AN4515

Processing Analog Sensor Data with Digital Filtering

Introduction

Authors: Bjørnar Moe Remmen, Lloyd Clark, Phillip Olk, Microchip Technology Inc.

This application note implements four different filter algorithms on MCUs of the AVR® EA Family. The filters are based either on original code or libraries. Code examples are publicly available.

In the code examples, a set of static sample data is fed into the filters, and the performance is analyzed. The actual filter speed can be of secondary importance when the overall processing speed is limited by the duration of the data acquisition. On the other hand, effective filtering reduces the number of CPU cycles and hence, power consumption.

Median Filter

The median filter is based on comparisons, which do not require CPU-intensive operations such as multiplications or divisions. The center value of an odd number of samples is determined by a sequence of comparisons.

Median filters are excellent for removing spikes in signals like gaussian-, burst- or impulse noise.

Fourier Transform

The Fast Fourier Transform (FFT) is a method to decompose a signal into its frequency components, where each component has a frequency, an amplitude, and a phase.

The FFT creates a rather intensive CPU load, but it also produces a large set of information about the input signal. A single FFT operation can identify frequencies and amplitudes of peaks and hence help isolate/reject higher harmonics and side-bands and can be the core algorithm of a digital equalizer.

Infinite Impulse Response Filters

Infinite Impulse Response (IIR) filters are a class of filters that are feedback-based, i.e., the previous output plays a role in the current output. The implementation is simple but versatile: High and low-pass filters, notches and band-pass filters, even gain can be implemented with a rather small set of coefficients.

Due to the feedback principle, these filters lose the phase information and might be unstable - nonetheless, they are good for filtering sensor readings.

Kalman Filter

The Kalman filter is based on weighted averaging. The weight factors are determined by uncertainty calculations, where measurements with higher certainty are weighted heavier. From the weighted average, a prediction of the following measurement is created. The “correctness” of that prediction, i.e., how much the next measurement is off of the prediction, is used for weighting that measurement.

This filter is well-suited for noisy sensor data in systems with continuous, stepless behavior.

Overview

Processing analog sensor data with digital filtering can be a resource-intensive job for a Microcontroller Unit (MCU). This application note presents multiple digital filter algorithms, compares their properties, and provides code examples that are good starting points for application-specific filtering.

In digital signal processing, a vast choice of filtering algorithms is readily available in libraries or can be easily implemented in C code. Not all are well-suited for the relatively limited resources of 8-bit MCUs: Restrictions in memory size, speed, and power consumption force the application designers to compromise. In this application note, we present readily available libraries for several filtering algorithms that are easy to implement and use on AVR® MCUs. These filters are:

- Median filter
- Fast Fourier Transform (FFT) using the kissFFT library
- Infinite Impulse Response (IIR) using a bi-quadratic algorithm
- Kalman filter using the kalman-clib library

In the following sections, we are giving a short introduction to the filters, comment on their usefulness and applications, quantify their CPU load, and provide sample code.

Table of Contents

1. Relevant Devices

The code examples on GitHub are created for AVR EA devices. The parts of the code examples dealing with algorithms are generally valid for all AVR devices, as are the general principles and performance reports in this document. Limitations due to memory sizes apply. The code sections for configuring and executing the communication between the device and Data Visualizer (i.e., USART peripheral and pins) may require adaptation.

2. Median Filter

Principle

The Median filter smooths a signal by removing spikes. The median filter can be used both on images and on 1-dimensional signal problems, which will be covered here. The median is calculated by sorting a list of numbers and finding the number in the middle.

The median filter has a window size, determining how far it will look back. For example, a window size of `9` will have eight elements looking back and one new element. The filter returns the median of these nine elements.

Example 2-1. Finding the Median of an Array

Consider the array `[31, 15, 8, 91, 31, 90, 1, 5]`. After sorting, the element in the middle (here: The 5th position) is the median of the array:

```
[1,5,8,9,15,31,31,90,91]
```

When conducting continuous measurements, the oldest element in the array is replaced with the new value, and after sorting, a new median can be determined.

The implementation we chose can be found at github.com/accabog/MedianFilter and was written by Alexandru Bogdan.

Example

In this section, we will discuss the implementation and the example code.

In the *main.c* file, the code has two integer values: The original value and the filtered value. The variable `i` iterating over the sine wave from the file *filter/sine.h*.

```
volatile uint16_t original = 0;
volatile uint16_t filtered = 0;
uint8_t i = 0;
```

