DSP: Underlying Concepts ## (Slides/Additional Notes) ## William Eustace M0WJE ## m0+rsgb2020@wje.io

Note: this document contains the set of slides presented with some additional derivations and explanation. Interactive diagrams have been converted into static ones for PDF export; the code used to generate them is shown, which may prove interesting for readers wishing to play around further with the filtering slides shown later.

Allergen notice/Disclaimer/Apology

This talk contains mathematics (with explanation) but is guaranteed to be free of practical advice.

1. The description I wrote back when I had to come up with a title was somewhat misleading; even then, I never described this as "easy-to-understand"!
2. Roughly A level mathematics knowledge is advised: DSP is by nature not a maths-free zone.
3. This talk aims to give you the underlying background to signal processing: why does it work? It will **not** be a tutorial to build a working system, and will not give you most of the tools necessary to do so.
4. That said: if you want to build something that works in this field, it will help to understand **why** it works!

Today's Objectives

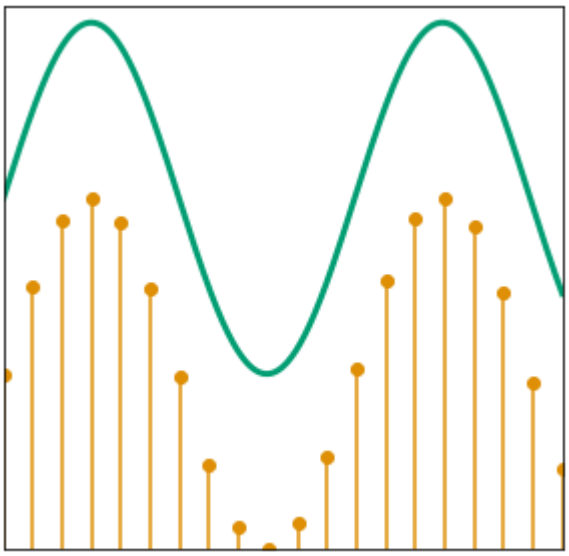
1. Time domain, Frequency domain, and how to go from one to the other.
2. Nyquist's Theorem (very briefly!)
3. Impulse Responses & Convolution.

What is a signal?

- For our purposes: a time-varying function $f(t)$.
- Analogue or digital?
- Discrete or continuous?

In [1]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,10,1000)
s = np.sin(t)+2
plt.plot(t,s,lw=3,color=colour_palette[2],linestyle='-')
t_discrete = np.linspace(0,10,20)
s_discrete = np.sin(t_discrete)+1
plt.scatter(t_discrete,s_discrete,color=colour_palette[1])
for td,sd in zip(t_discrete,s_discrete):
    plt.plot([td,td],[0,sd],color=colour_palette[1])
plt.xticks([])
plt.yticks([])
plt.xlim([0,10])
plt.ylim([0,3.1])
# plt.grid(True)
print()
```



Essential Concepts

- A **sum**:

$$\sum_{n=0}^{M} f(nT) = f(0 \times T) + f(1 \times T) + \ldots + f(M \times T)$$

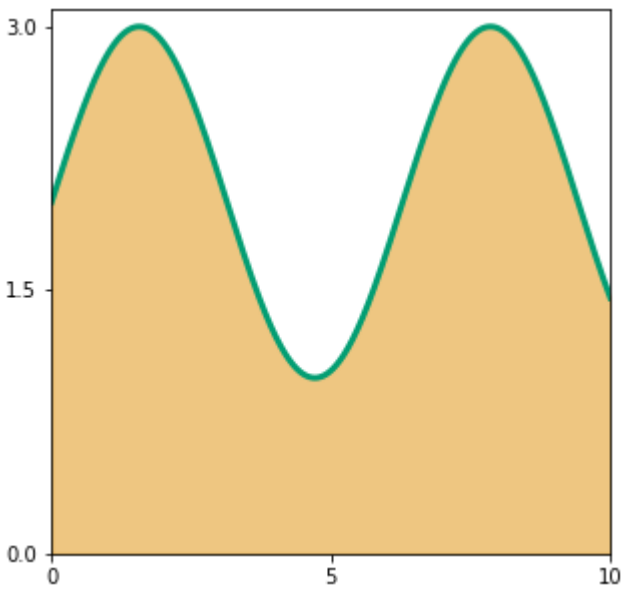
(for each value of n, n=0, n=1, n=2 ... n=M, evaluate the contents of the sum (in this case f(nT)) and add them up)

- An **integral**:

$$\int_{-\pi}^{\pi} f(t)dt$$

(find the area under f(t) with variable t along the x-axis (indicated by "dt") between $t = -\pi$ and $t = \pi$

In [2]:
```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,10,1000)
s = np.sin(t)+2
plt.plot(t,s,lw=3,color=colour_palette[2],linestyle='-')
plt.fill_between(t,0,s,alpha=0.5,color=colour_palette[1])
plt.xticks([0,5,10])
plt.yticks([0,1.5,3])
plt.xlim([0,10])
plt.ylim([0,3.1])
# plt.grid(True)
print()
```



$$f(t) = sin(t) + 2$$

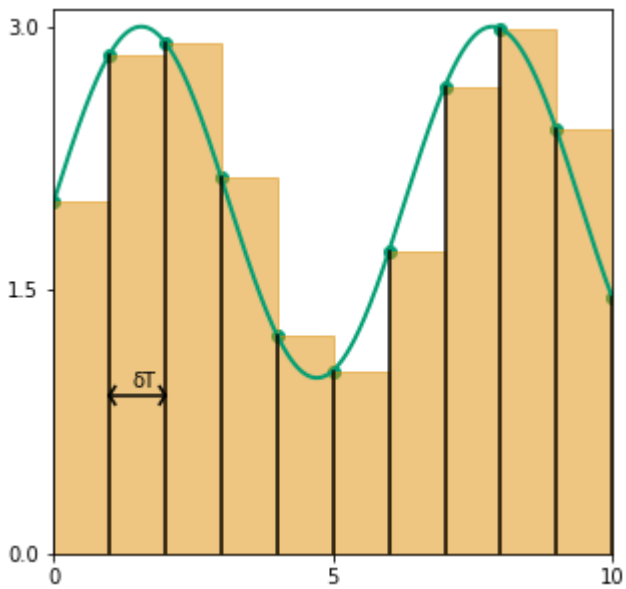
The integral:

$$\int_0^{10} f(t)dt$$

is just the area shaded in orange above. We can also approximate this by a sum:

In [3]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,10,1000)
s = np.sin(t)+2
plt.plot(t,s,lw=2,color=colour_palette[2],linestyle='-')
t_points = np.linspace(0,10,11)
s_points = np.sin(t_points)+2
plt.scatter(t_points,s_points,color=colour_palette[2])
for (x,y) in zip(t_points,s_points):
    plt.plot([x,x],[0,y],color='black')
    plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_pal
ette[1])

plt.xticks([0,5,10])
plt.yticks([0,1.5,3])
plt.xlim([0,10])
plt.ylim([0,3.1])
plt.annotate(s='δT', xy=(1.4,0.95))
plt.plot([1,2],[0.9,0.9],color="black")
plt.plot([1,1.1],[0.9,0.85],color="black")
plt.plot([1,1.1],[0.9,0.95],color="black")
plt.plot([2.0,1.9],[0.9,0.85],color="black")
plt.plot([2.0,1.9],[0.9,0.95],color="black")
# plt.grid(True)
print()
```



By adding up the areas of the rectangles, width δT, we clearly approximate the area; in fact, if we allow the width of the rectangles to tend to zero, this gives the precise area. This is a definition of an integral. Mathematically:

- A **sum** which (**under some conditions**) *converges* to an **integral**[1]:

$$\lim_{\delta T \to 0} (\sum_{n} f(n\delta T)\delta T) = \int f(t)dt$$

(as drawn, $\delta T = 1$)

$$\sum_{n=0}^{10} 1 \times f(n \times 1)$$

Now, intuitively it seems that as $\delta T \to 0$, we get a better estimate of the area: when we get to an infinitesimal δT, we obtain the integral.

