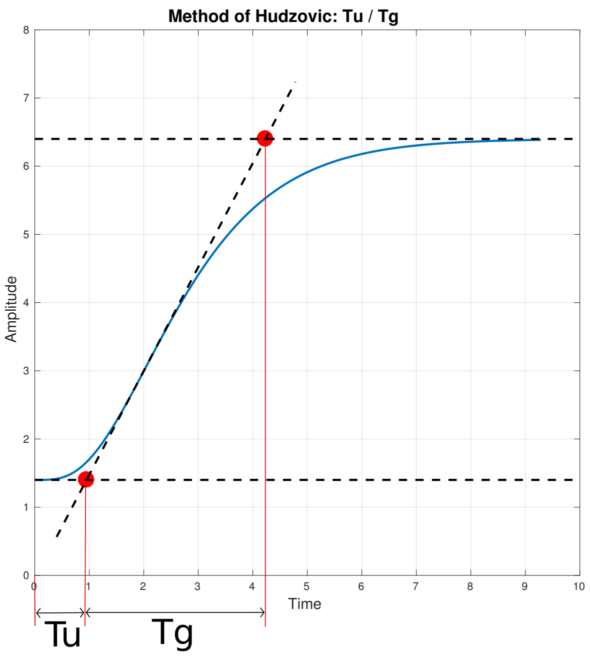


Figure 1: Method of P. Hudzovic, determine Tu and Tg

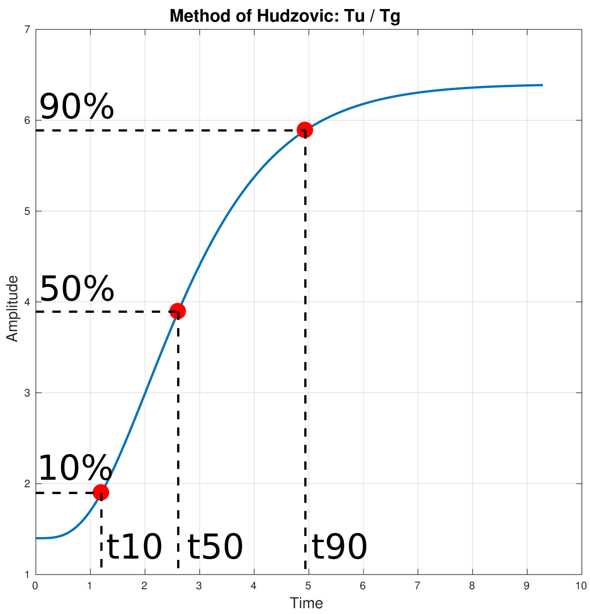


Figure 2: Method of L. Sani, determine t10, t50 and t90

2.3 Approximating the Step Response

So far, two methods for characterising step response functions were illustrated, namely the method proposed by P. Hudzovic (equation 2.9) where you calculate T_u and T_g, and the method proposed by L. Sani (equation 2.10) where you calculate t_{10}, t_{50}, t_{90}.

Furthermore, two similar methods for constructing a transfer function $G_n(s, r)$ (equation 2.5) were shown, again, one proposed by P. Hudzovic (equation 2.7) and one proposed by L. Sani (equation 2.8) for calculating the individual time constants $T_1 \ldots T_n$ based on the input parameters r and T.

By combining the various approaches with each other, 4 different ways for determining $G_n(s, r)$ exist.

It is not possible to *directly* calculate appropriate values for n, T and r when given an unknown step response function, however, by using equations 2.5, 2.7 and 2.8, it is possible to construct a lookup table by calculating a (theoretically) infinite number of step responses in function of n and r, characterise their step response using equations 2.9 or 2.10 (i.e. determine their "complexity"), and perform a reverse lookup on those results to find the parameters n, r. This method of reverse lookup works because – as you will soon see – the lookup curves are monotonically increasing/decreasing.

In practice, it is sufficient to calculate about 50 step responses for each order n and interpolate between those values when performing the lookup.

As mentioned earlier, a remarkable observation is that the parameters r and n are independent of time and amplitude (and offset); that is, the normalised step response does not change its shape when the parameter T is changed. This is fantastic, because it allows us to eliminate a dimension from the lookup table.

If $T = 1$, $K_s = 1$ and $y_0 = 0$, equations 2.5, 2.7 and 2.8 can be used to calculate a series of transfer functions $G_n(s, r)$, their time domain step responses $g_{r,n}(t)$ can be calculated, and they can be characterised by calculating g_{T_u/T_g} and g_λ.

These two values alone aren't yet enough. The result will still be dependent on T_g or t_{50}. In order to fully "denormalise" the result, g_{1/T_g} and $g_{1/t_{50}}$ must also be calculated for each step response.