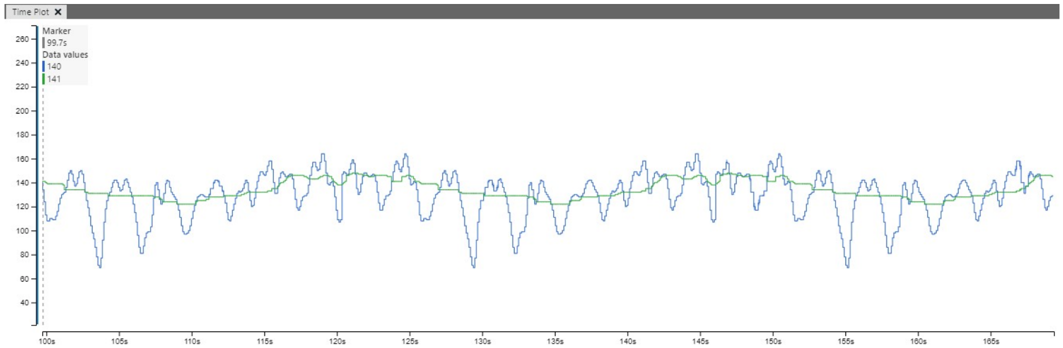
The main code first initializes PORT D 6 (PD6) as an output. A `while` loop generates a new integer value to be fed as an input to the median filter. The PD6 output is set, and then the median filter is called. Just after the median filter, the PD6 output is cleared. Thus, an oscilloscope or a logic analyzer can be connected to pin PD6. By measuring the duration of the high-level pulse, the execution time of the `MEDIANFILTER_Insert()` function can be determined .

```
original = sinewave[i++];
_delay_ms(100);
// Using PINs to measure duration of median filter
PORTD.OUTSET = PIN6_bm; // Make PD6 output logic low
filtered = MEDIANFILTER_Insert(&medianFilter, original);
PORTD.OUTCLR = PIN6_bm; // Make PD6 output logic high
```

Example 2-2. Visualization of Median filter

A code block has been added that sends both the original signal and the filtered signal over USART0. This signal can be viewed using the Data Visualizer. Here visualization is done using window size 31.

```
// sending data over usart
variableWrite_SendFrame(original, filtered);
```



The green is the filtered signal, and the blue is the original signal. Using the MPLAB® Data Visualizer to compare the two signals sent over the Universal Synchronous and Asynchronous Receiver and Transmitter - USART, there is a clear difference between the two. The median filter removes the spikes in the signal, and it becomes the median of the current signal and the previous samples. In this case, the filtered signal results in a cleaner sine wave.

To Use the Data Visualizer, click Load Workspace → Choose *data_visualizer.dvws*

Performance and Properties

The median filter has a selectable window size. This window size affects both the ability to filter and how long it takes to process.

To measure the time/cycles on **PORT D** on **PIN 6**, connect the PIN to a logic analyzer, and by knowing the device speed, it is possible to time the filter.

Table 2-1. Cycle Times - Median Filter

Filter Size	Cycles
7	215 - 385.8
15	225.8 - 601.8
31	344.8 - 1034

Because of the type of implementation, the number of cycles varies from sample to sample. The reason is the amount of sorting work when inserting a sample. It depends on how many updates the `if`-statement does and if the `for`-loop can exit early: If a number is smaller than one of the values in the array, the loop is exited. In the worst case, it needs to iterate through the entire list of values, which is the window size.

The value of the `i` is later used for adjusting the median when needed. The algorithm does not account for the extraordinary situation with even numbers where the median would be the average of two numbers.

```
for(i = 0; i < medianFilter->numNodes - 1; i++)
{
    if(sample < it->value)
    {
        if(i == 0)
        {   //replace value head if new node is the smallest
             medianFilter->valueHead = newNode;
        }
        break;
    }
    it = it->nextValue;
}
```

Conclusion and Use Cases

We have shown briefly how a median filter works and how to use it. We also demonstrated how to measure the cycle times for your application using an oscilloscope or logic analyzer.

A median filter is a nice method for removing noise or peaks from the signal and can often be used as a preprocessing step in front of more advanced filters, like a Kalman filter. The difference between a Median filter and more advanced filters is that a Median filter does not fold the extreme values into the signal like in an average filter and therefore removes their impact on the signal.

3. Infinite Impulse Response (IIR) Filters

Principle
Infinite Impulse Response (IIR) filters are feedback-based filters, i.e., the previous output plays a role in the current output. Due to the feedback principle, these filters lose the phase information and might be unstable, so they are not always the first choice for audio applications. But for sensor readings, they can be a great tool.

For this application note, a simple and flexible implementation of IIR was chosen, with emphasis on ease of use.

An IIR filter can be described by the differential equation:

$$y[n] = \sum_{k=0}^{M-1} b_k x[n-k] - \sum_{k=1}^{N-1} a_k y[n-k]$$

The implementation we have is an example of a third-order IIR filter.

In the initialization phase, the b_{0-2} and a_{0-2} are set based on the type of filter, e.g., Low-Pass Filter. When running the filter, x_{n-k} and y_{n-k} values are updated continuously by the current signal, while a_k and b_k are constants.

Note: The notation in the code differs from the notation used in the general equations.