[1]This is a Riemann integral. Thankfully this is all most of us will ever need...

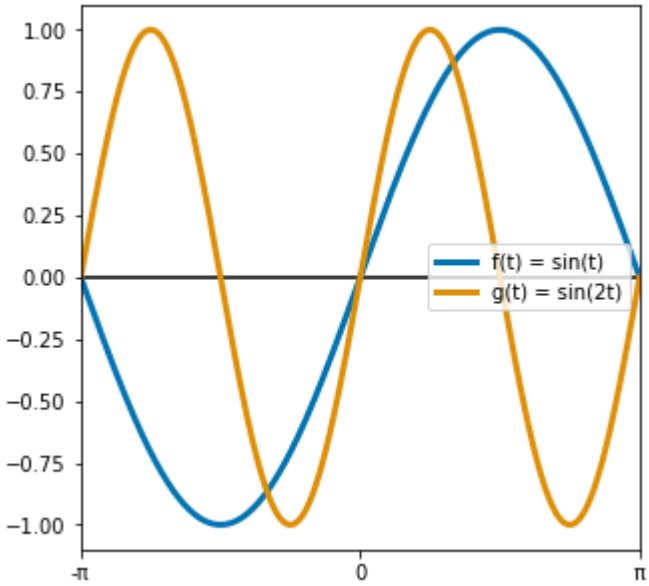
Sine of the times?

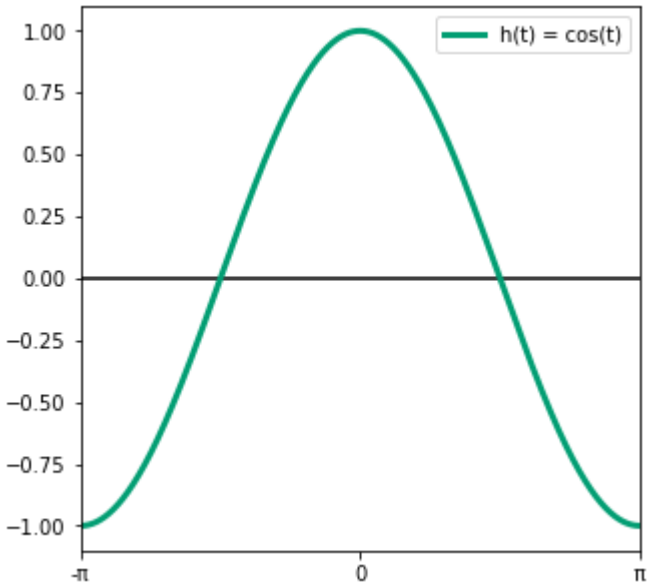
$$f(t) = sin(t)$$
$$g(t) = sin(2t)$$
$$h(t) = cos(t)$$

Sines and cosines are trigonometric functions; the key point from our perspective is that they are periodic. If we take $sin(t)$ and put in values of t between $-\pi$ and π, we will get the same sequence of values as if we put in π to 3π, or 7π to 9π etc etc.

In [4]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(-np.pi,np.pi,2000)
f = np.sin(t)
g = np.cos(t)
h = np.sin(2*t)
plt.plot([-100,100],[0,0],color="black")
plt.plot(t,f,color=colour_palette[0],label="f(t) = sin(t)",lw=
3)
# plt.plot(t,g,color=colour_palette[2],label="g(t) = cos(t)",l
w=3)
plt.plot(t,h,color=colour_palette[1],label="g(t) = sin(2t)",lw
=3)
# plt.xticks([0,5,10])
# plt.yticks([0,1.5,3])
plt.xlim([-np.pi,np.pi])
plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
# plt.ylim([0,3.1])
# plt.grid(True)
plt.legend()

plt.figure()

t = np.linspace(-np.pi,np.pi,2000)
f = np.sin(t)
g = np.cos(t)
h = np.sin(2*t)
plt.plot([-100,100],[0,0],color="black")
# plt.plot(t,f,color=colour_palette[0],label="f(t) = sin(t)",l
w=3)
plt.plot(t,g,color=colour_palette[2],label="h(t) = cos(t)",lw=
3)
# plt.plot(t,h,color=colour_palette[1],label="g(t) = sin(2t)",
lw=3)
# plt.xticks([0,5,10])
# plt.yticks([0,1.5,3])
plt.xlim([-np.pi,np.pi])
plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
# plt.ylim([0,3.1])
# plt.grid(True)
plt.legend()
print()
```



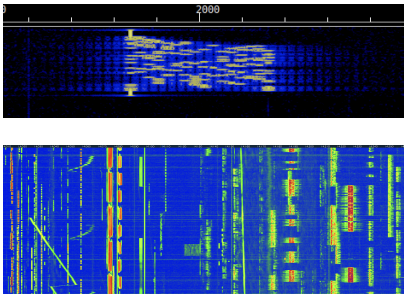


Since the period of $sin(t)$ is 2π, a sine wave at a given frequency k Hz would be $sin(2\pi kt)$.

A sine wave at $f = 100\text{Hz}$ would be $sin(100 \times 2\pi \times t)$ where t is in seconds.

Waterfalls & Heated Plates

- Probably all have heard of the Fourier Transform and seen a waterfall display like this; do you know why it works?



(images from George M1GEO and John M5ET respectively)

- A common goal in computing the Fourier **transform** is to determine how much energy/signal there is at a given frequency. We will fudge our way into this by starting from the basics...

Towards Fourier Space

- May have seen a Maclaurin or Taylor series for approximating a function:

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots$$

- How might we represent a *periodic function*? We might try [1]

$$f(t) \approx c + a_1 sin(t) + b_1 cos(t) + a_2 sin(2t) + b_2 cos(2t) + \dots$$

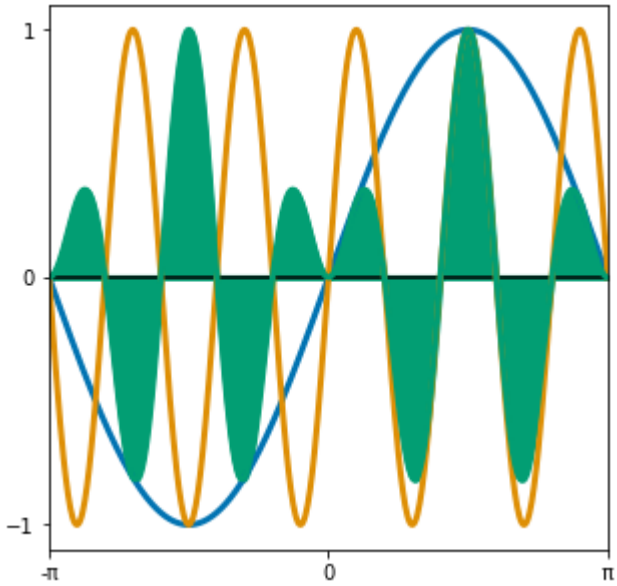
[1] Advanced students should explain to their neighbours in the next break why we need both sine and cosine terms here.

This is a *Fourier Series*. How do we find a_i and b_i? **Orthogonality of sines**

Orthogonality of Sines

In [5]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]


# fig,ax=plt.subplots(1,2)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
#     m = m_slider.val
    plt.cla()
    y_1 = np.sin(x)
    plt.plot(x,y_1,color=colour_palette[0],lw=3.0)
    plt.plot([-100,100],[0,0],color='black')
    y_2 = np.sin(m*x)
    plt.plot(x,y_2,color=colour_palette[1],lw=3.0)

    product = y_1 * y_2

    plt.plot(x,product,color=colour_palette[2],lw=3.0)
    plt.fill_between(x,0,product,color=colour_palette[2],lw=3.
0)
    plt.xlim(-np.pi,np.pi)

    plt.yticks([-1,0,1])
    plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
    #plt.title('$sin(x)sin(kx)$',fontsize=30)

#     mpld3.display(fig,closefig=True)

m = 2

# fig,ax = plt.subplots(1,1)
x = np.linspace(-np.pi,np.pi,500)
y_1 = np.sin(x)

update_m_value(5)

#add slider for m



# mpld3.display(fig)
print()
```