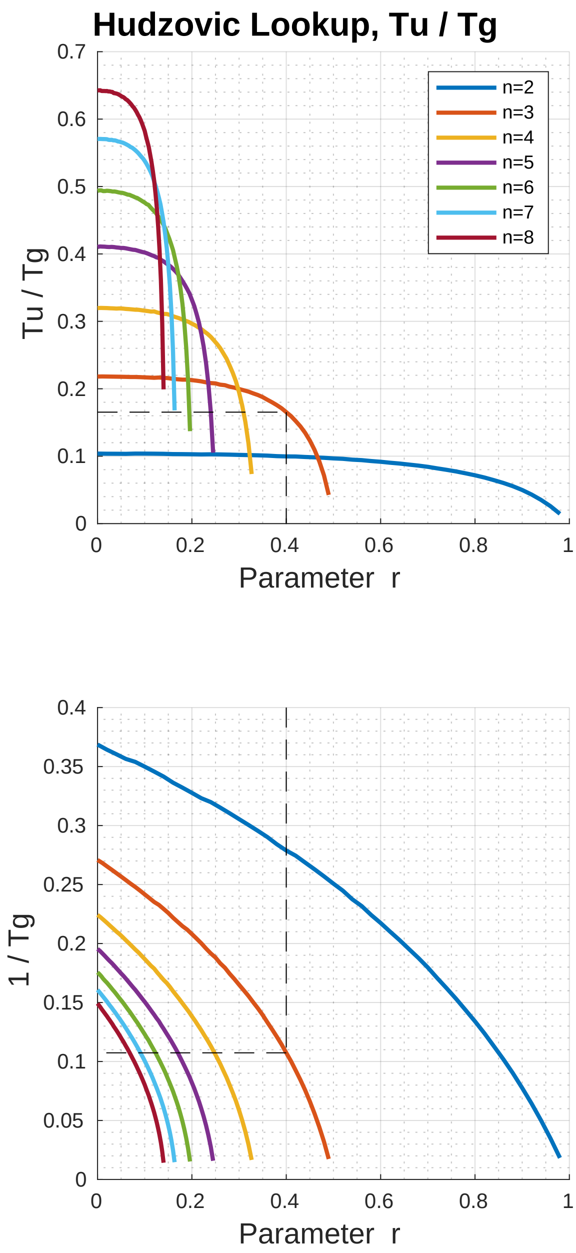


Figure 3: P. Hudzovic lookup curves. Top: Tu/Tg, bottom:1/Tg

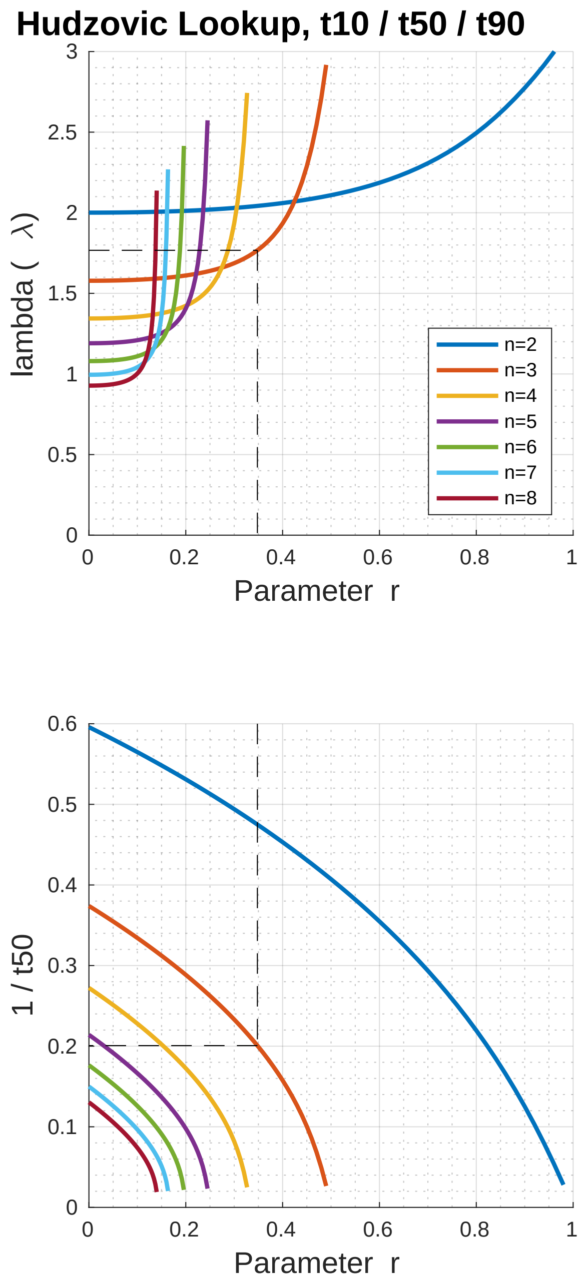


Figure 4: P. Hudzovic lookup curves. Top: (t90-t10/t50), bottom: 1/t50

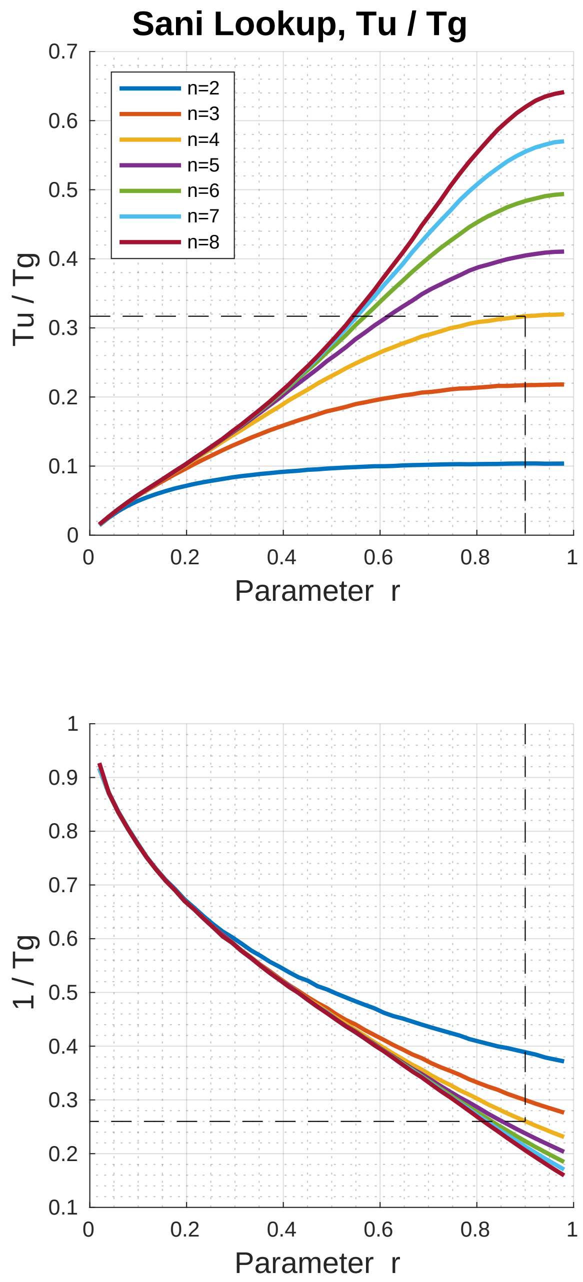


Figure 5: L. Sani lookup curves. Top: Tu/Tg, bottom: 1/Tg lookup

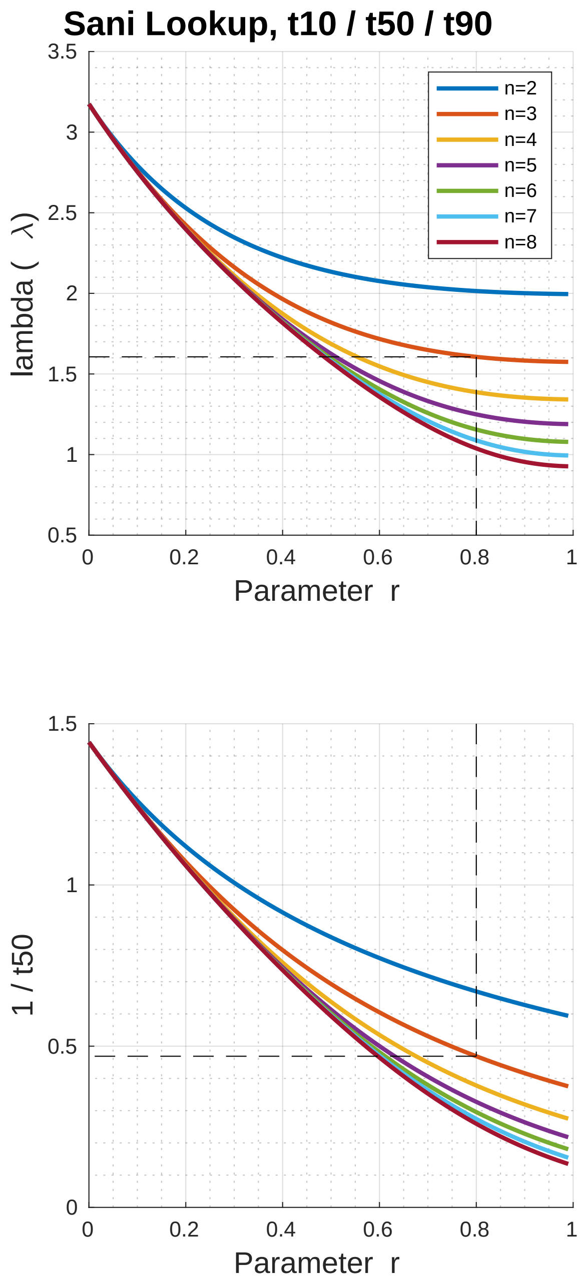


Figure 6: L. Sani lookup curves. Top: (t90-t10)/t50, bottom: 1/t50

The various characterisations of the step response function $g(t)$ can all be expressed as functions of r and n:

$$g_{T_u/T_g} = g_{T_u/T_g}(r,n) \tag{2.11}$$
$$g_{1/T_g} = g_{1/T_g}(r,n) \tag{2.12}$$
$$g_\lambda = g_\lambda(r,n) \tag{2.13}$$
$$g_{1/t_{50}} = g_{1/t_{50}}(r,n) \tag{2.14}$$

Using P. Hudzovic's approach from equations 2.5 and 2.7 the four functions $g_{T_u/T_g}(r,n)$, $g_{1/T_g}(r,n)$, $g_\lambda(r,n)$ and $g_{1/t_{50}}(r,n)$ are evaluated. The figures 3 and 4 are obtained.

Similarly, using L. Sani's approach from equations 2.5 and 2.8 we again evaluate the same four functions and obtain the figures 5 and 6.

What these plots show beautifully is that the higher the order n, the steeper – or more "complex" – the step response becomes (smaller values of T_g or $t_{90} - t_{10}$ mean faster rise times of the step responses). Another important thing to observe is how lower orders of $G_n(s,r)$ aren't able to rise as fast as higher orders are able to, **regardless** of r and T. This can also be seen in all four figures: Lower orders cannot reach ratios of T_u/T_g or λ that higher orders can.

Calculating T, r, n, by using Lookup Curves

As already mentioned, determining the complexity of a step response is directly related to the required order n of the model. Finding n is now a simple matter of computing plant_{T_u/T_g} or plant_λ and iterating through the different curves either in figure 3, 5, 4, or 6 until a value of n is found that satisfies one of either conditions:

$$g_{T_u/T_g}(r,n)|_{r=max} \leq plant_{T_u/T_g} \tag{2.15}$$
$$g_\lambda(r,n)|_{r=max} \leq plant_\lambda \tag{2.16}$$

Ideally, n should be as small as possible.

With the parameter n defined, the next step is to find the intersection point of the horizontal line that passes through either plant_{T_u/T_g} or plant_λ and $g_{T_u/T_g}(r,n)$ or $g_\lambda(r,n)$, respectively. This will yield parameter r. This can be achieved by solving a simple line intersection equation and plugging in the locations of the two horizontal lines.

The last parameter, T, can finally be determined by evaluating $\text{plant}_{g_{1/T_g}}(r,n)$ or $\text{plant}_{t_{50}} \cdot g_{1/t_{50}}$ Graphically, this equates to finding the intersection point of the vertical line going through r and the function $g_{1/T_g}(r,n)$ in figure 3 and multiplying the result by T_g.

Each of the figures 3, 5, 4, and 6 contain a dashed line, which is supposed to demonstrate an example lookup, to help visualise the process.

Calculating T, r, n, by using Interpolation Formulae

Instead of having to generate the lookup curves seen in figure 6, Sani[1] included two formulae that very closely approximate these curves. They are:

$$\frac{t_{50}}{T} = \log(2) - 1 + \frac{1-r^{n+1}}{1-r} \tag{2.17}$$

$$\frac{t_{90}-t_{10}}{T} = 1.315\sqrt{3.8\frac{1-r^{2(n+1)}}{1-r^2} - 1} \tag{2.18}$$

Using the formula for calculating λ in equation 2.10, the interpolation formulae can be made independent of T:

$$\frac{t_{90}-t_{10}}{t_{50}} = \frac{1.315\sqrt{3.8\frac{1-r^{2(n+1)}}{1-r^2} - 1}}{\log(2) - 1 + \frac{1-r^{n+1}}{1-r}} \tag{2.19}$$

Plotting the functions $\frac{t_{90}-t_{10}}{t_{50}}$ and $\frac{1}{t_{50}}$ yields nearly identical curves as seen in figure 6.

The benefit of using these interpolation formulae over a lookup table is ease of implementation, lower memory footprint and flexibility. Orders of n can be chosen arbitrarily, whereas with lookup tables an entry must exist for every order that should be supported.

Finding the parameter n is identical to the previous example using equation 2.16, except that instead of using a lookup table, the function in equation 2.19 is evaluated.

Finding the parameter r can be achieved by performing a binary search on the function in equation 2.19. This works because the function is monotonically decreasing.

Finding T is a simple matter of solving equation 2.17 for the parameter T and evaluating it using the found parameters n, r as well as the plant's t_{50} value $\text{plant}_{t_{50}}$.

2.4 Least Squares Fit

Given a discrete set of data points which represent a measured step response of an unknown system, the input parameters of the function $G_n(s, r)$ can be tweaked such that the squared error between its step response and the input data is minimised. The squared error is computed using:

$$S = \sum_i (g(t_i) - y_i)^2 \tag{2.20}$$

where $g(t)$ is the inverse Laplace transform of $\frac{G_n(s,r)}{s}$ (the time domain step response) and (t_i,y_i) are data points of the measured step response.

By fitting a system $G_n(s, r)$ to the input data, it is possible to further refine the results obtained by the two methods mentioned thus far, or otherwise find optimal values for T and r when dealing with noisy input data.

The possibility of finding local minima exists. It is therefore advised to first use one of the four previous methods to find optimal initial values for T, r and n before performing the fit.

It will be shown that the least squares fit approach will yield the most accurate results by orders of magnitude. The downside to this method, of course, is the large amount of computation time required.

3 Simulations

A comprehensive set of subroutines required to compute and evaluate the transfer functions outlined in this report can be obtained from GitHub[3].

The subroutines are located in the folder *matlab/mfunctions*, along with various utility functions. The folder must be appended to MATLAB's path like so:

```
addpath([pwd,'/mfunctions']);
```

This section provides a short overview of what each subroutine does and how.

Pre-processing

Typically, the input data is never perfect. It might contain noise or it might not have equispaced time values. The function *preprocess_curve* returns equispaced time values and smooths the signal.

```
% Smooth input data and generate equispaced
% time vector
[x, y] = preprocess_curve(xr, yr);
```

This is achieved by using a combination of MATLAB's *smooth()* function and a custom sliding average function to smooth the beginning and ends of the signal.

Characterising the curve

With the data prepared (preprocessed), it is now possible to calculate T_u/T_g or t_{10}, t_{50}, t_{90}, depending on what you wish to do next. The function *characterise_curve()* can handle both cases for you, like so:

```
% Characterises the curve, either using
% Hudzovic's method or Sani's method.
[Tu, Tg] = characterise_curve(x, y);
[t10, t50, t90] = characterise_curve(x, y);
```

Determining T_u, T_g is achieved by calculating the derivative of the data to find the point of inflection. The maximum and minimum of the signal is determined by taking the value of the first and last element in y.

Determining t_{10}, t_{50}, t_{90} is achieved by using MATLAB's *spline()* function for higher accuracy and intersecting the data at 10%, 50% and 90% amplitude. In the case of noisy input data, if the spline fails, the fallback method is to simply find the nearest point. The minimum and maximum of the input signal is again determined by using the first and last element in y.

Because L. Sani's approach heavily relies on accurately determining the start and end values of the input step response function, and it was found that noisier input signals would significantly throw off correctly determining t_{10}, t_{50}, t_{90}, it is possible to override the default min/max values by passing in a third argument:

```
% Override Sani's min/max.
ymax = 37;  % Step function stops at 37
ymin = 15;  % Step function starts at 15
[t10, t50, t90] = characterise_curve(x, y, [ymin, ymax]);
```

The third argument is only valid for L. Sani's method.

In the case of noisy signals, it is usually better to determine the beginning and end values of the step response function manually.

Calculating T, r and n

With T_u, T_g or t_{10}, t_{50}, t_{90} determined, the three constants T, r and n can be calculated using either P. Hudzovic's or L. Sani's transfer function (equations 2.7 or 2.8). For this, the two functions *sani_lookup()* and *hudzovic_lookup()* may be used.

Both of these functions accept either T_u/T_g or t_{10}, t_{50}, t_{90} as parameters. Depending on which one you choose to use, the function will perform a different lookup:

```
% Hudzovic method, Sani method, and their permutations
[T, r, n] = hudzovic_lookup(Tu, Tg);
[T, r, n] = hudzovic_lookup(t10, t50, t90);
[T, r, n] = sani_lookup(t10, t50, t90);
[T, r, n] = sani_lookup(Tu, Tg);
```

Note how it's also possible to use the t_{10}, t_{50}, t_{90} approach with Hudzovic's method, and similarly, use T_u/T_g with Sani's method.

When calling these functions for the first time, they will spend some time generating the lookup curves discussed in the theory section. This can take about a minute. They are saved to disk and loaded again if available, so all subsequent calls will be fast.

The *sani_lookup()* function will use the interpolation formulae (equation 2.19) when passing t_{10}, t_{50}, t_{90}. All other combinations will have to use lookup curves.

Calculating the Transfer Function

Once T, r and n are obtained, the two functions *hudzovic_transfer_function()* and *sani_transfer_function()* will help convert those three constants into a continuous transfer function:

```
% Calculating the transfer functions
G_hudzovic = hudzovic_transfer_function(T, r, n);
G_sani = sani_transfer_function(T, r, n);
```

Of course, if the parameters T, r and n were determined using *sani_lookup()* then the function *sani_transfer_function()* must be used. The same is true for Hudzovic.

Once the transfer function is obtained, one can view the step response by using *step()*:

```
% Plot step response
[g, t] = step(G_hudzovic);
plot(t, g);
```

Fitting

So far, the typical work-flow for calculating the transfer function looks somewhat like the following code:

```
% Hudzovic, Tu/Tg
[Tu, Tg] = characterise_curve(x, y);
[T, r, order] = hudzovic_lookup(Tu, Tg);
G = hudzovic_transfer_function(T, r, order);
```

To further refine the result, it is possible to perform a least squares curve fit on a result obtained by either method using the **original** (non-smoothed) data. This can be achieved with the functions *hudzovic_fit()* and *sani_fit()*.

```
% Hudzovic, Tu/Tg, with fitting
[Tu, Tg] = characterise_curve(x, y);
[T, r, n] = hudzovic_lookup(Tu, Tg);
[T, r] = hudzovic_fit(T, r, n, xr, yr);
G = hudzovic_transfer_function(T, r, order);
```

Here, *xr*, *yr* contain the "raw" data, and *x*, *y* contain the pre-processed (smoothed) data.

Similarly, it is possible to further refine a result obtained by L. Sani's method using the function *sani_fit()*:

```
% Hudzovic, Tu/Tg, with fitting
[t10, t50, t90] = characterise_curve(x, y);
[T, r, n] = sani_lookup(t10, t50, t90);
[T, r] = sani_fit(T, r, n, xr, yr);
G = sani_transfer_function(T, r, order);
```

The fit is currently not able to determine the required order by itself, so it is necessary to first call *characterise_curve()* on the smoothed data to retrieve the order. A nice side effect of doing this is it also gives you good starting values for T and r. This avoids the possibility of falling into a local minimum while fitting the data.