Table 3-1. Variable Explanation for Code

Variable in Code	Variable in Equation
a_0	$\frac{b_0}{a_0}$
a_1	$\frac{b_1}{a_0}$
a_2	$\frac{b_2}{a_0}$
a_3	$\frac{a_1}{a_0}$
a_4	$\frac{a_2}{a_0}$

```
/* Computes a BiQuad filter on a sample */
smp_type BiQuad(const smp_type sample, biquad* const b)
{
  smp_type result;

  /* compute result */
  result = b->a0 * sample + b->a1 * b->x1 + b->a2 * b->x2 -
    b->a3 * b->y1 - b->a4 * b->y2;

  /* shift x1 to x2, sample to x1 */
  b->x2 = b->x1;
  b->x1 = sample;

  /* shift y1 to y2, result to y1 */
  b->y2 = b->y1;
  b->y1 = result;

  return result;
}             // Bandwidth in Octaves
```

When using the normalization factor a_0, the equation becomes:

$$y[3] = \sum_{k=0}^{3-1} b_k x[n-k] - \sum_{k=1}^{3-1} a_k y[n-k] \Rightarrow \left(\frac{b_0}{a_0} \times signal + \frac{b_1}{a_0} \times x_1 + \frac{b_2}{a_0} \times x_2\right) - \left(\frac{a_1}{a_0} \times y_1 + \frac{a_2}{a_0} \times y_2\right)$$

For further reading, we refer to the IIR filtering sections from the music department at the University of California - San Diego: musicweb.ucsd.edu/%7Etrsmyth/filters/Biquad_Section.html.

Even if the IIR filter implementation we have used has support for multiple filters, we will only cover **Low-Pass** and **Band-Pass**. The complete list of supported filters is:

- Low-Pass Filter
- High-Pass Filter
- Band-Pass Filter
- Notch Filter
- Peaking Band EQ Filter
- Low Shelf Filter
- High Shelf Filter

Examples

In this section we will examine the implementation in the example code.

Note: The implementation uses floating-point arithmetic, which can slow down the speed of the filter compared to an integer implementation.

In `main.c`, we create a hard-coded signal to illustrate the filter.

```
const uint8_t  sinewave[] = {147, ..., 135};
```

The signal passes the filter. To handle the the filtered results, the library creates a `bq` struct, which is made by setting the sample rate, frequency, bandwidth and gain. (Gain is used only by the Low Shelf and High Shelf filters.)

```
biquad *bq = BiQuad_new(FILTER_TYPE, dbGain,      /* gain of filter */
                          freq,                   /* center frequency */
                          srate,                  /* sampling rate */
                          bandwidth);             /* Bandwidth in Octaves */
```

To filter, send one value to biquad with the initialized `bq` struct.

Note: Further down in the application note, it says that it is possible to switch data type. The input value will be implicitly cast to that data type.

```
filtered = BiQuad(sinevalue, bq);
```

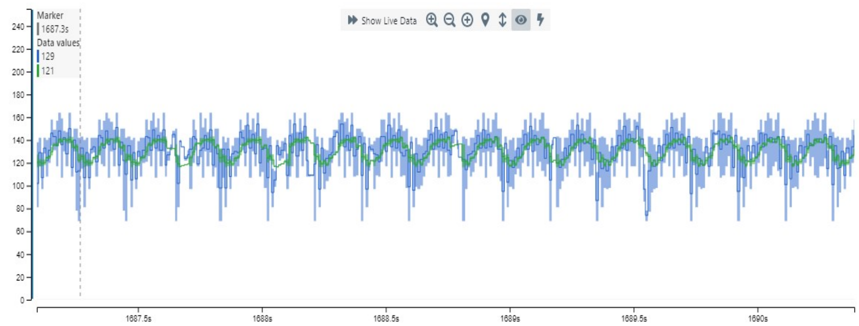
Example 3-1. Example: Low-Pass Filter

A low-pass filter is a filter that passes the signals lower than a given frequency threshold *freq* and attenuates/rejects the signal outside that range. Often, for filters with high/low discrimination, the term "bandwidth" denotes that frequency threshold (think of a low-pass filter as a "band-pass from zero to *bandwidth*" and a high-pass filter as a "band-pass from *bandwidth* to infinity"). Here, the variable *bandwidth* determines the shape of the transition between high/low areas. The variable *srate* helps to determine the phase Ω between samples.

```
    smp_type freq = 8.0;                    // Frequency in Hertz
    smp_type srate = 255.0;                 // Samples per second
    smp_type bandwidth = 5;                 // Bandwidth in Octaves
```

To Use the Data Visualizer, click Load Workspace → Choose *data_visualizer.json*

Figure 3-1. Input Signal and Low-Pass Filter Output



The green is the filtered signal, and the blue is the original signal. Using the MPLAB® Data Visualizer to compare the two signals sent over the Universal Synchronous and Asynchronous Receiver and Transmitter - USART, it is clear that the Low-Pass signal has removed high-frequency components.

