DigiSys

Simple object to represent digital systems, controllers, filters, etc.

This file contains LaTeXequations and is much better read on a dedicated markdown renderer. A PDF version is included for readability, but no guarantee that it's up to date with the markdown.

Background

Discrete linear time-invariant systems can be described by a linear constant coefficient difference equation of the following form

$$a_0 y_k + a_1 y_{k-1} + \cdots + a_{n-1} y_{k-(n-1)} = b_0 u_k + b_1 u_{k-1} + \cdots + b_{m-1} u_{k-(m-1)},$$

where u_k and y_k are the system input and output at timestep k. Applying the Z-transform yields the equivalent system description as a discrete transfer function

$$\frac{Y[z]}{U[z]} = \frac{b_0 + b_1 z^{-1} + \cdots + b_{m-1} z^{-(m-1)}}{a_0 + a_1 z^{-1} + \cdots + a_{n-1} z^{-(n-1)}}.$$

Specification of the coefficients $\{a_i, b_j\}$ fully defines a discrete-time system. In the most general case it is common to have an additional gain parameter K multiplying the transfer function, in which case a fully specified system must include the gain as well.

Signatures

This library defines a single object called `DigiSys` which can be used to represent a discrete time system up to order $\max\{n, m\} \leq 10$. **This constraint is arbitrary and can be increased by altering the `MAX_LEN` parameter in `DigiSys.h`**

The possible class signatures are as follows:

```
DigiSys(double num[], double den[], int num_len, int den_len);
DigiSys(double gain, double num[], double den[], int num_len, int den_len);
```

- `num[]` is an array of coefficients for the transfer function numerator, arranged as $\begin{bmatrix} b_0 & b_1 & \ldots & b_{m-1} \end{bmatrix}$.
- `den[]` is an array of coefficients for the transfer function denominator, arranged as $\begin{bmatrix} a_0 & a_1 & \ldots & a_{n-1} \end{bmatrix}$.
- `num_len` is the length of the transfer function numerator. Must satisfy $m \leq$ `MAX_LEN`.
- `den_len` is the length of the transfer function denominator. Must satisfy $n \leq$ `MAX_LEN`.
- `gain` is an optional parameter defining a constant K which scales the whole transfer function. Assumed to be unity by default.

`DigiSys` contains a single public method with the following signature

```
double DigiSys::update(double input);
```

- The argument `input` corresponds to the value of u_k at the current timestep.
- The return value corresponds to the value of y_k at the current timestep.
- The necessary values of u_{k-j} and y_{k-i} are stored internally in the object.

Example Code

Low-pass filter

Consider designing a first-order discrete filter which operates at 100 Hz has a cutoff frequency of 10 Hz. We will use Matlab to design the filter since it creates coefficient vectors in the desired form for the `DigiSys` object.

The filter can be designed using the following Matlab commands

```
ord = 1;          % filter order
fs = 100;         % sampling frequency [Hz]
fnyq = fs/2;      % Nyquist frequency [Hz]
fc = 10;          % filter cutoff frequency [Hz]

% calculate filter coefficients
[num, den] = butter(ord, fc/fnyq);
```

which results the following coefficient vectors

```
num =

    0.2452    0.2452


den =

    1.0000   -0.5095
```

These coefficients correspond to the transfer function

$$\frac{Y[z]}{U[z]} = \frac{0.2452 + 0.2452z^{-1}}{1.0000 - 0.5095z^{-1}}$$

which is a first-order Butterworth filter. Translating these into code:

```
#include <DigiSys.h>

const double num[] = {0.2452, 0.2452};
const double den[] = {1.0000, -0.5095};
const int m = 2;
const int n = 2;
DigiSys myFilter(num, den, m, n);
```

To use the filter on incoming sensor measurements, simply call the `DigiSys.update` method and pass in the raw measurement data. For example, in an Arduino environment:

```
void loop() {

    // get sensor reading
    int raw_measurement = analogRead(sensor_pin);

    // apply filter
    double filtered_measurement = myFilter.update(raw_measurement);

    /*
     * do something with the filtered data...
     *
     * wait until the next sample time...
     */
}
```

When applying the filter to sensor measurements, **ensure that the measurements are obtained at the same sample frequency that the filter was designed for.** Sampling at different frequencies will result in erroneous filter calculations. I strongly encourage the use of timer interrupts for this rather than loop delay.

Note on numerical accuracy: The transfer function defined above has a static gain of $(0.2452 + 0.2452)/(1.0000 - 0.5095) = 0.9998$, rather than the intended Butterworth gain of unity. This will result in numerical errors compared to the theoretical filter response which may or not be significant depending on the nature of the input and output signals. Accuracy can be improved by simply including more significant figures when coding the filter coefficients, but be aware that the results will not be exact.

Lead compensator with filtering

Consider a single-input single-output system controlled by a lead compensator. The compensator is given by the continuous-time transfer function

$$C(s) = 50\frac{s+1}{s+10}$$

which adds approximately 55° of phase at $\sqrt{10}$ rad/s and has a static gain of 5. To make the problem more interesting, also consider a first-order Butterworth filter applied to the sensor measurements like in the previous example, this time with a cutoff frequency of 100 rad/s. Rather than using the `butter` command in Matlab, the filter is described by the continuous-time transfer function

$$F(s) = \frac{100}{s+100}$$

The following Matlab commands will convert the combined compensator and filter into a discrete-time transfer function based on a sampling frequency of 1000 Hz

```
s = tf('s');
C = 50*(s + 1)/(s + 10);     % compensator
F = 100/(s + 100);           % filter
contr = F*C;                 % series connection of compensator and filter

dt = 1e-3;                       % sampling period [s]
contr_disc = c2d(contr, dt);     % convert to discrete
```

resulting in the following coefficient vectors

```
contr_disc.Numerator{:} =

         0    4.7364   -4.7317


contr_disc.Denominator{:} =

    1.0000   -1.8949    0.8958
```

These coefficients correspond to the discrete-time transfer function

$$\frac{Y[z]}{U[z]} = \frac{4.7364z^{-1} - 4.7317z^{-2}}{1.0000 - 1.8949z^{-1} + 0.8958z^{-2}}$$

Notice that the numerator coefficients include a leading zero which cannot be ignored without changing the transfer function. Also note that the gain parameter has been absorbed into the numerator coefficients. To demonstrate the overloaded constructor, we will separate the static gain and rescale the numerator coefficients to obtain the new transfer function with unit static gain

$$\frac{Y[z]}{U[z]} = 5.000\frac{0.9473z^{-1} - 0.9463z^{-2}}{1.0000 - 1.8949z^{-1} + 0.8958z^{-2}}$$

Translating this into code:

```
#include <DigiSys.h>

const double gain = 5.000;
const double num[] = {0, 0.9473, -0.9463};
const double den[] = {1.0000, -1.8949, 0.8958};
const int m = 3;
const int n = 3;
DigiSys myController(gain, num, den, m, n);
```

This filter + compensator can then be called in an appropriately timed loop and the control signals calculated from the raw measurements in one line

```
void control_function() {
// call this at 1000 Hz using a timer interrupt

    // get sensor reading
    int raw_measurement = analogRead(sensor_pin);

    // calculate control signal using combined filter and compensator
    double control_signal = myController.update(raw_measurement);

    /*
     * send control signal to actuator...
     *
     * maybe do some other stuff...
     *
     * wait until the next control cycle...
     */
}
```