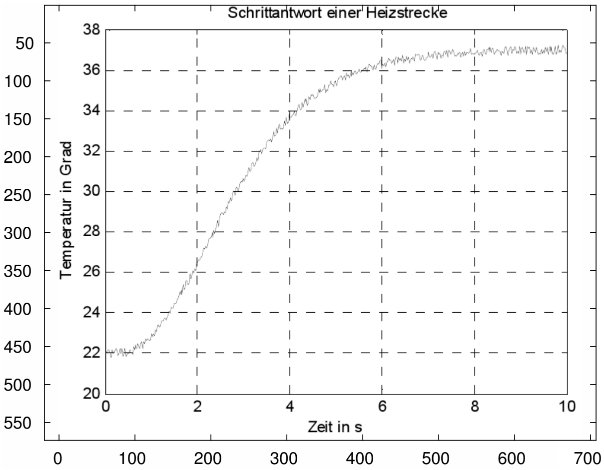
3.1 Reading curves from images

More often than not we are required to measure T_u and T_g by hand from visual plots, without access to the underlying data. To overcome the need to print out the plots onto paper and measure the data by hand, an algorithm for extracting this data from an image was developed.

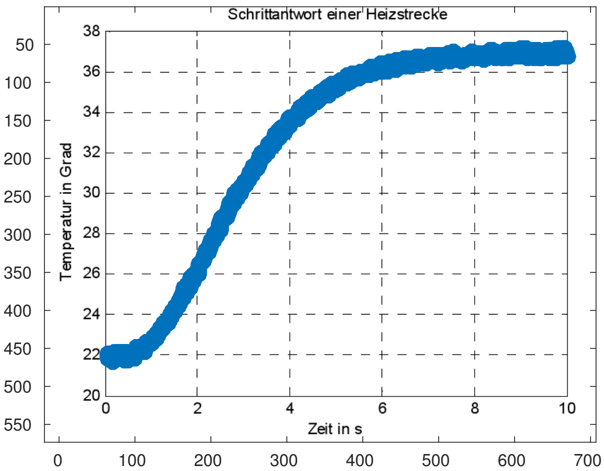
The algorithm is quite simple: The image data is imported into HSV colour space. In the majority of cases, the grid and background will be white, black or some grey-scale. The data on the other hand will have a specific colour, which allows us to first detect what the colour is and then filter the image by hue to get all data points that have a similar colour.

If the plot happens to be black and white with no colour, then it is possible to filter the data using the "value" component from the HSV data.

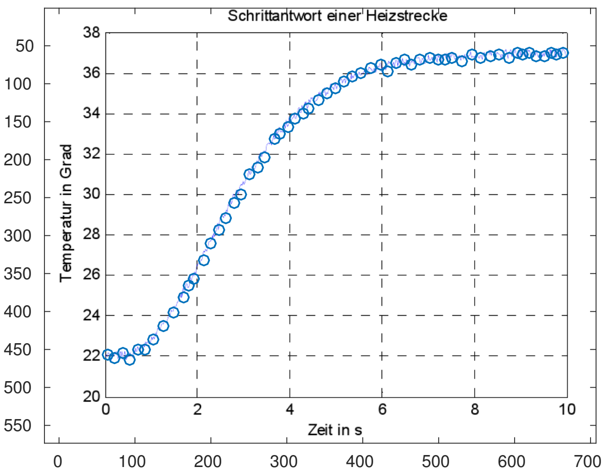
The resulting data is usually noisy, but statistically evenly distributed, which makes decimation very easy: Just select every nth data point. The data from 7b was decimated by a factor of 50, resulting in figure 7c.



(a) Typical image of a plant



(b) XY scatter of detected data



(c) Decimated by 50

Figure 7: Process of importing curve data from an image

4 Results

A number of comparisons and evaluations were performed on the 6 methods using the MATLAB functions described above. The simulation code can also be found on GitHub[3] in the directory *matlab/*.

4.1 Comparison of Accuracy vs Order

It would be interesting to investigate whether the accuracy changes depending on the transfer function's order n.

For each order n 100 random PTn transfer functions $G_n(s)$ were generated in the form of:

$$G(s) = \prod_{k=1}^{n} \frac{1}{s + T_k} \tag{4.1}$$

where n was a random integer in the range of $[2..8]$ and T_k was a random number in the range of $[1..9]$.

The step response of each $G_n(s)$ was fed into each method. The step responses of each resulting transfer function was then compared to the original input function's step response and the root mean square error (RMSE) was calculated. The 100 errors were then averaged, resulting in a final error value for each method, for each order.

The data portrayed in figure 8 shows these errors.

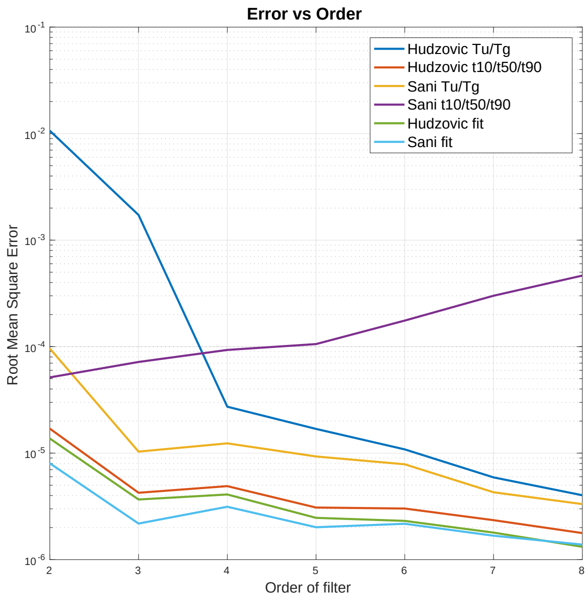


Figure 8: The mean square error (MSE) of each method to a randomly generated step response, in function of filter order

Conclusions

As expected, the fitted methods yield the most accurate results. The Sani fit appears to be better than the Hudzovic fit.

The most accurate non-fitted method is Hudzovic's transfer function in combination with Sani's characterisation (orange curve).

An interesting observation is that the method proposed by L. Sani[1] using the t_{10}, t_{50}, t_{90} characterisation (purple curve) performs worse and worse the higher the order, whereas all other methods perform better and better. The exact reason as to why this happens is a consequence of using the interpolation formulae proposed by L. Sani (equation 2.19) rather than using lookup curves.

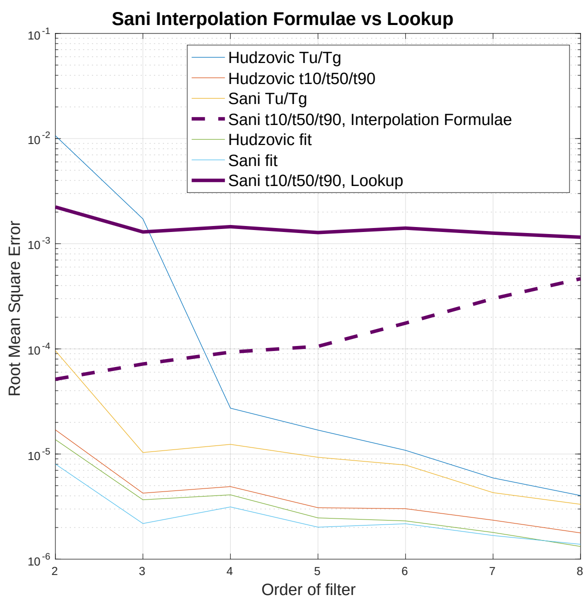


Figure 9: Interpolation formulae proposed by L. Sani compared to "brute force" calculating the individual t10, t50, t90 parameters and creating lookup curves

To see if this is indeed the cause, the MATLAB code was adapted such that L. Sani's method for determining r, T, n based on the input data t_{10}, t_{50}, t_{90} used a lookup curve instead of using the interpolation formulae. The same simulation was performed again with the modified code.

The result can be seen in figure 9. The dashed purple line is the same purple curve from figure 8. The continuous purple line shows the result of the modification.

It appears that by using lookup curves the resulting transfer function is less accurate than if interpolation formulae were used.

4.2 Comparison of Accuracy vs Noise

To see how noise affects the accuracy of each method, a number simulations were performed, where the noise amplitude on the input step response function was continuously increased and the deviation for each method was calculated.

In the first simulation – see figure 10 – the input function was constructed using a PT4 element by using Hudzovic's formula with parameters $r = 0.5$, $n = 4$, and $T = 1$. This particular function was chosen because it yielded the most stable results in previous tests.

1000 iterations were performed, starting at 0% input noise and ending at 35% input noise. The ever increasing noisy input function was fed into each of the 6 methods. Each method's result was then compared to the original, non-noisy input function by calculating the root mean squared error.

The resulting error data was smoothed using a sliding average filter of width 10, resulting in figure 10.

One issue that arose later with this simulation was that a systematic bias exists because all of the calculations are based on a single input function. There exist a certain combination of time constants T_k that will "align" better with L. Sani's transfer function (equation 2.8) than with P. Hudzovic's transfer function (equation 2.7) and vica-versa.

In this case, the results in figure 10 happened to use an input function that favoured P. Hudzovic's method. This should be kept in mind when interpreting the data.

In an effort to eliminate this bias, the simulation was repeated 500 times and a random input function was generated using equation 4.1. This way, no single method was favoured. The 500 resulting errors were averaged and can be seen in figure 11.

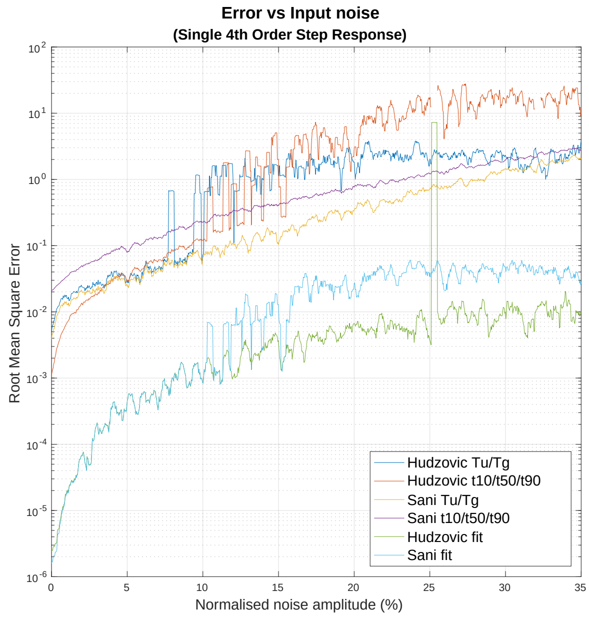


Figure 10: The root mean square error (RMSE) of each method to the original 4th order step response function, in function of normalised input noise amplitude.

Due to the intensive nature of this simulation, the fitting methods were omitted, because they took too long to compute. It is fairly certain, though, that they would perform better than the other methods.

Finally,by using the same PT4 element from the first simulation as an input function, the third simulation covers an input noise range of 0% to 200%. The results of this simulation are visualised in figure 12.

Conclusions

An immediate conclusion to be drawn is that the two fitting methods (cyan and olive curves in figure 10) yield more accurate results than any of the other methods by an order of magnitude. This is to be expected, of course. While the Sani fit proves to be the most accurate fitting method for non-noisy signals, as can be seen by zooming into the bottom left of the plot, the Hudzovic fit proves itself to be the most accurate for very noisy signals.

Also as expected, the accuracy of each method appears to get less accurate the more noisy the input signal is.

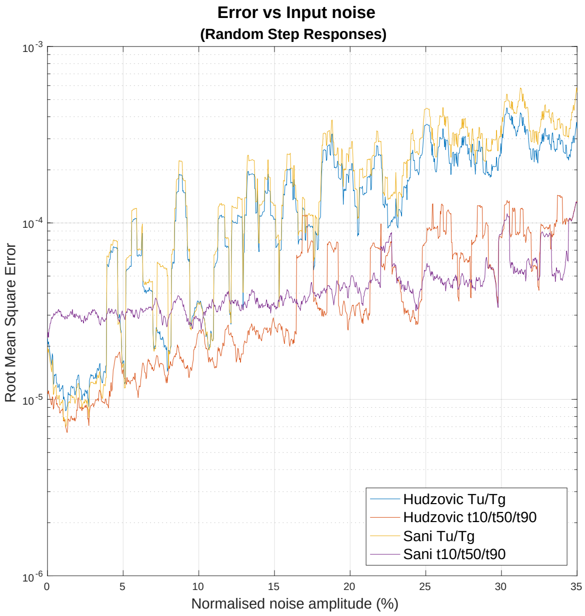


Figure 11: The root mean square error (RMSE) of each method to a randomly generated nth order step response function, in function of normalised input noise amplitude. The fitted curves were omitted due to their time consuming nature.