Example 3-2. Example: Band-Pass Filter

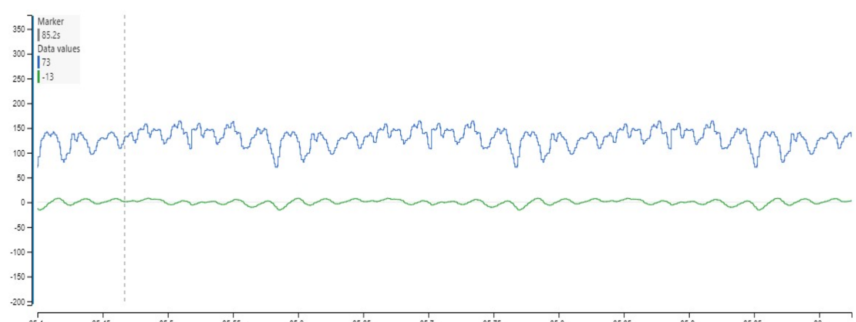
A band-pass filter is a filter that rejects or attenuates signals outside of a given frequency range and accepts those inside.

Ideally, a band-pass filter does not attenuate or amplify signals inside the range and completely remove the signal outside the frequency range. The center frequency of the band-pass filter is determined by `bandwidth`.

```
smp_type freq = 3.0;                  // Frequency in Hertz
smp_type srate = 255.0;               // Samples per second
smp_type bandwidth = 1;               // Bandwidth in Octaves
```

Note: A calculator for bandwidth in octaves can be found at www.sengpielaudio.com/calculator-bandwidth.htm.

Figure 3-2. Low-Pass Filter Output Before and After Band-Pass Filter



Using the MPLAB® Data Visualizer to compare the two signals sent over the Universal Synchronous and Asynchronous Receiver and Transmitter - USART, the filter effect is visible: The Band-Pass filter has removed noise outside of a set frequency using the parameters for the IIR filter, and the result is the green filtered signal. To Use the Data Visualizer, click Load Workspace → Choose *data_visualizer.json*.

Notice that the signal is centered around 0 because the DC offset of the input signal (blue) is equivalent to a frequency of 0 Hz. This is below the pass-band of the filter and hence, removed, meaning that the band-pass filter output (green) has no DC offset and is centered around 0.

Performance and Properties

The IIR filter can use either double or float for the filters, which has a minor impact on how fast the filter will be.

To measure the time/cycles, set **ONLY_SEND_USART** to '`0`' in `main.c`. This allows to measure the time between ticks on **PORT B** on **PIN 2**. Connect the PIN to a logic analyzer, and, by knowing the speed of the device, it is possible to time the filter.

Table 3-2. Cycle Times - IIR filter

Data Type	Cycles
`float`	1280

Some combinations of compiler and device architecture support changing the size of `double`, e.g., from 32-bit to 64-bit. An increase improves the precision, but with a cost of a slower run-time. Integers will not work with this implementation.

Use the type definition `typedef float smp_type;` in **biquad.h** to change the data type in the filter implementation. The compiler may require additional arguments, such as `-fno-short-double`. See the compiler's documentation for details.

Conclusion and Use Cases

We have demonstrated how to use a biquadratic IIR filter on AVR devices. In addition, we have introduced how to time the filter in the code.

The IIR, as applied here, offers several filtering options. In combination with a low CPU load, IIR is a valuable tool for digital signal processing, especially on an 8-bit microcontroller. Depending on the application, IIR can be used to reject noise or offset variations, which helps to isolate the desired signal component. Thanks to its rather low CPU load, the IIR can easily fit into load- or timing-sensitive applications.

4. Fast Fourier Transform (FFT)

Principle
The Fast Fourier Transform (FFT) is a versatile tool for signal analysis. The general idea, in terms of electronic signals, is to de-compose a given signal (in the time domain) into sine-shaped components (in the frequency domain). Each component has a frequency, a phase, and an amplitude. The inverse operation is adding all components and will return a signal that is, in a theoretical limit, identical to the original input signal. By providing the parameters of the components, FFT can serve as a filtering tool: It can isolate a periodic signal in a noisy environment and quantify amplitudes and frequencies, detect frequency shifts of a signal, or distinguish amplitudes in high and low-frequency bands for controlling your disco lights.

Several libraries are available that implement FFT and could run on AVR devices. Here, **kissFFT** was chosen because it focuses on simplicity and compactness: It is easy to integrate into an AVR project, it is implemented in C, and has a useful licensing model (BSD-3-Clause). On the other hand, there are no windowing functions (aside from the implicit rectangular window when picking data blocks out of a stream) and no analysis features such as peak localization - this is up to the application. You may prefer different criteria for your library or even implement your own FFT functionality, but the general upshots from this document are also valid there.

Example
In this section, we examine the implementation and the example code we have created.