If we multiply a sine or a cosine by $sin(nt)$ and integrate the product over a period, the result is zero if they are at different integer frequencies (whole-number frequencies), and π if they are at the same frequency.

$$\int_{-\pi}^{\pi} sin(nt)sin(mt)dt = \begin{cases} \pi & \forall\, n = m : n \in \mathbb{Z} \\ 0 & n \neq m \end{cases}$$

(the same is true for cos, I promise!)

Integrals between sine and cosine are always zero over a period:

$$\int_{-\pi}^{\pi} sin(nx)cos(mx)dx = 0\ \forall n, m \in \mathbb{Z}$$

- In essence this is a **filter**: we take a function which we assume to be composed of sines and cosines, and we can measure how much of each sine and cosine is in the signal. Because each sine and cosine is at a discrete frequency, this lets us measure the

- Why does this help us?

$$f(t) \approx c + a_1 sin(t) + b_1 cos(t) + a_2 sin(2t) + b_2 cos(2t) + \dots$$

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(nt)dt$$
$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)cos(nt)dt$$

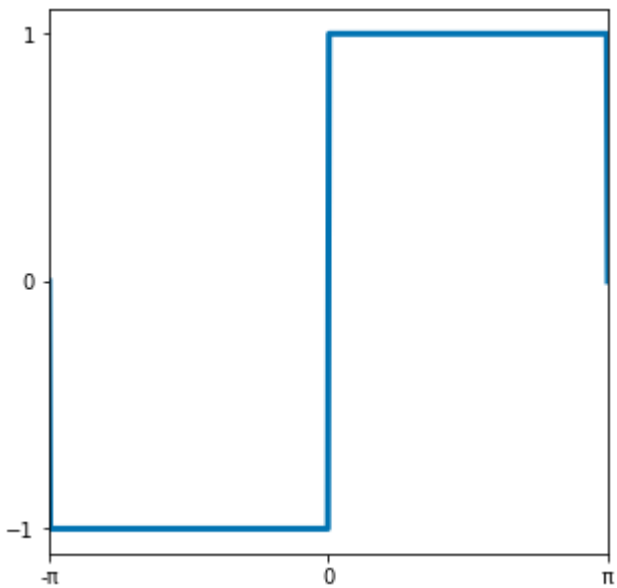
(plus a DC offset...)

$$c = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)dt$$

An example: the square wave

- This is a carefully chosen example: why?
- Formally: $f(t) = \begin{cases} -1 & -\pi < t < 0 \\ 1 & 0 \leq t < \pi \end{cases}$

In [6]:
```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
plt.rcParams['figure.figsize']=[5,5]

colour_palette = sns.color_palette("colorblind",3)

x = np.linspace(-np.pi,np.pi,500)

plt.xlim(-np.pi,np.pi)

plt.yticks([-1,0,1])
plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
y_1 = np.where(x<0,-1.0,1.0)
y_1[0] = 0
y_1[-1]=0
plt.plot(x,y_1,color=colour_palette[0],lw=3.0)
print()
```



- Sine series only: there is **no** DC offset and **no** cosine term. (those advanced students can discuss this later too)

Calculating the coefficients

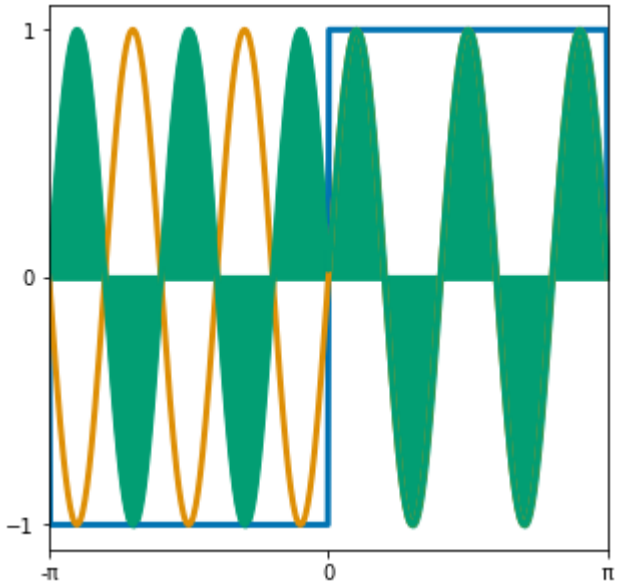
$$a_1 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(t)dt$$

$$= \frac{1}{\pi}\left[\int_{-\pi}^{0} -1sin(t)dt + \int_{0}^{\pi} 1sin(t)dt\right] = \frac{4}{\pi}$$

In [7]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]




fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
#     m = m_slider.val
    plt.cla()
    y_1 = np.where(x<0,-1.0,1.0)
    y_1[0] = 0
    y_1[-1]=0
    plt.plot(x,y_1,color=colour_palette[0],lw=3.0)
#     plt.plot([-100,100],[0,0],color='black')
    y_2 = np.sin(m*x)
    plt.plot(x,y_2,color=colour_palette[1],lw=3.0)

    product = y_1 * y_2

    plt.plot(x,product,color=colour_palette[2],lw=3.0)
    plt.fill_between(x,0,product,color=colour_palette[2],lw=3.
0)
    plt.xlim(-np.pi,np.pi)

    plt.yticks([-1,0,1])
    plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
    #plt.title('$sin(x)sin(kx)$',fontsize=30)


m = 2

# fig,ax = plt.subplots(1,1)
x = np.linspace(-np.pi,np.pi,500)
y_1 = np.sin(x)

update_m_value(5)