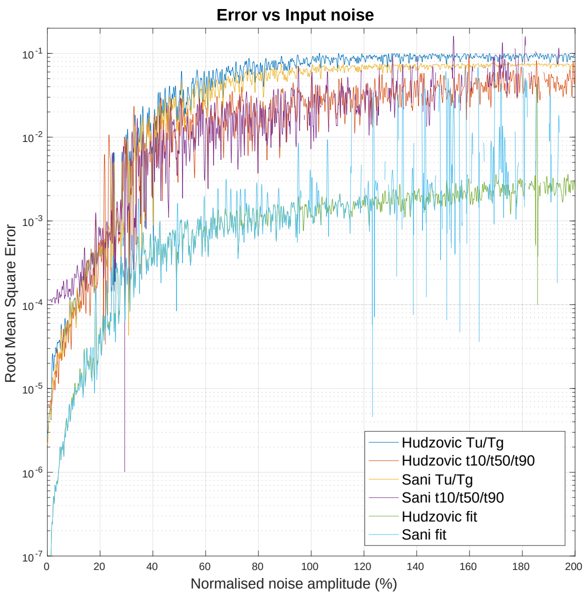


Figure 12: Long term view of the root mean square error (RMSE) of each method to the original 4th order step response function, in function of normalised input noise amplitude.

As already mentioned, the data in figure 10 should be interpreted with caution, due to a systematic bias. With this in mind, an unexpected result in figure 10 is that Hudzovic's characterisation approach using T_u/T_g shows to be more accurate for noisy signals than Sani's characterisation approach using t_{10}, t_{50}, t_{90}. This can be observed by comparing the (orange) and (blue) curves, or by comparing the (purple) and (yellow) curves. This result is unexpected because of how T_u/T_g is calculated: The derivative of the signal must be computed in order to find the point of inflection (see theory section). The derivative of a noisy signal is, of course, an even noisier signal, which should make correctly calculating the point of inflection much less accurate than calculating the threshold values t_{10}, t_{50}, t_{90}.

However, the data in figure 11 and in figure 8 directly contradicts this conclusion. When taking the average of lots of simulations using random input step response functions, it appears that the methods using the T_u/T_g characterisation perform worse than those using t_{10}, t_{50}, t_{90}. This can especially be seen in figure 11. At around 4% noise amplitude, both methods that use the T_u/T_g characterisation start jumping around eratically.

The most accurate non-fitting method is Hudzovic's transfer function in combination with Sani's characterisation method, t_{10}, t_{50}, t_{90}. This appears to be the case in both simulations.

4.3 Comparison of Fit vs Non-Fit

The method proposed by P. Hudzovic is compared to itself with and without a least squared fit of the parameters T and r. Similarly, the method proposed by L. Sani is also compared to itself with and without a least squared fit of T and r.

The input curve in Sani's case was a 4th order Hudzovic transfer function with $r = \frac{1}{6}$ and $T = 1$. The input curve in Hudzovic's case was a 4th order Sani transfer function with $r = 0.5$ and $T = 1$.

The reason for choosing an "unsuitable" input curve was to make it impossible for the method under test to calculate a perfectly correct transfer function such that the error would be 0.

The test was performed twice for each method, once with no input noise and once with random noise

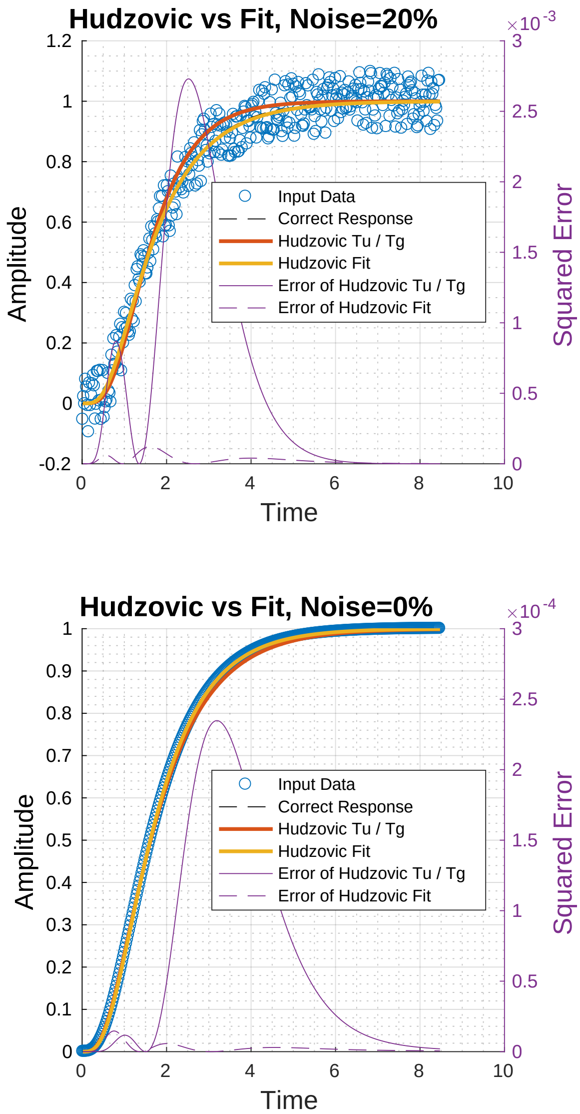


Figure 13: Comparison of a normal Hudzovic Tu/Tg lookup to a least square refinement. Order n=4, Noise=20% and 0%

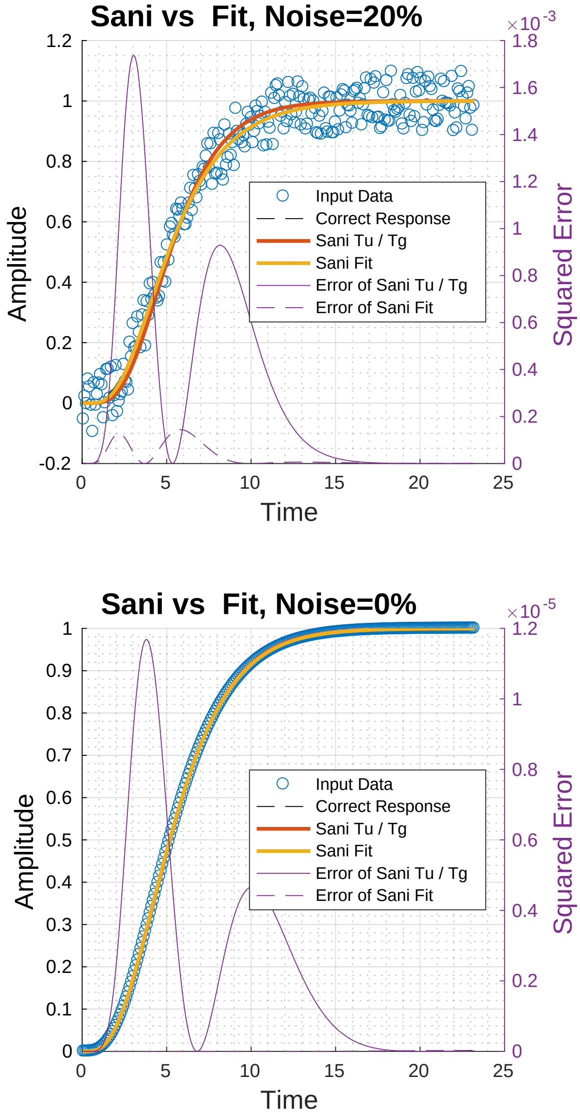


Figure 14: Comparison of a normal Sani Tu/Tg lookup to a least square refinement. Order n=4, Noise=20% and 0%

(20% normalised amplitude) modulated onto the input signal.

The results can be seen in figures 13 and 14. The blue circles show the raw input data to the method under test. The red curve shows the step response of the resulting transfer function *without* fitting and the yellow curve shows the step response *with* a least squares fit.

Further, the two purple curves on a second Y axis show the *squared error* of the output function to the original non-noisy input function of each data point. The dashed purple line is the error of the fitted result whereas the continuous purple line is the error of the non-fitted result.

It is fairly clear that in all four cases the least square fit yields more accurate results. The maximum squared error of the non-fitted methods (with 20% input noise) is around $2x10^{-3}$, whereas the maximum squared error of the fitted methods is around $1x10^{-4}$. This leads to an improvement factor of about 5.

5 Discussion

The two methods proposed by L. Sani and P. Hudzovic and two combinations thereof were investigated and compared to one another.

The most accurate method – in general – turns out to be P. Hudzovic's transfer function in conjunction with L. Sani's characterisation method (by measuring t_{10}, t_{50}, t_{90} instead of T_u/T_g). This is true for all orders of n and this is true for noisy and non-noisy input functions.

Of course, by performing a least-square curve fit, an even more accurate result can be obtained.

One factor that was not considered is how many data points are required in the lookup curves to yield accurate enough results. The simulations in this report used 50 data points. It would be worth investigating this further to see how higher resolution (or lower resolution) lookup curves affect the accuracy of each method.

P. Hudzovic's characterisation method (by measuring T_u/T_g) isn't as robust as L. Sani's characterisation method (by measuring t_{10}, t_{50}, t_{90}), especially with noisy input data. This is primarily due to the derivative that must be computed to find the point of inflection.

L. Sani's method is definitely the easiest to implement due to the simple interpolation formulae and due to the simplicity of finding t_{10}, t_{50}, t_{90}. It also performs quite fast and has a low memory footprint (since it doesn't require any lookup curves). Unfortunately, it does perform the worst of all methods.

References

[1] A. Balestrino, G. Marani, and L. Sani. "Aperiodic filter analysis and design by symbolic computation". In: *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*. Vol. 3. 2001, 1289–1292 vol.3. DOI: 10.1109/ICECS.2001.957451.

[2] P. Hudzovic. "Algorithm for step response parameterization". In: *Proceedings of RIP 1994 Process – Control Conference with international participation Horni Becva*. 1994, pp. 145–148.

[3] Alex Murray (TheComet93). *Report for "mathematical laboratory", FHNW Windisch*. https://github.com/thecomet93/mlab. 2016.

A Appendices

A.1 MATLAB

A.1.1 preprocess_curve()

```
function [xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw)
    % The xdata vector is not monotonically increasing with evenly spaced time
    % samples. It is very close to it though, so we can approximate it with
    % linspace
    xdata = linspace(xdata_raw(1), xdata_raw(end), length(xdata_raw))';

    % Input data is quite noisy, smooth it with a sliding average filter
    ydata = sliding_average(ydata_raw, 5);
    ydata = smooth(xdata_raw, ydata_raw, 0.2, 'loess');
end
```

A.1.2 hudzovic_curves()

```
% This function calculates the Hudzovic curves, which are later used
% for looking up time constants of a specific plant. The curves range from
% order 2 to order 8 (this is hardcoded).
%
% The return value is an array of structures, where the first element is
% for order=2, the second element is for order=3 and so on.
% Each structure containes 3 fields:
%   - r     : The "r" vector, which is an array of datapoints that belong
%             to the x axis. r will range from 0 <= r < 1/(order-1)
%   - tu_tg : The result of Tu/Tg for a specific value of r.
%   - t_tg  : The result of t/Tg for a specific value of r.
%
% Resolution specifies how finely grained the r vector should be.
function curves = hudzovic_curves(resolution)
    if nargin < 1
        resolution = 50;
    end

    % Check if we can load the curves
    if exist('hudzovic_curves.mat', 'file') == 2
        s = load('hudzovic_curves.mat');
        curves = s.curves;
        if length(curves(1).r) == resolution
            return;
        end
    end

    fprintf('Hudzovic curves need to be generated (only needs to be done once).\n');
    fprintf('This may take a while. Go get a coffee or something.\n');
    curves = hudzovic_gen_curves(resolution);
    save('hudzovic_curves.mat', 'curves');
end

function curves = hudzovic_gen_curves(resolution)
    curves = struct('r', 0, 'tu_tg', 0, 't_tg', 0, 'lambda', 0, 't_t50', 0);

    for order = 2:8
        fprintf('Generating hudzovic curve, order %d/8\n', order);

        % 0 <= r < 1/(order-1)
        r = linspace(0, 1/(order-1), resolution+1);
        r = r(1:end-1);

        % Reserve space in curves object
        curves(order-1).r = r;
        curves(order-1).tu_tg = zeros(1, resolution);
        curves(order-1).t_tg = zeros(1, resolution);

        for r_index = 1:resolution
            % Set T=1 for calculating Tk, construct transfer function H(s)
            H = hudzovic_transfer_function(1, r(r_index), order);

            % Get Tu/Tg from step response of resulting transfer function
            [h, t] = step(H);
            [Tu, Tg] = characterise_curve(t, h);
            [t10, t50, t90] = characterise_curve(t, h, [0, 1]);

            % Now we can calculate Tu/Tg as well as T/Tg with T=1 to yield
            % the two plots seen in the Hudzovic method
            curves(order-1).tu_tg(r_index) = Tu/Tg;
```

```
                curves(order-1).t_tg(r_index) = 1/Tg;
                curves(order-1).lambda(r_index) = (t90-t10)/t50;
                curves(order-1).t_t50(r_index) = 1/t50;
            end
        end
        fprintf('Done.\n');
    end
```