The code shows how to run FFT and measure the speed of execution. For simplicity, we use a hard-coded input signal for the FFT algorithm:

```
const int16_t sinewave[1024] = { ... };
```

The signal is the same periodic signal as used in the *Infinite Impulse Response (IIR) Filters* section. It has these properties:

- Sine wave with fundamental frequency f_0 = 440 Hz
- Sampling rate f_S = 44.1 kHz
- DC offset
- Strong harmonic overtones (”fuzz” effect, deliberately added)

The `nfft` parameter selects the sample length in kissFFT, i.e., how many data points are used for the FFT operation. It also determines the number of frequency bins (`nfft / 2`) in the operation’s output and hence, the width of each bin (f_S/`nfft`). Consequently, `nfft` affects the speed of the filter.

In the example, we select:

```
int nfft = 800;
```

and therefore:

- Number of bins `nfft / 2 = 400`
- Bin width = $2f_S$ / `nfft` ≈ 110 Hz

The resulting bin width will work well with the known fundamental frequency of 440 Hz of the example - but for speed optimization, use values for `nfft` that are powers of two (e.g., 64 or 256).

For real signals in the time domain (just like a sequence of ADC conversions provides), the kissFFT library recommends this function:

```
    kiss_fftr(cfg, cpx_in , cpx_out);       // The actual FFT operation
```

where `cfg = kiss_fftr_alloc(nfft, 0, 0, 0)` is a required set of configuration parameters, `cpx_in` is the input data (real) and `cpx_out` is the complex output data.
Note: The kissFFT library provides other functions that may have complex input. For this reason, the input variable name has the prefix `cpx`. Nonetheless, the input signal is a sequence of ADC sampling values and hence, comprises only real numbers.

Now the application can proceed and evaluate specific information in `cpx_out[n]` for its purposes. In our case, we calculate the power spectrum `pwr`, send it to the PC using the USART peripheral, and plot it on the PC using the Data Visualizer:

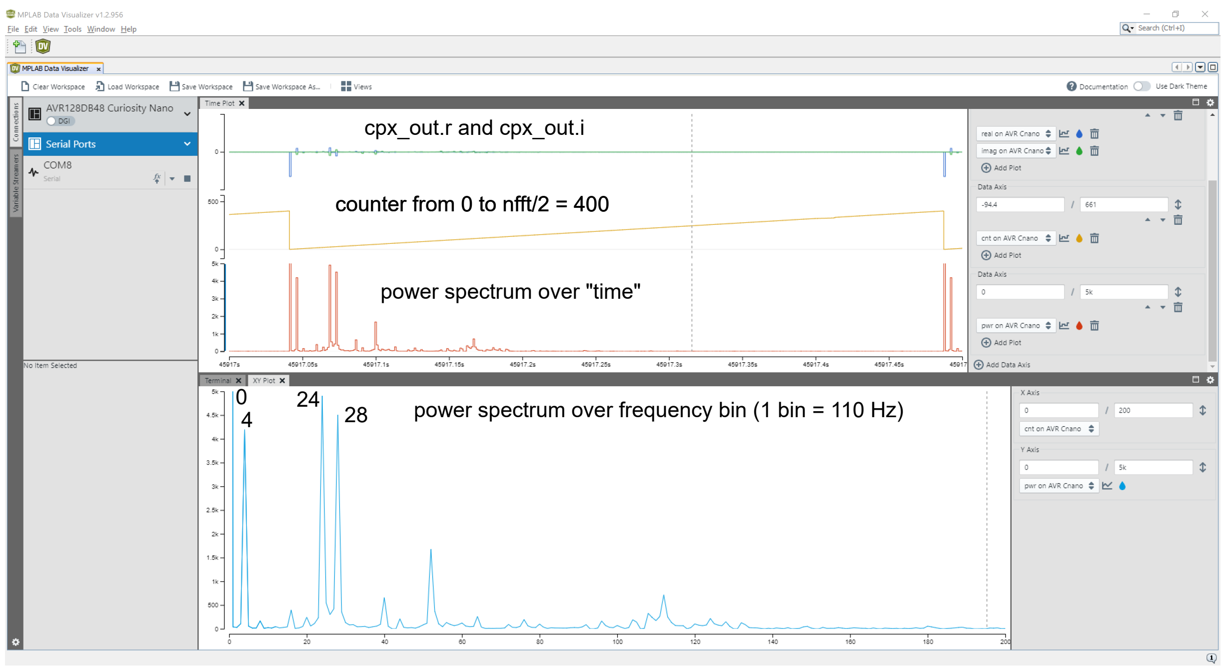
First, calculate `pwr`:

```
//Calculating the power spectrum
pwr = sqrt(cpx_out[n].r * cpx_out[n].r + cpx_out[n].i * cpx_out[n].i);
```

For each frequency bin *n*, the absolute value of the complex result vector `cpx_out[n]` is calculated from its real and imaginary components, `cpx_out[n].r` and `cpx_out[n].i`, respectively.

When the device is connected to a PC, the MPLAB® Data Visualizer can be used for signal analysis. The following figure shows the FFT of the aforementioned input signal. The upper plot displays the variable values, as sent by the USART: The real and the imaginary components of `cpx_out[n]` share one scale. Next is the iteration counter in yellow, followed by the power spectrum in red.