#add slider for m
```

```
# mpld3.display(fig)
print()
```



$$a_2 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(2t)dt$$
$$= \frac{1}{\pi}\left[\int_{-\pi}^{0} -1sin(2t)dt + \int_{0}^{\pi} 1sin(2t)dt\right] = 0$$

$$a_3 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(3t)dt$$
$$= \frac{1}{\pi}\left[\int_{-\pi}^{0} -1sin(3t)dt + \int_{0}^{\pi} 1sin(3t)dt\right] = \frac{4}{3\pi}$$

(...many hours later...)

Instead of working out each term, we could find a general expression: (an identity that makes it much easier!)

$$\int_a^b sin(nt)dt = -\frac{1}{n}[cos(nt)]_a^b$$

$$cos(n\pi) = \begin{cases} 1 & \text{even } n \\ -1 & \text{odd } n \end{cases}$$

Skipping some algebra (if you enjoy that sort of thing, fair enough...), it turns out that:

$$a_n = \begin{cases} 0 & \text{even n} \\ \frac{4}{n\pi} & \text{odd n} \end{cases}$$

So the Fourier Series for the square wave is

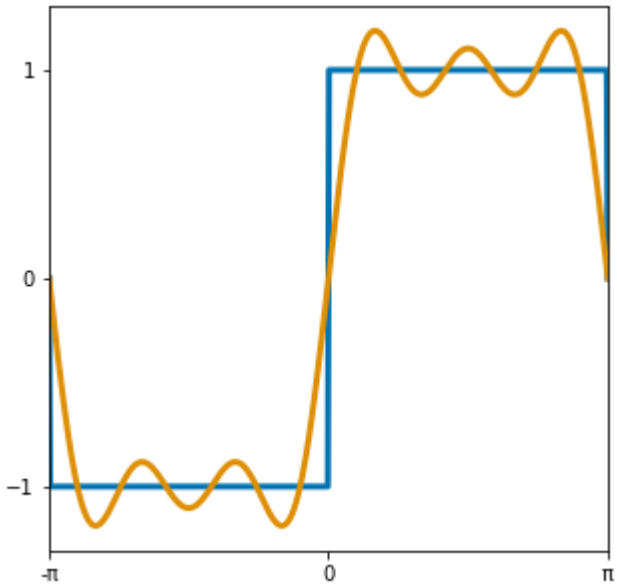
$$f(t) = \sum_{\text{All odd n}} \frac{4}{n\pi} sin(nt)$$

The following graphic illustrates this for a finite number m of sines...

In [8]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]


colour_palette = sns.color_palette("colorblind",3)


fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
#     m = m_slider.val
    plt.cla()
    y_1 = np.where(x<0,-1.0,1.0)
    y_1[0] = 0
    y_1[-1]=0
    plt.plot(x,y_1,color=colour_palette[0],lw=3.0)
#     plt.plot([-100,100],[0,0],color='black')
    bases = np.linspace(1,m,m)

    sin_coefficients = np.where(bases%2==0,0,4 /(bases*np.pi))
    sinusoids = np.sin(x*np.reshape(bases,(-1,1)))
    y_2 = np.sum(np.reshape(sin_coefficients,(-1,1))*sinusoids,a
xis=0)
    plt.plot(x,y_2,color=colour_palette[1],lw=3.0)

    plt.xlim(-np.pi,np.pi)

    plt.yticks([-1,0,1])
    plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
    #plt.title('$sin(x)sin(kx)$',fontsize=30)


m = 2

# fig,ax = plt.subplots(1,1)
x = np.linspace(-np.pi,np.pi,500)
y_1 = np.sin(x)

update_m_value(5)



# mpld3.display(fig)
```

```
print()
```



The ringing around the transitions in this graphic is referred to as the Gibbs Phenomenon.

- In practice, Fourier techniques use complex exponentials combining sine and cosine in one term. Easier to integrate and more compact: $e^{i\theta} = cos(\theta) + isin(\theta)$ (Euler's Formula). This necessitates the use of complex numbers: no explanation will be given here, but when you see $e^{i\theta}$ in the following slides, consider that it is simply a convenient way of bundling the sine and cosine terms into one term to make life easier.
- For discrete signals, we pretend the signal is periodic; calculating the Fourier series is then referred to as the "discrete Fourier transform".

Adding Complexity

(or complex numbers, anyway)

You may recall that you can't square-root a negative number: after all, $(-2)^2 = +4$, so it's obvious that nothing square-roots to a negative number...right? For mathematical convenience, an additional *field* of numbers is defined called the "complex numbers" where this is possible. A complex number is written $a + bi$, where:

$$i = \sqrt{-1}$$
$$i^2 = -1$$

Conveniently, this allows the use of Euler's Formula:

$$e^{i\theta} = cos(\theta) + isin(\theta)$$

which combines both the sine and cosine terms in such a way that we can do half the work in finding Fourier coefficients!

We might *intuitively* suspect that, if $e^{i\theta}$ is a mixture of sines and cosines, we can represent our old friend the Fourier Series in this fashion. Sure enough, (without proof),

$$f(t) = \sum_{n=0}^{\infty} a_n e^{int}$$
$$a_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-int} dt$$

N.B. advanced students might wonder why we can bundle the coefficients into one here when earlier we had to separate the sine and cosine terms: remember that the coefficients are now complex as well.

Arbitrary Periods

- Why limit ourselves to integrating over the interval $[-\pi, \pi]$?
- Use simple algebra to make the period T instead of 2π...
- In short, make t go through a range of 2π radians in T seconds: multiply by $\frac{2\pi}{T}$ and iron out the constants!

$$f(t) = \sum_{n=0}^{\infty} a_n e^{in \times \frac{2\pi t}{T}}$$
$$a_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-in \times \frac{2\pi t}{T}} dt$$

- This is **all** we need to do before we can move to the Fourier **transform**!

Fourier Transform

Problem

- We want to find the frequency content of an aperiodic function
- We do not expect the function to be representable by a sum of discrete frequency trig functions/complex exponentials...
- We will define the time function $f(t)$ and its corresponding Fourier transform $\tilde{F}(k)$ (where k is the frequency in units of $[\frac{1}{[t]}]$)
- To represent the operation of taking the Fourier transform, we will write
$$\tilde{F}(k) = \mathcal{F}\left[f(t)\right]$$
- There also exists an *inverse* Fourier transform:
$$f(t) = \mathcal{F}^{-1}\left[\tilde{F}(k)\right]$$

The algebra in this slide is fiddly: ignore the details unless you find them obvious and work them out later. Focus on the concepts.

Begin with the Fourier Series for a function of period T.

$$f(t) = \sum_{n=0}^{\infty} a_n e^{int \times \frac{2\pi}{T}}$$
$$a_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t) e^{-int \times \frac{2\pi}{T}}\, dt$$

- Define a new variable $k = \frac{n}{T}$.
- It follows that $n = kT$.