A.1.3 hudzovic_fit()

```
function [Tfit, rfit] = hudzovic_fit(T, r, order, xdata, ydata)
    x(1) = T;
    x(2) = r;
    function ydata = fun(x, xdata)
        H = hudzovic_transfer_function(x(1), x(2), order);
        ydata = step(H, xdata);
    end
    x = lsqcurvefit(@fun, x, linspace(xdata(1), xdata(end), length(xdata)), ydata);
    Tfit = x(1);
    rfit = x(2);
end
```

A.1.4 hudzovic_lookup()

```
function [T, r, order] = hudzovic_lookup(a, b, c, xdata, ydata)
    if nargin == 2
        [T, r, order] = hudzovic_lookup_tu_tg(a, b);
    elseif nargin == 3
        [T, r, order] = hudzovic_lookup_t10_t50_t90(a, b, c);
    else
        error('Invalid input arguments');
    end

    if r < 0 || r >= 1/(order-1)
        warning('hudzovic lookup failed, parameters are too extreme. Falling back to
            default values.');
        a, b
        if nargin == 3
            c
        end
        T, r
        r = 1/(order-1)/2;
        T = 1;
    end
end

function [T, r, order] = hudzovic_lookup_tu_tg(Tu, Tg)
    curves = hudzovic_curves();

    % First, determine required order. We check Tu/Tg against the tu_tg
    % hudzovic curve for this
    tu_tg = Tu/Tg;
    for order = 2:8
        if tu_tg <= curves(order-1).tu_tg(1)
            break
        end
    end
    fprintf('Hudzovic Tu/Tg, order %d\n', order);

    % Next, look up r in tu_tg table. Use cubic interpolation for higher
    % accuracy.
    r = spline(curves(order-1).tu_tg, curves(order-1).r, tu_tg);

    % With r, look up T in T/Tg table. Use cubic interpolation for higher
    % accuracy.
    T = spline(curves(order-1).r, curves(order-1).t_tg, r) * Tg;
end

function [T, r, order] = hudzovic_lookup_t10_t50_t90(t10, t50, t90)
    curves = hudzovic_curves();

    % First, determine required order. We check lambda against the
    % hudzovic curve for this
    lambda = (t90-t10)/t50;
    order = hudzovic_determine_order(lambda);

    % Next, look up r in lambda table. Use cubic interpolation for higher
    % accuracy.
    r = spline(curves(order-1).lambda, curves(order-1).r, lambda);

```

```
    % With r, look up T in t/t50 table. Use cubic interpolation for higher
    % accuracy.
    T = t50 * spline(curves(order-1).r, curves(order-1).t_t50, r);
end

function order = hudzovic_determine_order(lambda)
    curves = hudzovic_curves();
    for order = 2:8
        if lambda >= curves(order-1).lambda(1)
            break
        end
    end
    fprintf('Hudzovic t10/t50/t90, order %d\n', order);
end
```

A.1.5 hudzovic_transfer_function()

```
function [G, Tk] = hudzovic_transfer_function(T, r, order)
    s = tf('s');
    G = 1;
    for k = 1:order
        Tk(k) = T / (1-(k-1)*r);
        G = G / (1 + s*Tk(k));
    end
end
```

A.1.6 import_curve_from_image()

```
function [x, y, image] = import_curve_from_image(filename, decimation_factor, color_key,
    hue_threshold)
    image = imread(filename);

    if nargin < 4
        hue_threshold = 0.1; % good starting value I think
    end
    if nargin < 3
        color_key = auto_detect_color_key(image, 10);
    end
    if nargin < 2
        decimation_factor = 1;
    end

    % colors are in rgb, convert to hsv
    color_key = single_rgb2hsv(color_key);
    image_data = rgb2hsv(image);

    hue = image_data(:,:,1);
    hue_key = color_key(1);
    [y, x] = find(hue > hue_key - hue_threshold & hue < hue_key + hue_threshold);

    % decimate vectors, you don't need that many points for what we're doing
    x = x(1:decimation_factor:end);
    y = y(1:decimation_factor:end);

    % Image coordinates start in the top left rather than in the bottom
    % left, so invert y axis
    y = -y;

    % The data should be normalised to [0 .. 1] on both axes so further
    % computations are easier. We assume there is some amount of noise
    % present in the data, and we assume that the the function is more or
    % less flat in the beginning and end.
    x = x - x(1);
    x = x / x(end);
    % average start and end to be closer to the "true" start and end
    % values.
    ymin = mean(y(1:10));
    ymax = mean(y(length(y)-10:end));
    y = y - ymin;
    y = y / (ymax - ymin);
end

function color_key = auto_detect_color_key(image_data, threshold)
    % The assumption is that the grid and background will be some kind of
    % grey (r == g == b). If we find some number of pixels that don't
    % satisfy this requirement and at the same time have a similar color
    % to one another, we can assume that this is the correct color key
    color_key = [0, 0, 0];
    r = image_data(:,:,1);
    g = image_data(:,:,2);
```

```
    b = image_data(:,:,3);
    for i = 1:numel(r)
        if r(i) ~= g(i) || r(i) ~= b(i) || g(i) ~= b(i)
            color_key = [r(i), g(i), b(i)];
            return;
        end
    end
    fprintf('Warning: auto-detection of color key failed.\n');
end

function hsv = single_rgb2hsv(rgb)
    c(1, 1, 1) = rgb(1);
    c(1, 1, 2) = rgb(2);
    c(1, 1, 3) = rgb(3);
    c = rgb2hsv(c);
    hsv = [c(:,:,1), c(:,:,2), c(:,:,3)];
end
```

A.1.7 preprocess_curve()

```
function [xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw)
    % The xdata vector is not monotonically increasing with evenly spaced time
    % samples. It is very close to it though, so we can approximate it with
    % linspace
    xdata = linspace(xdata_raw(1), xdata_raw(end), length(xdata_raw))';

    % Input data is quite noisy, smooth it with a sliding average filter
    ydata = sliding_average(ydata_raw, 5);
    ydata = smooth(xdata_raw, ydata_raw, 0.2, 'loess');
end
```

A.1.8 sani_curves()

```
% This function calculates the Sani curves, which are later used
% for looking up time constants of a specific plant. The curves range from
% order 2 to order 8 (this is hardcoded).
%
% The return value is an array of structures, where the first element is
% for order=2, the second element is for order=3 and so on.
% Each structure containes 3 fields:
%   - r      : The "r" vector, which is an array of datapoints that belong
%              to the x axis. r will range from 0 <= r < 1/(order-1)
%   - tu_tg  : The result of Tu/Tg for a specific value of r.
%   - t_tg   : The result of t/Tg for a specific value of r.
%
% Resolution specifies how finely grained the r vector should be.
function curves = sani_curves(resolution)
    if nargin < 1
        resolution = 50;
    end

    % Check if we can load the curves
    if exist('sani_curves.mat', 'file') == 2
        s = load('sani_curves.mat');
        curves = s.curves;
        if length(curves(1).r) == resolution
            return;
        end
    end

    fprintf('Sani curves need to be generated (only needs to be done once).\n');
    fprintf('This may take a while. Go get a coffee or something.\n');
    curves = sani_gen_curves(resolution);
    save('sani_curves.mat', 'curves');
end

function curves = sani_gen_curves(resolution)
    curves = struct('r', 0, 'tu_tg', 0, 't_tg', 0, 'lambda', 0, 't_t50', 0);

    for order = 2:8
        fprintf('Generating sani curve, order %d/8\n', order);

        r = linspace(0, 1, resolution+2);
        r = r(2:end-1);

        % Reserve space in curves object
        curves(order-1).r = r;
        curves(order-1).tu_tg = zeros(1, resolution);
        curves(order-1).t_tg = zeros(1, resolution);
        curves(order-1).lambda = zeros(1, resolution);
```

```


        for r_index = 1:resolution
            % Set T=1 for calculating Tk, construct transfer function H(s)
            H = sani_transfer_function(1, r(r_index), order);

            % Get Tu/Tg from step response of resulting transfer function
            [h, t] = step(H);
            [Tu, Tg] = characterise_curve(t, h);
            [t10, t50, t90] = characterise_curve(t, h, [0 1]);

            % Now we can calculate Tu/Tg as well as T/Tg with T=1 to yield
            % the two plots seen in the Sani method
            curves(order-1).lambda = (t90-t10)/t50;
            curves(order-1).t_t50 = 1/t_50;
            curves(order-1).tu_tg(r_index) = Tu/Tg;
            curves(order-1).t_tg(r_index) = 1/Tg;
        end
    end
    fprintf('Done.\n');
end
```

A.1.9 sani_fit()

```
function [Tfit, rfit] = sani_fit(T, r, order, xdata, ydata)
    x(1) = T;
    x(2) = r;
    function ydata = fun(x, xdata)
        H = sani_transfer_function(x(1), x(2), order);
        ydata = step(H, xdata);
    end
    x = lsqcurvefit(@fun, x, linspace(xdata(1), xdata(end), length(xdata)), ydata);
    Tfit = x(1);
    rfit = x(2);
end
```

A.1.10 sani_lookup()

```
function [T, r, order] = sani_lookup(a, b, c)
    if nargin == 2
        [T, r, order] = sani_lookup_tu_tg(a, b);
    elseif nargin == 3
        [T, r, order] = sani_lookup_t10_t50_t90(a, b, c);
    else
        error('Invalid input arguments');
    end

    % sanity check
    if r < 0 || r > 1
        warning('sani lookup failed, parameters are too extreme. Falling back to default
            values.');
        a, b
        if nargin == 3
            c
        end
        T, r
        r = 0.5;
        T = 1;
    end
end

function [T, r, order] = sani_lookup_tu_tg(Tu, Tg)
    curves = sani_curves();

    % First, determine required order. We check Tu/Tg against the tu_tg
    % sani curve for this
    tu_tg = Tu/Tg;
    for order = 2:8
        if tu_tg <= curves(order-1).tu_tg(end)
            break
        end
    end
    fprintf('Sani Tu/Tg, order %d\n', order);

    % Next, look up r in tu_tg table. Use cubic interpolation for higher
    % accuracy.
    r = spline(curves(order-1).tu_tg, curves(order-1).r, tu_tg);

    % With r, look up T in T/Tg table. Use cubic interpolation for higher
    % accuracy.
```

```
    T = spline(curves(order-1).r, curves(order-1).t_tg, r) * Tg;
end

function [T, r, order] = sani_lookup_t10_t50_t90(t10, t50, t90)
    % Calculate lambda and determine the required filter order by doing a
    % quick lookup on all orders.
    lambda = (t90 - t10) / t50;
    order = sani_determine_order(lambda);

    % Next, do a binary search on the lambda function for the chosen order
    % to find r.
    fun = @(r)sani_lambda(r, order);
    r = binary_search(fun, lambda, 0, 1);

    % With r, calculate T using the t50 formula.
    T = t50 / (log(2) - 1 + (1-r^order)/(1-r));
end

function order = sani_determine_order(lambda)
    for order = 2:8
        if lambda >= sani_lambda(1-1e-6, order);
            break;
        end
    end
    fprintf('Sani t10/t50/t90, order %d\n', order);
end

function lambda = sani_lambda(r, order)
    lambda = (1.315*sqrt(3.8 * (1-r^(2*order))/(1-r^2) - 1)) / (log(2) - 1 + (1-r^order)/(1-r));
end

function result = binary_search(fun, target, lower, upper)
    mid = (upper - lower) / 2;
    x = mid / 2;
    max_iter = 20;
    while max_iter > 0
        max_iter = max_iter - 1;
        y = fun(x);
        mid = mid / 2;
        if y > target
            x = x + mid;
        else
            x = x - mid;
        end
    end

    result = x;
end
```