Figure 4-1. FFT of a Sine Wave with Low Amplitude



The presentation of the power spectrum over “USART transmission time” (red) is not the most useful since it is difficult to pinpoint exact peak locations. The MPLAB Data Visualizer’s XY Plot feature can plot the power spectrum over the frequency bin number using the iteration counter as X-value, see the lower plot in the figure above (blue). Keeping in mind that one bin is approximately 110 Hz in width, we can identify and quantify peak locations and amplitudes:

- Bin 0 is defined by the kissFFT library to contain the DC offset of the signal (clipped for legibility)
- Bin 4 contains the fundamental, around 440 Hz
- The natural harmonics can be identified:
  - 2nd harmonic at bin 8
  - (There is no 3rd harmonic component at bin 12)
  - 4th harmonic at bin 16
  - 5th harmonic at bin 20
- The “fuzzy” look of the original signal is caused by the dominant components in bins 24 and 28, representing the sixth and seventh harmonics at around 2.65 kHz and 3.09 kHz, respectively
- Even higher harmonics and high-frequency noise components can be identified, for example, the tenth harmonic at bin 40

Performance and Properties

In comparison to other filter techniques, FFT is not easy on CPU load: depending on `nfft` (and the application's requirements), the device's CPU can be busy for a substantial time.

When the actual device is accessible, the duration of one FFT can be measured by observing the output level of a digital output pin with an oscilloscope or a logic analyzer. In the example, the Curiosity Nano board LED pin can be toggled:

```
PORTD.OUTSET = PIN6_bm; // Make PD6 output logic high
kiss_fftr(cfg, cpx_in , cpx_out);      // The actual FFT operation
PORTD.OUTCLR = PIN6_bm; // Make PD6 output logic low
```

The measured duration can be converted to 'number of FFT per MHz' to compare performance or scale it with clock frequency. This method is recommended for applications where timing and CPU load are critical.

When the device or the required tools are not accessible, you can use MPLAB X to measure the speed of the algorithm on a *simulated* AVR device:

1. Select "Simulator" as the connected tool in the project properties.
2. Add suitable breakpoints right before and after the function call `kiss_fftr(...);`.
3. Find the Stopwatch window (Window → Debugging → Stopwatch).
4. Run the debugger.
5. When the debugger halts at a breakpoint, the Stopwatch window will display the number of virtual clock cycles used since the previous breakpoint or start and the current breakpoint.

This approach provides a quick and easy estimation over the upper theoretical performance limit.

The following table lists measured performance done on the Curiosity Nano.

Table 4-1. Cycle Times - FFT filter

nfft	Cycles Per FFT
4	5,300
8	12,000
16	30,100
32	67,000
64	161,000
128	348,000
256	810,000
512	1,750,000

As an example, an FFT with `nfft=64` uses about 161 kilocycles per FFT, corresponding to 6.2 FFT per MHz. Consequently, a device running at a core frequency of 20 MHz could perform twenty times as many, i.e., 124 FFT per second. If the device uses an external crystal with only 32.768 kHz, the same FFT operation takes apporximately five seconds.

Note:

The maximum `nfft` value is depending on the actual memory configuration of the device and the memory requirements of the rest of the application.

For speed optimization, use values for `nfft` that are powers of two (e.g., 64 or 256).

Conclusion and Use Cases

We have shown that an AVR device can perform Fast Fourier Transform using a third-party library (kissFFT). The FFT can provide valuable information other digital filtering techniques can't, but at the cost of CPU cycles: It depends on the application's requirements and details whether the FFT can be run continuously or not. Alternatively, an FFT can only occasionally be used to optimize parameters for other, faster filters.

Another point to keep in mind is the virtual precision, as suggested by the power spectrum, vs. the actual accuracy of the measurement:

One aspect is the quality of the input sample data and the time base used: The oscillator providing the time base can suffer a temperature drift of several percent. This is less of a problem for self-contained applications for signal analysis, especially when there is a short time between sampling and usage of the FFT. Though, it is relevant when interacting with external devices that have their own time base.