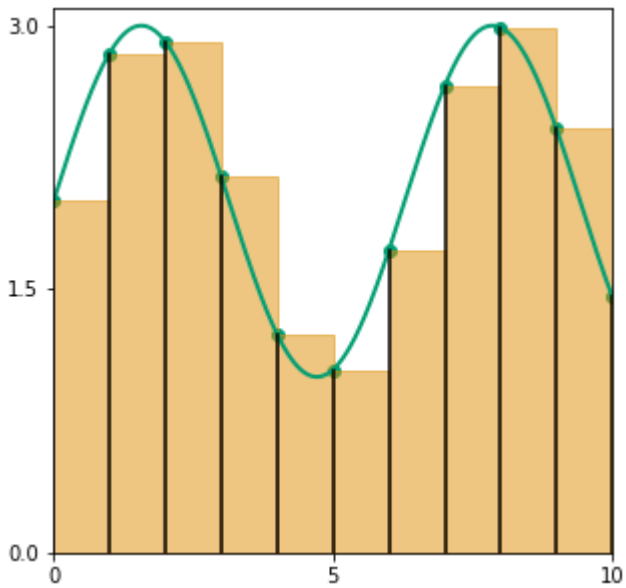
(there is no key concept in this - just a constant relationship defined for algebraic convenience)

- Replace a_n with some function that yields the same coefficients: let's call it $\tilde{F}(k)$.
- The sum incremented n by 1 each pass. Incrementing k by 1 is equivalent to incrementing n by T, so we write $a_n = \tilde{F}(k)\delta k$. For equivalence, $\delta k = \frac{1}{T}$

We're subtly preparing for a move from discrete 'frequencies' n to continuous values k. For now, we 'discretise' the continuous value by multiplying by δk and fiddle with the algebra so that it is exactly equivalent to the Fourier series. Remember the Riemann integral earlier?

In [9]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,10,1000)
s = np.sin(t)+2
plt.plot(t,s,lw=2,color=colour_palette[2],linestyle='-')
t_points = np.linspace(0,10,11)
s_points = np.sin(t_points)+2
plt.scatter(t_points,s_points,color=colour_palette[2])
for (x,y) in zip(t_points,s_points):
    plt.plot([x,x],[0,y],color='black')
    plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_pal
ette[1])



plt.xticks([0,5,10])
plt.yticks([0,1.5,3])
plt.xlim([0,10])
plt.ylim([0,3.1])
# plt.grid(True)
print()
```



$$f(t) = \sum_{k=0}^{\infty} \tilde{F}(k)\delta k \; e^{it \times kT \times \frac{2\pi}{T}}$$
$$= \sum_{k=0}^{\infty} \tilde{F}(k)\delta k \; e^{2\pi itk}$$

- Remember that $\delta k = \frac{1}{T}$.
- What about an aperiodic function: $T \to \infty$??
- We can take the *limit* of this, as $\delta k \to 0$.

Concept: as the function becomes aperiodic, the increment in frequency becomes infinitessimal: we end up with a ***continuous frequency spectrum****.*

$$f(t) = \lim_{\delta k \to 0} \left(\sum_{k=0}^{\infty} \tilde{F}(k)\delta k e^{2\pi itk} \right)$$

- This is just like the sum converging to an integral from earlier!

$$f(t) = \int_{-\infty}^{\infty} \tilde{F}(k) e^{2\pi itk} dk$$

This is the inverse Fourier Transform: what we've just written is equivalent to $f(t) = \mathcal{F}^{-1}(\tilde{F}(k))$

Fourier, finally!

Forward

$$\tilde{F}(k) = \int_{-\infty}^{\infty} f(t) e^{-2\pi itk} dt$$

Inverse

$$f(t) = \int_{-\infty}^{\infty} \tilde{F}(k) e^{2\pi itk} dk$$

But there's always a catch.

- Can we integrate over infinite time? Rather an expensive operation...
- What about sampled signals? Then we need a sum again! In fact, the "discrete time Fourier transform" is really just a Fourier series, where we pretend the sampled signal is periodic...
- No time to look at the detail of the solutions.

Just a moment:

What does the "frequency domain"/"time domain" mean?

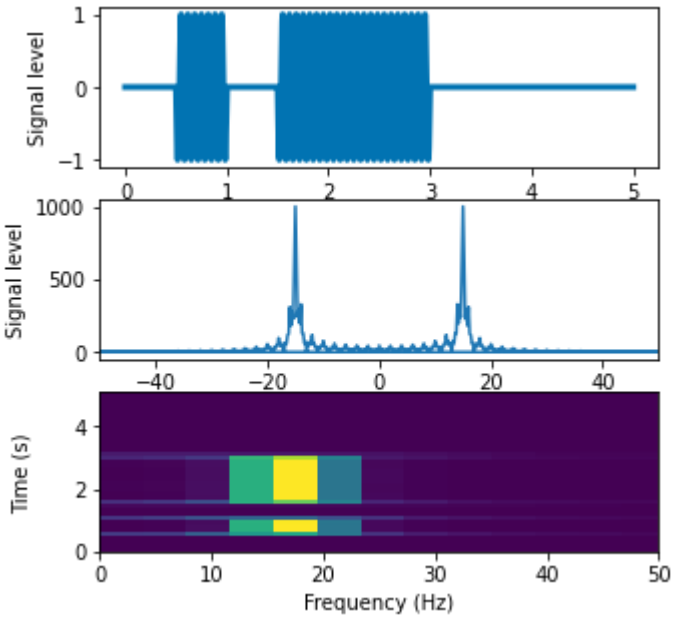
- I said earlier that you should be familiar with the waterfall display on a radio and that this was from the Fourier transform: this was not entirely transparent.
- The Fourier Transform provides a direct mapping between the frequency and time domains.
- Both the spectrum $\tilde{F}(k)$ and the time domain signal $f(t)$ represent a signal completely.
- The Fourier transform is **not quite** what you see on a waterfall: it covers the **whole** signal.
- Waterfall is generated by the STFT: Short Time Fourier Transform. Basically, slide a window over the time-domain signal and take the Fourier transform of the data in each window to form a row of the waterfall.

An example with a Morse letter A, at a carrier frequency of 15Hz

- The Fourier transform produces signed (positive and negative) frequencies; for our applications here these are the same.
- We turn the sine wave carrier for the Morse character on and off instantly; this is bad practice because it generates lots of harmonics. You can see these in the Fourier transform and even in the waterfall: around each edge, the harmonics generated radiate outwards across the 'band'.

In [10]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns


import scipy
import scipy.signal
plt.rcParams['figure.figsize']=[5,5]
fig,(ax1,ax2,ax3) = plt.subplots(3,1)
colour_palette = sns.color_palette("colorblind",3)
plt.subplots_adjust(hspace=0.2)
x = np.linspace(0,5,5000)
y = [0]*len(x)
y = np.sin(15*np.pi*2*x)
y = np.where(((x>1.5) & (x<3.0)) | ((x>0.5) & (x<1.0)),y,0)

# h = scipy.signal.firwin(50,55,fs=1000)
# y=np.convolve(y,h)
ax1.plot(x,y[0:len(x)],color=colour_palette[0],lw=3.0)
ax1.set_xlabel("Time (s)")
ax1.set_ylabel("Signal level")
ax2.plot(np.fft.fftfreq(len(y),0.001),np.abs(np.fft.fft(y)))
ax2.set_ylabel("Signal level")
ax2.set_xlim(-50,50)
ax2.set_xlabel("Frequency (Hz)")

f,t,zxx = scipy.signal.stft(y,fs=1000)
ax3.pcolormesh(f,t, np.abs(zxx.transpose()))
ax3.set_ylabel("Time (s)",labelpad=20)
ax3.set_xlabel("Frequency (Hz)")
ax3.set_xlim(0,50)
print()
```