A.1.11 sani_transfer_function()

```
function G = sani_transfer_function(T, r, order)
    s = tf('s');
    G = 1;
    for k = 0:order-1
        G = G / (1+s*T*r^k);
    end
end
```

A.1.12 sliding_average()

```
% Moves a sliding average filter over the input signal. The output vector
% will be the same size as the input signal, however, this comes at the
% cost of the first "num" elements not fully being averaged.
function y = sliding_average(ydata, num)
    y = ydata(:);
    half = floor(num/2);
    for i = 1:length(ydata)
        y(i) = mean(ydata(max(1, i-half):min(end, i+half)));
    end
end
```

A.2 MATLAB simulations

A.2.1 error_calculations()

```matlab
function error_calculations()
    close all;

    % subroutines are located in this folder
    addpath([pwd,'/mfunctions']);

    %error_calculations_noise();
    %error_calculations_order();

    display_error_vs_order();
    display_error_vs_noise();
    display_error_vs_noise_avg();
    %display_failure_rate();
    display_sani_lookup_vs_interpolation();
end

function display_error_vs_order()
    load('errors_order.mat', 'errors_order');

    % Error vs Order
    for k = 2:8
        tmp(k-1, 1) = mean(errors_order(k-1).hudzovic_tu_tg(isfinite(errors_order(k-1).hudzovic_tu_tg)));
        tmp(k-1, 2) = mean(errors_order(k-1).hudzovic_t10_t50_t90(isfinite(errors_order(k-1).hudzovic_t10_t50_t90)));
        tmp(k-1, 3) = mean(errors_order(k-1).sani_tu_tg(isfinite(errors_order(k-1).sani_tu_tg)));
        tmp(k-1, 4) = mean(errors_order(k-1).sani_t10_t50_t90(isfinite(errors_order(k-1).sani_t10_t50_t90)));
        tmp(k-1, 5) = mean(errors_order(k-1).hudzovic_fit(isfinite(errors_order(k-1).hudzovic_fit)));
        tmp(k-1, 6) = mean(errors_order(k-1).sani_fit(isfinite(errors_order(k-1).sani_fit)));
    end

    figure;
    semilogy(2:8, tmp, 'LineWidth', 2);
    grid on
    legend('\fontsize{14}Hudzovic Tu/Tg',...
        '\fontsize{14}Hudzovic t10/t50/t90',...
        '\fontsize{14}Sani Tu/Tg',...
        '\fontsize{14}Sani t10/t50/t90',...
        '\fontsize{14}Hudzovic fit',...
        '\fontsize{14}Sani fit');
    %axis([2 8 10e-7 20e-3]);
    axis square
    xlabel('\fontsize{14}Order of filter');
    ylabel('\fontsize{14}Root Mean Square Error');
    title('\fontsize{16}Error vs Order');
end

function display_sani_lookup_vs_interpolation()
    tmp = load('errors_order_sani_lookup.mat', 'errors_order');
    lookup = tmp.errors_order;
    tmp = load('errors_order_sani_interpolation.mat', 'errors_order');
    interpolation = tmp.errors_order;

    % Error vs Order
    tmp = zeros(7, 6);
    for k = 2:8
        tmp(k-1, 1) = mean(interpolation(k-1).hudzovic_tu_tg(isfinite(interpolation(k-1).hudzovic_tu_tg)));
        tmp(k-1, 2) = mean(interpolation(k-1).hudzovic_t10_t50_t90(isfinite(interpolation(k-1).hudzovic_t10_t50_t90)));
        tmp(k-1, 3) = mean(interpolation(k-1).sani_tu_tg(isfinite(interpolation(k-1).sani_tu_tg)));
        tmp(k-1, 4) = mean(interpolation(k-1).sani_t10_t50_t90(isfinite(interpolation(k-1).sani_t10_t50_t90)));
        tmp(k-1, 5) = mean(interpolation(k-1).hudzovic_fit(isfinite(interpolation(k-1).hudzovic_fit)));
        tmp(k-1, 6) = mean(interpolation(k-1).sani_fit(isfinite(interpolation(k-1).sani_fit)));
    end

    figure;
    for i = 1:6
        if i == 4
            semilogy(2:8, tmp(:,i), 'LineWidth', 2);
        else
            semilogy(2:8, tmp(:,i));  hold on, grid on
        end
    end
    tmp2 = zeros(1, 7);
```

```
    for k = 2:8
        tmp2(k-1) = mean(lookup(k-1).sani_t10_t50_t90(isfinite(lookup(k-1).sani_t10_t50_t90)));
    end
    semilogy(2:8, tmp2, '--', 'LineWidth', 2);

    legend('\fontsize{14}Hudzovic Tu/Tg',...
        '\fontsize{14}Hudzovic t10/t50/t90',...
        '\fontsize{14}Sani Tu/Tg',...
        '\fontsize{14}Sani t10/t50/t90, Interpolation Formulae',...
        '\fontsize{14}Hudzovic fit',...
        '\fontsize{14}Sani fit',...
        '\fontsize{14}Sani t10/t50/t90, Lookup');
    %axis([2 8 10e-7 20e-3]);
    axis square
    xlabel('\fontsize{14}Order of filter');
    ylabel('\fontsize{14}Root Mean Square Error');
    title('\fontsize{16}Sani Interpolation Formulae vs Lookup');
end

function display_error_vs_noise()
    load('errors_noise_fit.mat', 'errors_noise');

    % Error vs Input noise
    tmp = zeros(length(errors_noise.hudzovic_tu_tg), 6);
    tmp(:,1) = errors_noise.hudzovic_tu_tg;
    tmp(:,2) = errors_noise.hudzovic_t10_t50_t90;
    tmp(:,3) = errors_noise.sani_tu_tg;
    tmp(:,4) = errors_noise.sani_t10_t50_t90;
    tmp(:,5) = errors_noise.hudzovic_fit;
    tmp(:,6) = errors_noise.sani_fit;
    for i = 1:6
        tmp(:,i) = sliding_average(tmp(:,i), 10);
    end
    figure;
    semilogy(errors_noise.noise_amplitude * 100, tmp);
    grid on
    legend('\fontsize{14}Hudzovic Tu/Tg',...
        '\fontsize{14}Hudzovic t10/t50/t90',...
        '\fontsize{14}Sani Tu/Tg',...
        '\fontsize{14}Sani t10/t50/t90',...
        '\fontsize{14}Hudzovic fit',...
        '\fontsize{14}Sani fit',...
        'Location', 'southeast');
    %axis([0 200 10e-8 20e-2]);
    axis square
    xlabel('\fontsize{14}Normalised noise amplitude (%)');
    ylabel('\fontsize{14}Root Mean Square Error');
    title({'\fontsize{16}Error vs Input noise', '\fontsize{14}(Single 4th Order Step Response)'});
end

function display_error_vs_noise_avg()
    load('errors_noise_avg.mat', 'errors_noise');

    % Error vs Input noise
    tmp = zeros(length(errors_noise.hudzovic_tu_tg), 6);
    tmp(:,1) = errors_noise.hudzovic_tu_tg;
    tmp(:,2) = errors_noise.hudzovic_t10_t50_t90;
    tmp(:,3) = errors_noise.sani_tu_tg;
    tmp(:,4) = errors_noise.sani_t10_t50_t90;
    for i = 1:4
        tmp(:,i) = sliding_average(tmp(:,i), 20);
    end
    figure;
    semilogy(errors_noise.noise_amplitude * 100, tmp);
    grid on
    legend('\fontsize{14}Hudzovic Tu/Tg',...
        '\fontsize{14}Hudzovic t10/t50/t90',...
        '\fontsize{14}Sani Tu/Tg',...
        '\fontsize{14}Sani t10/t50/t90',...
        'Location', 'southeast');
    %axis([0 200 10e-8 20e-2]);
    axis square
    xlabel('\fontsize{14}Normalised noise amplitude (%)');
    ylabel('\fontsize{14}Root Mean Square Error');
    title({'\fontsize{16}Error vs Input noise', '\fontsize{14}(Random Step Responses)'});
end

function display_failure_rate()
    load('errors_order.mat', 'errors_order');

    % Plot the "failure rate" in function of order. Whenever the error is
```

```
    % too large, the error is set to Inf. For every order, we created 100
    % random step responses which means the number of Inf items is directly
    % the failure rate in percent.
    tmp = zeros(7, 6);
    for k = 2:8
        tmp(k-1, 1) = 100 - sum(isfinite(errors_order(k-1).hudzovic_tu_tg));
        tmp(k-1, 2) = 100 - sum(isfinite(errors_order(k-1).hudzovic_t10_t50_t90));
        tmp(k-1, 3) = 100 - sum(isfinite(errors_order(k-1).sani_tu_tg));
        tmp(k-1, 4) = 100 - sum(isfinite(errors_order(k-1).sani_t10_t50_t90));
        tmp(k-1, 5) = 100 - sum(isfinite(errors_order(k-1).hudzovic_fit));
        tmp(k-1, 6) = 100 - sum(isfinite(errors_order(k-1).sani_fit));
    end
    figure;
    plot(2:8, tmp, 'LineWidth', 2);
    grid on
    legend('\fontsize{14}Hudzovic Tu/Tg',...
        '\fontsize{14}Hudzovic t10/t50/t90',...
        '\fontsize{14}Sani Tu/Tg',...
        '\fontsize{14}Sani t10/t50/t90',...
        '\fontsize{14}Hudzovic fit',...
        '\fontsize{14}Sani fit');
    axis square
    xlabel('\fontsize{14}Order of filter');
    ylabel('\fontsize{14}Failure Rate (%)');
    title('\fontsize{16}Failure vs Order');
end

function error_calculations_noise()
    rand('state', 0);
    %errors_noise = struct
    load('errors_noise.mat', 'errors_noise');

    num_simulations = 1000;
    num_avg = 500;
    for i = 851:num_simulations
        fprintf('Current iteration: %d\n', i);
        for k = 1:num_avg
            % generate transfer function
            [xdata_orig, ydata_orig] = gen_random_ptn(randi([2 8], 1, 1));

            % apply noise
            amp_rand = 0.35 * (i-1) / (num_simulations-1);
            xdata_raw = xdata_orig;
            ydata_raw = ydata_orig + amp_rand * (rand(length(ydata_orig),1)-0.5);

            err_acc = zeros(1, 6);

            try
                [xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw);
                [Tu, Tg] = characterise_curve(xdata, ydata);
                [t10, t50, t90] = characterise_curve(xdata, ydata, [0 1]); % We know it's
                    normalised to 0-1

                % Hudzovic, Tu/Tg
                [T, r, order] = hudzovic_lookup(Tu, Tg);
                G = hudzovic_transfer_function(T, r, order);
                g_hudzovic_tu_tg = step(G, xdata);

                % Hudzovic, t10/t50/t90
                [T, r, order] = hudzovic_lookup(t10, t50, t90);
                G = hudzovic_transfer_function(T, r, order);
                g_hudzovic_t3 = step(G, xdata);

                % Sani, Tu/Tg
                [T, r, order] = sani_lookup(Tu, Tg);
                G = sani_transfer_function(T, r, order);
                g_sani_tu_tg = step(G, xdata);

                % Sani, t10/t50/t90
                [T, r, order] = sani_lookup(t10, t50, t90);
                G = sani_transfer_function(T, r, order);
                g_sani_t3 = step(G, xdata);

                % Hudzovic fit of raw data
%                 [T, r, order] = hudzovic_lookup(t10, t50, t90);
%                 [T, r] = hudzovic_fit(T, r, order, xdata_raw, ydata_raw);
%                 G = hudzovic_transfer_function(T, r, order);
%                 g_hudzovic_fit = step(G, xdata);
%
%                 % Sani fit of raw data
%                 [T, r, order] = sani_lookup(t10, t50, t90);
%                 [T, r] = sani_fit(T, r, order, xdata_raw, ydata_raw);
%                 G = sani_transfer_function(T, r, order);
```

```
%                 g_sani_fit = step(G, xdata);
            catch
            end

            % accumulate errors so we can compute the average later
            err_acc(1) = err_acc(1) + sqrt(immse(g_hudzovic_tu_tg, ydata_orig));
            err_acc(2) = err_acc(2) + sqrt(immse(g_hudzovic_t3, ydata_orig));
            err_acc(3) = err_acc(3) + sqrt(immse(g_sani_tu_tg, ydata_orig));
            err_acc(4) = err_acc(4) + sqrt(immse(g_sani_t3, ydata_orig));
%             err_acc(5) = err_acc(5) + sqrt(immse(g_hudzovic_fit, ydata_orig));
%             err_acc(6) = err_acc(6) + sqrt(immse(g_sani_fit, ydata_orig));
        end

        % average
        err_acc = err_acc ./ num_avg;

        errors_noise.noise_amplitude(i) = amp_rand;
        errors_noise.hudzovic_tu_tg(i) = err_acc(1);
        errors_noise.hudzovic_t10_t50_t90(i) = err_acc(2);
        errors_noise.sani_tu_tg(i) = err_acc(3);
        errors_noise.sani_t10_t50_t90(i) = err_acc(4);
%         errors_noise.hudzovic_fit(i) = err_acc(5);
%         errors_noise.sani_fit(i) = err_acc(6);
    end

    save('errors_noise.mat', 'errors_noise');
end

function error_calculations_order()
    rand('state', 0);

    for k = 2:8
        num_simulations = 100;
        errors_order(k-1) = struct(...
            'hudzovic_tu_tg', 0,...
            'hudzovic_t10_t50_t90', 0,...
            'sani_tu_tg', 0,...
            'sani_t10_t50_t90', 0,...
            'hudzovic_fit', 0,...
            'sani_fit', 0);

        for i = 1:num_simulations
            [xdata, ydata] = gen_random_ptn(k);

            [Tu, Tg] = characterise_curve(xdata, ydata);
            [t10, t50, t90] = characterise_curve(xdata, ydata, [0 1]); % We know it's normalised to 0-1

            try
            % Hudzovic, Tu/Tg
            [T, r, order] = hudzovic_lookup(Tu, Tg);
            G = hudzovic_transfer_function(T, r, order);
            g_hudzovic_tu_tg = step(G, xdata);

            % Hudzovic, t10/t50/t90
            [T, r, order] = hudzovic_lookup(t10, t50, t90);
            G = hudzovic_transfer_function(T, r, order);
            g_hudzovic_t3 = step(G, xdata);

            % Sani, Tu/Tg
            [T, r, order] = sani_lookup(Tu, Tg);
            G = sani_transfer_function(T, r, order);
            g_sani_tu_tg = step(G, xdata);

            % Sani, t10/t50/t90
            [T, r, order] = sani_lookup(t10, t50, t90);
            G = sani_transfer_function(T, r, order);
            g_sani_t3 = step(G, xdata);

            % Hudzovic fit of raw data
            [T, r, order] = hudzovic_lookup(t10, t50, t90);
            [T, r] = hudzovic_fit(T, r, order, xdata, ydata);
            G = hudzovic_transfer_function(T, r, order);
            g_hudzovic_fit = step(G, xdata);

            % Sani fit of raw data
            [T, r, order] = sani_lookup(t10, t50, t90);
            [T, r] = sani_fit(T, r, order, xdata, ydata);
            G = sani_transfer_function(T, r, order);
            g_sani_fit = step(G, xdata);
            catch
            end