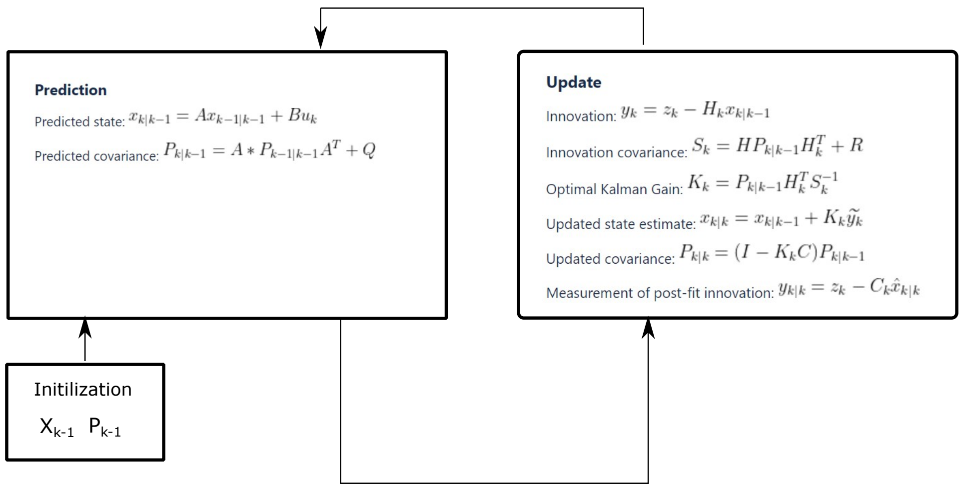
The other aspect is the width of a frequency bin, determined by the signal's sampling rate and `nfft`: In the example above, the power spectrum was well suited to identify higher harmonics. On the other hand, even when exploiting the tenth harmonic, we can not determine whether the fundamental component was actually tuned to 440 Hz or, musically speaking, de-tuned.

5. Kalman Filter

Principle

The Kalman filter, also called **linear quadratic estimation** (**LQE**), is an algorithm that uses a series of measurements to estimate unknown variables in the future. State estimation can, for example, be used to predict the placement of a robotic vacuum cleaner to avoid hitting walls or for creating a balancing robot.

The figure below shows the general principle of the Kalman filter. After initialization, the prediction and update phases happen when the algorithm runs. The update phase updates the variables based on the error it had from the prediction.



Example

This section will go through the example code on GitHub. The example we are using is done with a 3x3 matrix but can be changed for other examples if needed.

```
int main(void)
{
    SYSTEM_Initialize();


    PORTD.DIRSET = PIN6_bm;

    matrix_unittests();
    while(1){

        if(SEND_OVER_USART)
        {
            // Code that sends the data over USART.
            kalman_gravity_demo_usart();
        }else{
            // Regular code, without USART for easier cycle
            // measurement on PIN.
            kalman_gravity_demo();
        }

        kalman_gravity_demo_lambda();
    }
}
```

Example 5-1. Kalman Gravity Demo

This demo calculates the gravity *g* based on the position *S* of a free-falling test body and is written by Markus Mayer. The source can be found at https://github.com/sunsided/kalman-clib.

The code uses three states (position, velocity, gravity), where gravity is the estimated acceleration based on the traveled distance, velocity and previous estimate of acceleration. As mentioned in the code, the formulas for this experiment are:

$$S = S + v * T + g * 0.5 * T^2$$

$$v = v + g * T$$

$$g = g$$

with:

- *S* position in m
- *v* velocity in m/s
- T time, stepped in 1 s
- g gravity, with an initial value of 6 $\frac{m}{s^2}$. This is a “bad estimation” which the Kalman filter shall improve.

The values for *S* are “noisy” positions, recorded with one second in between them.

The code prepares the initial values and iterates over the distances to predict the relative acceleration of the test body. The sequence is:

1. Predict the following distance.
2. Measure the actual following distance.
3. Calculate the error.
4. Update the Z matrix.

The Z matrix is further used in the *kalman_correct()* routine, which updates and corrects the variables, so the sequence can repeat.

When going through the sequence a couple of times, the calculated value for *g* quickly approaches a final value.

To measure the speed of the Kalman filter, connect an oscilloscope or a logic analyzer to PORT D - PIN 6 and measure the time when the pin is high. A more thorough explanation of how to use the Kalman filter is in the README.md of the code.