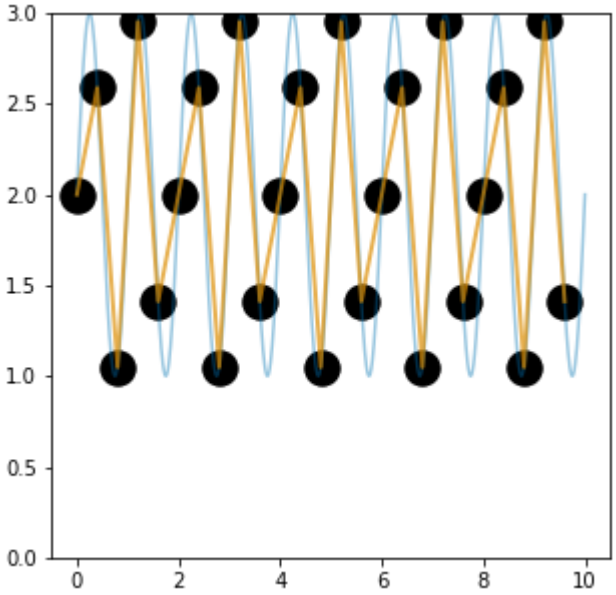


Sampling Theorem

- Consider a continuous signal $f(t)$ such that the bandwidth of the signal is W
- We want to obtain a digital representation by sampling it at discrete intervals, say every T seconds.
- The snappily titled "Whittaker–Nyquist–Kotelnikov–Shannon Sampling Theorem" ("The Sampling Theorem", as Shannon calls it) gives us glad tidings: we don't lose anything if we chose $T < \frac{1}{2W}$.
- (or equivalently the **sampling frequency** $f_s > 2W$

In [11]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]


fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
    plt.cla()
    freq = 1*2*np.pi
    offset = 2
    x_samples = np.arange(0,10,(1/m))
    x = np.linspace(0,10,1000)
    plt.plot(x,np.sin(freq*x)+offset,color=colour_palette[0],al
pha=0.4,linestyle="-")
    plt.scatter(x_samples,np.sin(freq*x_samples)+offset,color="
black",s=300) #weight?
    plt.ylim([0,3])
    plt.plot(x_samples,np.sin(freq*x_samples)+offset,color=colo
ur_palette[1],alpha=0.7,lw=2)
#     for xi in x_samples:
#         plt.plot([xi,xi],[0,np.sin(freq*xi)+offset],color="bl
ack")


m = 2

# fig,ax = plt.subplots(1,1)
x = np.linspace(-np.pi,np.pi,500)
y_1 = np.sin(x)

update_m_value(2.5)

#add slider for m

    # mpld3.display(fig)
print()
```



Proof

- We can prove this using the theory of Fourier series and transforms we've just worked out.
- Recall that a periodic function is described completely by the coefficients of its Fourier series...

$$f(t) = \sum_{n=0}^{\infty} a_n e^{int \times \frac{2\pi}{T}}$$

- Recall that the inverse Fourier Transform lets us find the time domain function given the Fourier transformed value...

$$f(t) = \mathcal{F}^{-1}[\tilde{F}(k)]$$

$$f(t) = \int_{-\infty}^{\infty} \tilde{F}(k) e^{2\pi i t k} dk$$

The time-domain function is given by the inverse Fourier transform of the frequency-domain function.

$$= \int_{-W}^{W} \tilde{F}(k) e^{2\pi i t k} dk$$

The signal is band-limited to frequencies between 0 and W units; as there's nothing outside this interval, we can change the limits with no ill effect. Note that frequency as commonly referred to is always positive, but in this context is signed: we integrate from $-W$ to W.

- Let us *sample* at $t = \frac{n}{2W}$ where n is the sample index.
- (i.e. sampling interval = $\frac{1}{2W}$, so sampling frequency = $2W$).

$$f(\frac{n}{2W}) = \int_{-W}^{W} \tilde{F}(k) e^{2\pi i k \frac{n}{2W}} \, dk$$

Now Step Back

- Imagine we want to find the Fourier **series** of $\tilde{F}(k)$ (odd thing to do??)

$\tilde{F}(k)$ *is a function just like any other; if it is periodic with period 2W, we can represent it by a Fourier series just like earlier.*

$$\tilde{F}(k) = \sum_{n=0}^{\infty} a_n e^{in \times \frac{2\pi k}{2W}}$$

$$a_n = \frac{1}{2W} \int_{-W}^{W} \tilde{F}(k) e^{-in \times \frac{2\pi k}{2W}} \, dk$$

That looks familiar.

We find the coefficients are

$$a_n = \frac{1}{2W} \int_{-W}^{W} \tilde{F}(k) e^{-in \times \frac{2\pi k}{2W}} \, dk$$

But imagine we take the earlier expression for the inverse Fourier transform of $\tilde{F}(k)$ and "sample" the signal by measuring at times $t = \frac{n}{2W}$ (such that n is a sample index). In other words, we measure the signal at these moments; throw in a minus sign for mathematical convenience (such that the sample numbering runs 'backwards' - it is arbitrary so who cares) and we end up with this:

$$f\left(\frac{-n}{2W}\right) = \int_{-W}^{W} \tilde{F}(k) e^{-2\pi i k \frac{n}{2W}} \, dk$$

In words, the sampled time-domain function exactly defines the Fourier series of the frequency spectrum:

$$f\left(\frac{-n}{2W}\right) = a_n$$

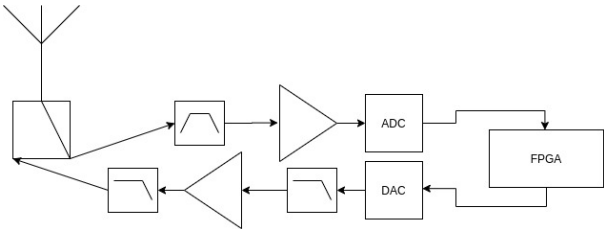
*It turns out that the samples at $t = \frac{n}{2W}$ are equal to the coefficients of a Fourier Series for $\tilde{F}(k)$: we can reconstruct the spectrum from this Fourier series, and since the signal is band-limited the spectrum is effectively periodic; we know that we can recover the full time-domain signal (including at times we **did not measure it**) by taking the inverse Fourier transform of the spectrum, $f(t) = \mathcal{F}^{-1}(\tilde{F}(k))$, so we're done!*

This proof is given in Claude Shannon's (a titan of information theory) famous paper "Communication in the Presence of Noise", section II: see https://web.archive.org/web/20100208112344/http://www.stanford.edu/class/ee104/shannonpaper.pdf (https://web.archive.org/web/20100208112344/http://www.stanford.edu/class/ee104/shannonpaper.pdf)

Anti-Aliasing & Reconstruction Filters

- The Sampling Theorem suggests two constraints: one at the ADC, one at the DAC.
- ADC input signal must be band-limited: an "anti-aliasing" filter is required on the input.
  - Filters aren't perfect: need wider bandwidth to allow for rolloff.
  - N.B. (advanced note) The criterion is that the **bandwidth** is limited, not that the **frequency** is limited: oversampling.
- DAC output signal must be band-limited: a "reconstruction filter" is used.

You see this in a typical software-defined radio architecture: there is a low-pass or band-pass filter before the ADC, and a low-pass filter after the DAC.



Recap

- An integral is the area under a curve between two points, $\int_a^b f(t)dt$
- The area under the product of two sinusoids at integer frequencies is zero unless they are at the same frequency (integrating over an integer number of periods).
- Thus we can measure the "amount" of a function at a given frequency.
- From this we derived the idea of a Fourier series: represent a periodic signal as a sum of content at the harmonic frequencies.
- We modified the Fourier Series into a transform that works for aperiodic signals by letting the period tend to infinity and tweaking the form of the coefficients into a "spectrum".
- Given the tools of the Fourier Series & the Fourier Transform, we proved that sampling a signal "at an adequate sampling frequency" loses no information.
- Now satisfied that our sampled signal is a good representation, what can we do with it?
- Filter the digital signal to a bandwidth of W... if listening to a CW signal and wanting to filter out an adjacent interferer, a digital filter can be trivially adjusted: an electronic filter might require a new crystal or a variable capacitor.

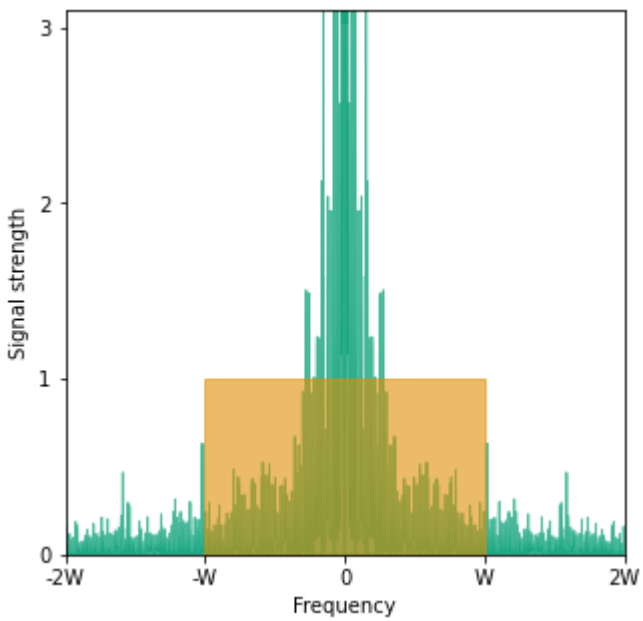
Hitting a Brick Wall

- Ideally we would:
  - exclude all frequencies outside $(-W, W)$
  - have gain 1 for frequencies in $(-W, W)$
- This is referred to as a "brick wall filter": the frequency domain representation of the filter is below in orange, while an example spectrum is plotted in green.
- Practical filters can't do this.

We will now take a detour to discover a common mechanism used for digital filters...

In [12]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,20,1000)
s = np.random.pareto( 40,1000)*30*(1/t)

plt.fill_between(t,0,s,alpha=0.7,color=colour_palette[2])
plt.fill_between(-t,0,s,alpha=0.7,color=colour_palette[2])

plt.fill_between([-10,10],0,[1,1],alpha=0.6,color=colour_palett
e[1])

# plt.plot([x,x],[0,y],color='black')
#plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_palett
e[1])

plt.xticks([-20,-10,0,10,20],["-2W","-W","0","W","2W"])
plt.xlim([-20,20])
plt.yticks([0,1,2,3])
plt.ylim([0,3.1])
plt.xlabel("Frequency")
plt.ylabel("Signal strength")
# plt.grid(True)
print()
```