```

```
                errors_order(k-1).hudzovic_tu_tg(i) = sqrt(immse(g_hudzovic_tu_tg, ydata));
                errors_order(k-1).hudzovic_t10_t50_t90(i) = sqrt(immse(g_hudzovic_t3, ydata));
                errors_order(k-1).sani_tu_tg(i) = sqrt(immse(g_sani_tu_tg, ydata));
                errors_order(k-1).sani_t10_t50_t90(i) = sqrt(immse(g_sani_t3, ydata));
                errors_order(k-1).hudzovic_fit(i) = sqrt(immse(g_hudzovic_fit, ydata));
                errors_order(k-1).sani_fit(i) = sqrt(immse(g_sani_fit, ydata));
            end
        end

        save('errors_order.mat', 'errors_order');
end

function error_calculations_order_sani_lookup()
    rand('state', 0);

    for k = 2:8
        num_simulations = 100;

        for i = 1:num_simulations
            [xdata, ydata] = gen_random_ptn(k);

            [Tu, Tg] = characterise_curve(xdata, ydata);
            [t10, t50, t90] = characterise_curve(xdata, ydata, [0 1]); % We know it's normalised to 0-1

            try
            % Sani, t10/t50/t90
            [T, r, order] = sani_lookup(t10, t50, t90);
            G = sani_transfer_function(T, r, order);
            g_sani_t3 = step(G, xdata);
            catch
            end

            errors_order(k-1) = rmse(g_sani_t3, ydata);
        end
    end

    save('errors_order_sani_lambda_lookup.mat', 'errors_order');
end

function [xdata, ydata] = gen_random_ptn(order)
    s = tf('s');
    G = 1;
    for k = 1:order
        Tk = (rand(1,1)*0.8+0.1) * 10;
        G = G / (1+s*Tk);
    end
    [ydata, xdata] = step(G);
    ydata = ydata - ydata(1);
    ydata = ydata / ydata(end);
end
```

A.2.2 plot_hudzovic_curves()

```
close all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

curves = hudzovic_curves();

figure;
subplot(211); grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).tu_tg, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}Tu / Tg');
title('\fontsize{16}Hudzovic Lookup, Tu / Tg');
axis square
legend('n=2', 'n=3', 'n=4', 'n=5', 'n=6', 'n=7', 'n=8');

subplot(212), grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).t_tg, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}1 / Tg');
axis square

% draw in an example lookup
subplot(211);
```

```
plot([0, 0.4], [0.1653, 0.1653], 'k--');
plot([0.4, 0.4], [0.1653, 0], 'k--');
subplot(212);
plot([0.4, 0.4], [0.4, 0.10735], 'k--');
plot([0.4, 0], [0.10735, 0.10735], 'k--');

figure;
subplot(211); grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).lambda, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}lambda (\lambda)');
title('\fontsize{16}Hudzovic Lookup, t10 / t50 / t90');
axis([0 1 0 3]);
axis square
legend('n=2', 'n=3', 'n=4', 'n=5', 'n=6', 'n=7', 'n=8', 'Location', 'southeast');

subplot(212); grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).t_t50, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}1 / t50');
axis square

% draw in an example lookup
subplot(211);
plot([0, 0.3479], [1.767, 1.767], 'k--');
plot([0.3479, 0.3479], [1.767, 0], 'k--');
subplot(212);
plot([0.3479, 0.3479], [0.6, 0.2007], 'k--');
plot([0.3479, 0], [0.2007, 0.2007], 'k--');
```

A.2.3 plot_hudzovic_tu_tg()

```
close all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

s = tf('s');
G = 1.4 + 5/(1+s)^2/(1+0.5*s)/(1+0.4*s);
[y, t] = step(G);

[Tu, Tg] = characterise_curve(t, y);
ymin = min(y);
ymax = max(y);
int_top = {Tu + Tg, 6.4};
int_bottom = {Tu, 1.4};
int_dir = [Tu + Tg, 6.4] - [Tu, 1.4];
int_dir = int_dir/norm(int_dir);

plot(t, y, 'LineWidth', 2); hold on, grid on
plot(int_top{:},'r.','MarkerSize',50);
plot(int_bottom{:},'r.','MarkerSize',50);
plot([0, 10], [1.4, 1.4], 'k--', 'LineWidth', 2);
plot([0, 10], [6.4, 6.4], 'k--', 'LineWidth', 2);
x = [int_bottom{1} - int_dir(1), int_top{1} + int_dir(1)];
y = [int_bottom{2} - int_dir(2), int_top{2} + int_dir(2)];
plot(x, y, 'k--', 'LineWidth', 2);
axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{16}Method of Hudzovic: Tu / Tg');
```

A.2.4 plot_sani_curves()

```
close all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

curves = sani_curves();

figure;
subplot(211); grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).tu_tg, 'LineWidth', 2);
end
```

```
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}Tu / Tg');
title('\fontsize{16}Sani Lookup, Tu / Tg');
axis square
legend('n=2', 'n=3', 'n=4', 'n=5', 'n=6', 'n=7', 'n=8', 'Location', 'northwest');
subplot(212), grid on, hold on, grid minor
for k = 1:7
    plot(curves(k).r, curves(k).t_tg, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}1 / Tg');
axis square

% draw in an example lookup
subplot(211);
plot([0, 0.9], [0.3169, 0.3169], 'k--');
plot([0.9, 0.9], [0.3169, 0], 'k--');
subplot(212);
plot([0.9, 0.9], [1, 0.26], 'k--');
plot([0.9, 0], [0.26, 0.26], 'k--');

figure;
subplot(211); hold on, grid on, grid minor
r = linspace(0, 1);
for k = 2:8
    lambda = (1.315*sqrt(3.8 * (1-r.^(2*k))./(1-r.^2) - 1)) ./ (log(2) - 1 + (1-r.^k)./(1-r));
    plot(r, lambda, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}lambda (\lambda)');
title('\fontsize{16}Sani Lookup, t10 / t50 / t90');
axis square
legend('n=2', 'n=3', 'n=4', 'n=5', 'n=6', 'n=7', 'n=8', 'Location', 'northeast');
subplot(212); hold on, grid on, grid minor
for k = 2:8
    t_t50 = 1 ./ (log(2) - 1 + (1-r.^k)./(1-r));
    plot(r, t_t50, 'LineWidth', 2);
end
xlabel('\fontsize{14}Parameter r');
ylabel('\fontsize{14}1 / t50');
axis square

% draw in an example lookup
subplot(211);
plot([0, 0.8], [1.6062, 1.6062], 'k--');
plot([0.8, 0.8], [1.6062, 0.5], 'k--');
subplot(212);
plot([0.8, 0.8], [1.5, 0.4688], 'k--');
plot([0.8, 0], [0.4688, 0.4688], 'k--');
```

A.2.5 plot_t10_t50_t90()

```
close all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

s = tf('s');
G = 1.4 + 5/(1+s)^2/(1+0.5*s)/(1+0.4*s);
[y, t] = step(G);

[t10, t50, t90] = characterise_curve(t, y);
y10 = spline(t, y, t10);
y50 = spline(t, y, t50);
y90 = spline(t, y, t90);

plot(t, y, 'LineWidth', 2); hold on, grid on
plot(t10, y10,'r.','MarkerSize',50);
plot(t50, y50,'r.','MarkerSize',50);
plot(t90, y90,'r.','MarkerSize',50);
plot([0,t10], [y10,y10], 'k--', 'LineWidth', 2);
plot([t10,t10], [y10,1], 'k--', 'LineWidth', 2);
plot([0,t50], [y50,y50], 'k--', 'LineWidth', 2);
plot([t50,t50], [y50,1], 'k--', 'LineWidth', 2);
plot([0,t90], [y90,y90], 'k--', 'LineWidth', 2);
plot([t90,t90], [y90,1], 'k--', 'LineWidth', 2);

axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{16}Method of Sani: Tu / Tg');
```

A.2.6 step_response_hudzovic_vs_fit()

```
close all, clear all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

rand('state', 0);

% Load the step response of a heater directly from an image plot. We have
% to manually specify the offset and Y scale Ks.
yoffset = 22;
xtime = 10;
Ks = 37 - yoffset;

% generate transfer function and apply some noise on top
amp_rand = 0.2;
G = sani_transfer_function(1, 0.5, 4);
[ydata_orig, xdata_raw] = step(G);
ydata_orig = ydata_orig - ydata_orig(1);
ydata_orig = ydata_orig / ydata_orig(end);
ydata_raw = ydata_orig + amp_rand * (rand(length(ydata_orig),1)-0.5);

[xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw);
[Tu, Tg] = characterise_curve(xdata, ydata);
[T, r, n] = hudzovic_lookup(Tu, Tg);
g_normal = step(hudzovic_transfer_function(T, r, n), xdata);
[T, r] = hudzovic_fit(T, r, n, xdata_raw, ydata_raw);
g_fit = step(hudzovic_transfer_function(T, r, n), xdata);

err_normal = (g_normal - ydata_orig).^2;
err_fit = (g_fit - ydata_orig).^2;

figure;
subplot(211); hold on, grid on, grid minor
scatter(xdata_raw, ydata_raw);
plot(xdata_raw, ydata_orig, 'k--');
plot(xdata, g_normal, 'LineWidth', 2);
plot(xdata, g_fit, 'LineWidth', 2);

axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{15}Hudzovic vs Fit, Noise=20%');

yyaxis right
plot(xdata, err_normal);
plot(xdata, err_fit);
ylabel('\fontsize{14}Squared Error');

legend('Input Data', 'Correct Response', 'Hudzovic Tu / Tg', 'Hudzovic Fit', 'Error of
    Hudzovic Tu / Tg', 'Error of Hudzovic Fit', 'Location', 'east');

[xdata, ydata] = preprocess_curve(xdata, ydata_orig);
[Tu, Tg] = characterise_curve(xdata, ydata);
[T, r, n] = hudzovic_lookup(Tu, Tg);
g_normal = step(hudzovic_transfer_function(T, r, n), xdata);
[T, r] = hudzovic_fit(T, r, n, xdata_raw, ydata_orig);
g_fit = step(hudzovic_transfer_function(T, r, n), xdata);

err_normal = (g_normal - ydata_orig).^2;
err_fit = (g_fit - ydata_orig).^2;


subplot(212); hold on, grid on, grid minor
scatter(xdata, ydata_orig);
plot(xdata_raw, ydata_orig, 'k--');
plot(xdata, g_normal, 'LineWidth', 2);
plot(xdata, g_fit, 'LineWidth', 2);

axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{15}Hudzovic vs Fit, Noise=0%');

yyaxis right
plot(xdata, err_normal);
plot(xdata, err_fit);
ylabel('\fontsize{14}Squared Error');

legend('Input Data', 'Correct Response', 'Hudzovic Tu / Tg', 'Hudzovic Fit', 'Error of
    Hudzovic Tu / Tg', 'Error of Hudzovic Fit', 'Location', 'east');
```