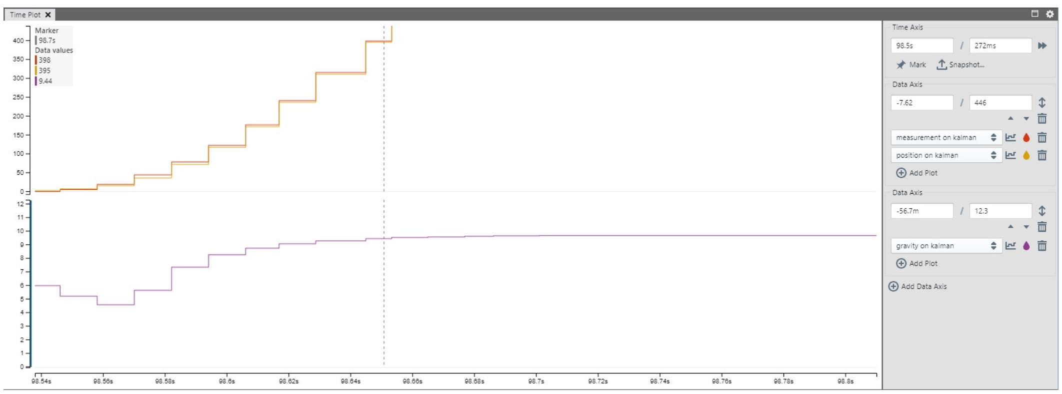
```
void kalman_gravity_demo()
{
    PORTD.DIRSET = PIN6_bm;

    // initialize the filter
    PORTD.OUTSET = PIN6_bm;
    kalman_gravity_init();
    PORTD.OUTCLR = PIN6_bm;

    // fetch structures
    kalman_t *kf = &kalman_filter_gravity;
    kalman_measurement_t *kfm = &kalman_filter_gravity_measurement_position;

    matrix_t *x = kalman_get_state_vector(kf);
    matrix_t *z = kalman_get_measurement_vector(kfm);

    // filter!
    for (int i = 0; i < MEAS_COUNT; ++i)
    {
        PORTD.OUTSET = PIN6_bm;
        // prediction.
        kalman_predict(kf);
        // x[0] -> s_i -> estimate next position
        // x[1] -> v_i -> estimate next velocity
        // g[2] -> g_i -> estimate next gravity
```

```
            // measure ...
            matrix_data_t measurement = real_distance[i] + measurement_error[i];
            // reading position as float -> converting to uint8 array
            volatile uint8_t* measurement_b = (uint8_t*) &measurement;
            matrix_set(z, 0, 0, measurement);

            // update
            kalman_correct(kf, kfm);
            PORTD.OUTCLR = PIN6_bm;
            PORTD.DIRCLR = PIN6_bm;

        }


        // fetch estimated g
        matrix_data_t g_estimated = x->data[2];
        assert(g_estimated > 9 && g_estimated < 10);
    }
```

To illustrate the continuous improvement of the estimates, set `SEND_OVER_USART` to TRUE in peripherals/usart.h. This will make sure to use the `kalman_gravity_demo_usart()`, which sends the data via USART.

Note: To get a more accurate cycles measurement of the algorithm, set `SEND_OVER_USART` to FALSE when doing cycle measurements.

Configure the Data Visualizer by selecting *Load Workspace*. Choose the configuration file *data_visualizer.dvws*.



The yellow line is the measurement of the position (i.e., sensor reading), the red line is the predicted position for the following measurement, and the purple line is the calculated gravity.

One new measurement is taken every second, giving feedback to the estimates of the next position and gravity *g*. The plot of gravity *g* starts with the initialization value of $6\ \frac{m}{s^2}$ and improves the estimates towards $9.81\ \frac{m}{s^2}$.

Performance and Properties

Table 5-1. Cycle Times - Median Filter

Matrix Size	Initialization	One Estimation
A=3x3, B=null	1700	440

The Kalman filter is an advanced filter algorithm, and therefore, the long run-time of this algorithm is not unexpected. The time consumption further increases for larger matrix sizes. Smaller matrix sizes run faster.

Conclusion and Use Cases

This application note has given an overview of how to use this filter, provides typical cycle times of the filter, and walks us through an example to calculate the acceleration from position measurements.

A Kalman filters can be used in many situations where it is beneficial to know the next state based on previous input. Use cases for the Kalman filter are often found in mobility applications, such as balancing robots, vacuum cleaners, and even self-driving cars.

6. Conclusion

This document has introduced four digital signal filtering methods and demonstrated their use on AVR devices. Each of the filtering methods has a spectrum of advantages - and disadvantages:

Median Filter The median filter, often overlooked, is a rather simple but effective “signal smoothing” technique: It removes spikes and other glitches nicely and demands only a few CPU cycles. The downside is that the result could become too smooth and “rounded.”

IIR Filter The Infinite Impulse Response filter offers several features from “classic” signal filtering - namely variants of high, low, and band-pass. The versatility and moderate consumption of CPU cycles make it an attractive tool.

FFT The Fast Fourier Transform is not a classic filter per se, but it reveals a lot of information about the input signal. Hence, FFT is a mainstay of digital signal processing. From an 8-bit MCU perspective, FFT can demand a lot of resources in terms of memory and CPU cycles. It can provide substantial help in interpreting the input signal when applied thoughtfully.

Kalman Filter Neither a classic filter, the Kalman filter has similarities with a controller: A noisy input signal can be used to determine an (output) parameter. The CPU consumption is moderate. The Kalman filter shines in settings where the actual value is expected to propagate continuously.

7. Get Code Examples from GitHub

The code examples are available through GitHub, which is a web-based server that provides the application codes through a Graphical User Interface (GUI). The code examples can be opened in MPLAB X.

The GitHub webpage: GitHub.

Code Examples

View Code Examples on GitHub
Click to browse repositories

Download the code as a `.zip` file from the example page on GitHub by clicking the **Clone** or **download** button.

8. Revision History

Doc. Rev.	Date	Comments
A	04/2022	Initial document release

Microchip Information

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "unbreakable". Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, CryptoMemory, CryptoRF, dsPIC, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, Flashtec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet- Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, TrueTime, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, GridTime, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, Knob-on-Display, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, NVM Express, NVMe, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SmartHLS, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, Symmcom, and Trusted Time are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-6683-0166-1

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820