Acting on Impulse

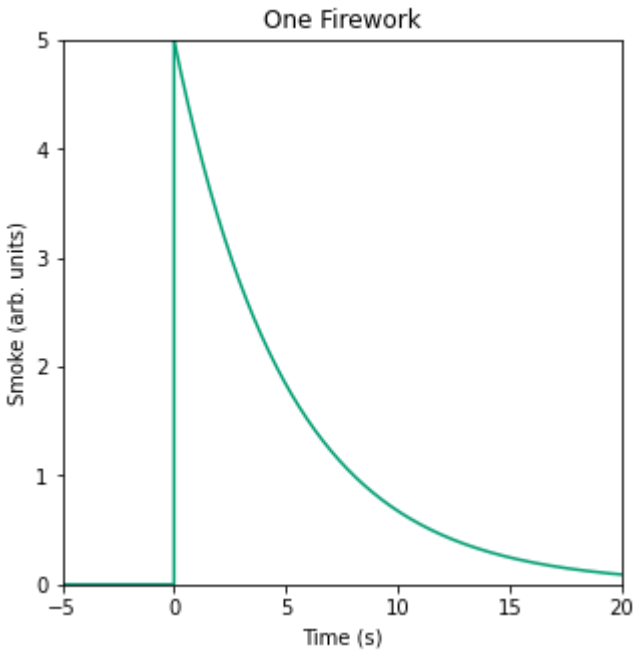
How Much Smoke?

(explanation due with thanks and deference to Prof. Tom Hynes).

- Determine how much smoke is in the air at a given moment during a firework display.
- Measure one firework...

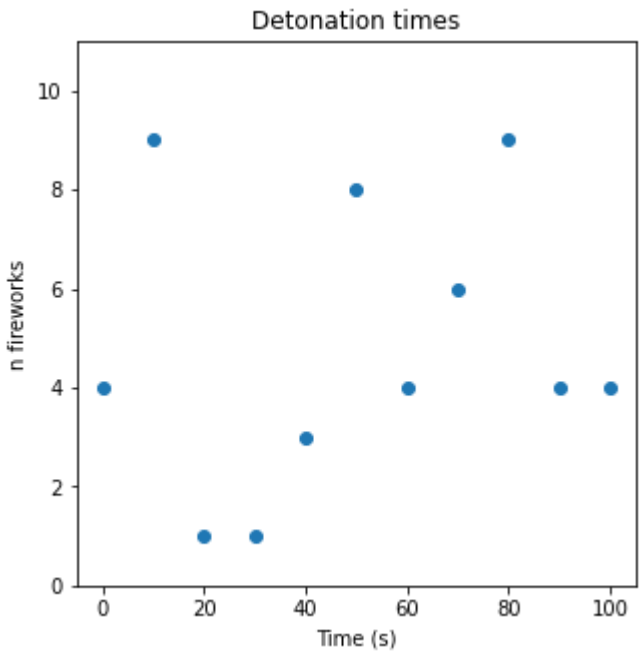
In [13]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,20,2000)
s = 5*np.exp(-0.2*t)
plt.plot(t,s,color=colour_palette[2])
plt.plot([-5,0,0.001],[0,0,5],color=colour_palette[2])
#plt.plot([x,x],[0,y],color='black')
#plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_palett
e[1])
plt.ylabel("Smoke (arb. units)")
plt.xlabel("Time (s)")
plt.xticks([-5,-0,5,10,15,20])
plt.xlim([-5,20])
plt.ylim([0,5])
plt.title("One Firework")
# plt.grid(True)
print()
```



In [14]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]
N_POINTS = 11
counts = np.around(np.random.rand(N_POINTS)*10)
times = np.linspace(0,100,N_POINTS)

plt.scatter(times,counts)

#plt.plot([x,x],[0,y],color='black')
#plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_palett
e[1])
plt.ylabel("n fireworks")
plt.xlabel("Time (s)")
# plt.xticks([-5,-0,5,10,15,20])
# plt.xlim([-5,20])
plt.title("Detonation times")
plt.ylim([0,11])
# plt.grid(True)
print()
```