A.2.7 step_response_image()

```
close all;

% subroutines are located in this folder
addpath([pwd, '/mfunctions']);

% Load the step response of a heater directly from an image plot. We have
% to manually specify the offset and Y scale Ks.
yoffset = 22;
Ks = 37 - yoffset;
decimation_factor = 10;
[xdata_raw, ydata_raw, img] = import_curve_from_image('images/plant1.png',
    decimation_factor);

% The xdata vector is not monotonically increasing with evenly spaced time
% samples. It is very close to it though, so we can approximate it with
% linspace
xdata = linspace(xdata_raw(1), xdata_raw(end), length(xdata_raw))';

% Input data is quite noisy, smooth it with a sliding average filter
ydata = sliding_average(ydata_raw, ceil(length(ydata) * 0.08));

[Tu, Tg] = characterise_curve(xdata, ydata);
[t10, t50, t90] = characterise_curve(xdata, ydata);

% Hudzovic, Tu/Tg
[T, r, order] = hudzovic_lookup(Tu, Tg);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_tu_tg = step(G * Ks + yoffset, xdata);

% Hudzovic, t10/t50/t90
[T, r, order] = hudzovic_lookup(t10, t50, t90);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_t3 = step(G * Ks + yoffset, xdata);

% Sani, Tu/Tg
[T, r, order] = sani_lookup(Tu, Tg);
G = sani_transfer_function(T, r, order);
g_sani_tu_tg = step(G * Ks + yoffset, xdata);

% Sani, t10/t50/t90
[T, r, order] = sani_lookup(t10, t50, t90);
G = sani_transfer_function(T, r, order);
g_sani_t3 = step(G * Ks + yoffset, xdata);

% Hudzovic fit of raw data
[T, r, order] = hudzovic_lookup(Tu, Tg);
[T, r] = hudzovic_fit(T, r, order, xdata_raw, ydata_raw);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_fit = step(G * Ks + yoffset, xdata);

% Sani fit of raw data
[T, r, order] = sani_lookup(t10, t50, t90);
[T, r] = sani_fit(T, r, order, xdata_raw, ydata_raw);
G = sani_transfer_function(T, r, order);
g_sani_fit = step(G * Ks + yoffset, xdata);

figure; hold on, grid on, grid minor
scatter(xdata_raw, ydata_raw * Ks + yoffset);
scatter(xdata, ydata * Ks + yoffset);
plot(xdata, g_hudzovic_tu_tg);
plot(xdata, g_hudzovic_t3);
plot(xdata, g_sani_tu_tg);
plot(xdata, g_sani_t3);
plot(xdata, g_hudzovic_fit);
plot(xdata, g_sani_fit);
legend('Original data', 'Hudzovic Tu/Tg', 'Hudzovic t10/t50/t90', 'Sani Tu/Tg', 'Sani t10/
    t50/t90', 'Hudzovic fit', 'Sani fit');
return;

% Try fitting the time constants individually
% NOTE ydata needs to be normalised
Tk = ptn_fit(xdata, ydata, order);
s = tf('s');
H = 1;
for k = 1:length(Tk)
    H = H / (1 + s*Tk(k));
end
H = H * Ks + yoffset;
h = step(H, xdata);
plot(xdata, h);
```

```
legend('correct', 'hudzovic', 'fit');
```

A.2.8 step_response_noisy()

```
close all, clear all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

rand('state', 0);

% Load the step response of a heater directly from an image plot. We have
% to manually specify the offset and Y scale Ks.
yoffset = 22;
xtime = 10;
Ks = 37 - yoffset;

% generate transfer function and apply some noise on top
amp_rand = 0;
G = hudzovic_transfer_function(1, 1/3/2, 4);
[ydata_orig, xdata_orig] = step(G);
ydata_orig = ydata_orig - ydata_orig(1);
ydata_orig = ydata_orig / ydata_orig(end);
xdata_raw = xdata_orig;
ydata_raw = ydata_orig + amp_rand * (rand(length(ydata_orig),1)-0.5);

[xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw);
[Tu, Tg] = characterise_curve(xdata, ydata);
[t10, t50, t90] = characterise_curve(xdata, ydata);

% Hudzovic, Tu/Tg
[T, r, order] = hudzovic_lookup(Tu, Tg);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_tu_tg = step(G * Ks + yoffset, xdata);

% Hudzovic, t10/t50/t90
[T, r, order] = hudzovic_lookup(t10, t50, t90);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_t3 = step(G * Ks + yoffset, xdata);

% Sani, Tu/Tg
[T, r, order] = sani_lookup(Tu, Tg);
G = sani_transfer_function(T, r, order);
g_sani_tu_tg = step(G * Ks + yoffset, xdata);

% Sani, t10/t50/t90
[T, r, order] = sani_lookup(t10, t50, t90);
G = sani_transfer_function(T, r, order);
g_sani_t3 = step(G * Ks + yoffset, xdata);

% Hudzovic fit of raw data
[T, r, order] = hudzovic_lookup(t10, t50, t90);
[T, r] = hudzovic_fit(T, r, order, xdata_raw, ydata_raw);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_fit = step(G * Ks + yoffset, xdata);

% Sani fit of raw data
[T, r, order] = sani_lookup(t10, t50, t90);
[T, r] = sani_fit(T, r, order, xdata_raw, ydata_raw);
G = sani_transfer_function(T, r, order);
g_sani_fit = step(G * Ks + yoffset, xdata);

figure; hold on, grid on, grid minor
scatter(xdata_raw, ydata_raw * Ks + yoffset);
scatter(xdata, ydata * Ks + yoffset);
plot(xdata, g_hudzovic_tu_tg);
plot(xdata, g_hudzovic_t3);
plot(xdata, g_sani_tu_tg);
plot(xdata, g_sani_t3);
plot(xdata, g_hudzovic_fit);
plot(xdata, g_sani_fit);
legend(...
    'Original data',...
    'Smoothed data',...
    'Hudzovic Tu/Tg',...
    'Hudzovic t10/t50/t90',...
    'Sani Tu/Tg',...
    'Sani t10/t50/t90',...
    'Hudzovic fit',...
    'Sani fit');
```

A.2.9 step_response_perfect()

```
close all, clear all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

% Generate a "perfect" step response curve
yoffset = 22;
Ks = 37 - yoffset;
G = yoffset + Ks * hudzovic_transfer_function(1, 0.1, 2);
[ydata, xdata] = step(G);
% normalise Y data
ydata = ydata - ydata(1);
ydata = ydata / ydata(end);

[Tu, Tg] = characterise_curve(xdata, ydata);
[t10, t50, t90] = characterise_curve(xdata, ydata);

% Hudzovic, Tu/Tg
[T, r, order] = hudzovic_lookup(Tu, Tg);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_tu_tg = step(G * Ks + yoffset, xdata);

% Hudzovic, t10/t50/t90
[T, r, order] = hudzovic_lookup(t10, t50, t90);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_t3 = step(G * Ks + yoffset, xdata);

% Sani, Tu/Tg
[T, r, order] = sani_lookup(Tu, Tg);
G = sani_transfer_function(T, r, order);
g_sani_tu_tg = step(G * Ks + yoffset, xdata);

% Sani, t10/t50/t90
[T, r, order] = sani_lookup(t10, t50, t90);
G = sani_transfer_function(T, r, order);
g_sani_t3 = step(G * Ks + yoffset, xdata);

% Hudzovic fit of raw data
[T, r, order] = hudzovic_lookup(t10, t50, t90);
[T, r] = hudzovic_fit(T, r, order, xdata, ydata);
G = hudzovic_transfer_function(T, r, order);
g_hudzovic_fit = step(G * Ks + yoffset, xdata);

% Sani fit of raw data
[T, r, order] = sani_lookup(t10, t50, t90);
[T, r] = sani_fit(T, r, order, xdata, ydata);
G = sani_transfer_function(T, r, order);
g_sani_fit = step(G * Ks + yoffset, xdata);

figure; hold on, grid on, grid minor
scatter(xdata, yoffset + Ks * ydata);
plot(xdata, g_hudzovic_tu_tg);
plot(xdata, g_hudzovic_t3);
plot(xdata, g_sani_tu_tg);
plot(xdata, g_sani_t3);
plot(xdata, g_hudzovic_fit);
plot(xdata, g_sani_fit);
legend('Original data', 'Hudzovic Tu/Tg', 'Hudzovic t10/t50/t90', 'Sani Tu/Tg', 'Sani t10/
    t50/t90', 'Hudzovic fit', 'Sani fit');
```

A.2.10 step_response_sani_vs_fit()

```
close all, clear all;

% subroutines are located in this folder
addpath([pwd,'/mfunctions']);

rand('state', 0);

% Load the step response of a heater directly from an image plot. We have
% to manually specify the offset and Y scale Ks.
yoffset = 22;
xtime = 10;
Ks = 37 - yoffset;

% generate transfer function and apply some noise on top
amp_rand = 0.2;
G = hudzovic_transfer_function(1, 1/3/2, 4);
[ydata_orig, xdata_raw] = step(G);
ydata_orig = ydata_orig - ydata_orig(1);
```

```matlab
ydata_orig = ydata_orig / ydata_orig(end);
ydata_raw = ydata_orig + amp_rand * (rand(length(ydata_orig),1)-0.5);

[xdata, ydata] = preprocess_curve(xdata_raw, ydata_raw);
[Tu, Tg] = characterise_curve(xdata, ydata);
[T, r, n] = sani_lookup(Tu, Tg);
g_normal = step(sani_transfer_function(T, r, n), xdata);
[T, r] = sani_fit(T, r, n, xdata_raw, ydata_raw);
g_fit = step(sani_transfer_function(T, r, n), xdata);

err_normal = (g_normal - ydata_orig).^2;
err_fit = (g_fit - ydata_orig).^2;

figure;
subplot(211); hold on, grid on, grid minor
scatter(xdata_raw, ydata_raw);
plot(xdata_raw, ydata_orig, 'k--');
plot(xdata, g_normal, 'LineWidth', 2);
plot(xdata, g_fit, 'LineWidth', 2);

axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{16}Sani vs  Fit, Noise=20%');

yyaxis right
plot(xdata, err_normal);
plot(xdata, err_fit);
ylabel('\fontsize{14}Squared Error');

legend('Input Data', 'Correct Response', 'Sani Tu / Tg', 'Sani Fit', 'Error of Sani Tu / Tg', 'Error of Sani Fit', 'Location', 'east');


[xdata, ydata] = preprocess_curve(xdata_raw, ydata_orig);
[Tu, Tg] = characterise_curve(xdata, ydata);
[T, r, n] = sani_lookup(Tu, Tg);
g_normal = step(sani_transfer_function(T, r, n), xdata);
[T, r] = sani_fit(T, r, n, xdata_raw, ydata_orig);
g_fit = step(sani_transfer_function(T, r, n), xdata);

err_normal = (g_normal - ydata_orig).^2;
err_fit = (g_fit - ydata_orig).^2;

subplot(212); hold on, grid on, grid minor
scatter(xdata_raw, ydata_orig);
plot(xdata_raw, ydata_orig, 'k--');
plot(xdata, g_normal, 'LineWidth', 2);
plot(xdata, g_fit, 'LineWidth', 2);

axis square
xlabel('\fontsize{14}Time');
ylabel('\fontsize{14}Amplitude');
title('\fontsize{16}Sani vs  Fit, Noise=0%');

yyaxis right
plot(xdata, err_normal);
plot(xdata, err_fit);
ylabel('\fontsize{14}Squared Error');

legend('Input Data', 'Correct Response', 'Sani Tu / Tg', 'Sani Fit', 'Error of Sani Tu / Tg', 'Error of Sani Fit', 'Location', 'east');
```