Impulse Responses and Convolution

- The amount of smoke produced by one firework is termed the "impulse response": here the impulse is the detonation of a firework.
- We define an impulse function $\delta(t)$ (a Dirac delta function)

$$\int_{-\infty}^{\infty} f(t)\delta(t)dt = f(0)$$

- It "sifts out" the value at $t = 0$.
- Note: $\delta(t - T)$ 'fires' at $t = T$ (i.e. the *argument* is zero)
- What a system does after being excited with $\delta(t)$ is the *impulse response* $h(t)$

Impulse Responses and Convolution

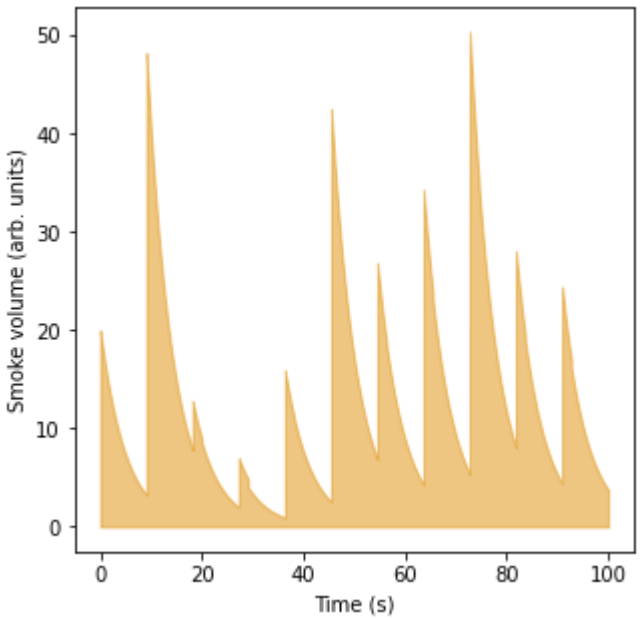
- Here, the impulse is delivered by detonation of the firework; the smoke decays away over time afterwards.
- We calculate the total volume of smoke $f(t)$ by summing up all the smoke left in the air from previous firework detonations
- Let us say the fireworks go off at times $t = 1, 2, 3...$ in quantity $g(t)$.
- Don't be confused: two variables, τ and t!!! Questionable choice but invariably what are used; take care not to confuse them. τ (tau) is just a Greek t...

Based on https://commons.wikimedia.org/wiki/File:Flag_of_Greece.png by user Chantel

$$f(t) = \sum_{\tau=0} [h(t - \tau)g(\tau)]$$

- "play back" h(t) each time a firework goes off, and sum up the smoke from **all** the fireworks that have gone off to date

In [15]:
```
SPACED_POINTS = 10000
full_timebase = np.linspace(0,100,SPACED_POINTS)
plt.figure()
padded_detonations = np.zeros(SPACED_POINTS)
for time,val in enumerate(counts):
    padded_detonations[int(time*SPACED_POINTS/N_POINTS)] = val

t = np.linspace(0,20,2000)
s = 5*np.exp(-0.2*t)
smoke = np.convolve(padded_detonations,s)
# plt.plot(full_timebase,padded_detonations,colour_palette[0])
plt.fill_between(full_timebase,0,smoke[0:SPACED_POINTS],color=c
olour_palette[1],alpha=0.5)
plt.ylabel("Smoke volume (arb. units)")
plt.xlabel("Time (s)")
print()
```



$$f(13) = g(0) \times h(13) + g(10) \times h(3) + g(20) \times h(-7) + \ldots$$

...wait...$h(-7)$?

- Causal vs non-causal impulse responses. A "non-causal" impulse response is often encountered in the context of filtering: it's annoying, because we can't implement it in real time. An example of a theoretical system with a non-causal impulse response is a bell which begins to ring 5 seconds **before** you hit it: from a physics perspective it is obvious that the ringing is not causally related to the hammer blow, hence the name.

Viewers Shocked as Presenter turns Sum Into Integral AGAIN!

$$h(t) * g(t) = \int_{-\infty}^{\infty} h(t-\tau)g(\tau)d\tau$$

We can convolve any two functions $h(t)$ and $g(t)$; if we define $h(t)$ to be the impulse response of the system, $g(t)$ is the "stimulus" or driving function.

- $g(t)$ can be **anything** (well almost)
- Not all systems can be modelled like this: examples?

- Several:
  - Magnetics
  - Diodes (sometimes)
  - Transistors (sometimes)
  - Wires????
- System must be linear (so adding up works) and time-invariant (so $h(t)$ is constant). (i.e. if the wind changes between measuring our single firework and firing the whole display, obviously the smoke may decay in a different fashion).

Final Useful Property of Convolution

"The Convolution Theorem"

$$\mathcal{F}\left[f(t) * g(t)\right] = \mathcal{F}[f(t)] \times \mathcal{F}[g(t)]$$

- The Fourier transform of the convolution of two functions is equal to the **product** of the Fourier transform of each of the functions

Convolution in the time domain is multiplication in the frequency domain and vice versa

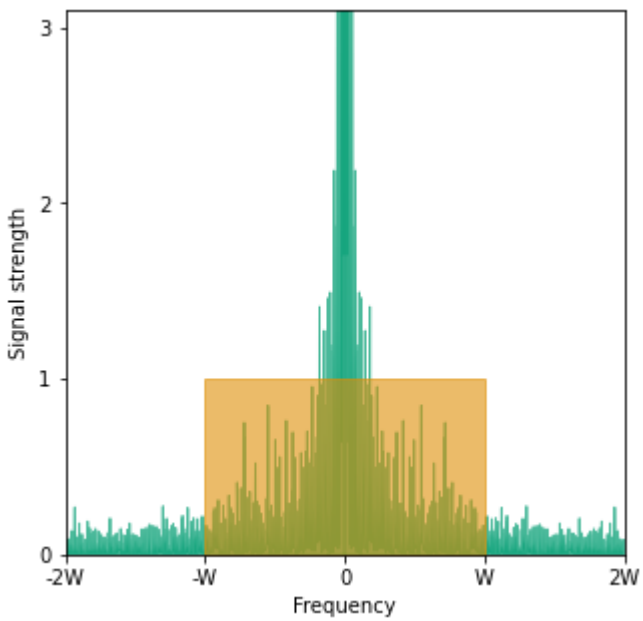
Why Bother: back to digital filtering

- Recall our sampled signal, measured at a series of points. How might we recover intermediate values?
- **Filtering**
- We want to **remove** all signal at frequencies $f : |f| \geq W$
- i.e. multiply by

$$\tilde{H}(k) = \begin{cases} 0 & |k| \geq W \\ 1 & |k| < W \end{cases}$$

In [16]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(0,20,1000)
s = np.random.pareto( 40,1000)*30*(1/t)

plt.fill_between(t,0,s,alpha=0.7,color=colour_palette[2])
plt.fill_between(-t,0,s,alpha=0.7,color=colour_palette[2])

plt.fill_between([-10,10],0,[1,1],alpha=0.6,color=colour_palett
e[1])

# plt.plot([x,x],[0,y],color='black')
#plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_palett
e[1])

plt.xticks([-20,-10,0,10,20],["-2W","-W","0","W","2W"])
plt.xlim([-20,20])
plt.yticks([0,1,2,3])
plt.ylim([0,3.1])
plt.xlabel("Frequency")
plt.ylabel("Signal strength")
# plt.grid(True)
print()
```



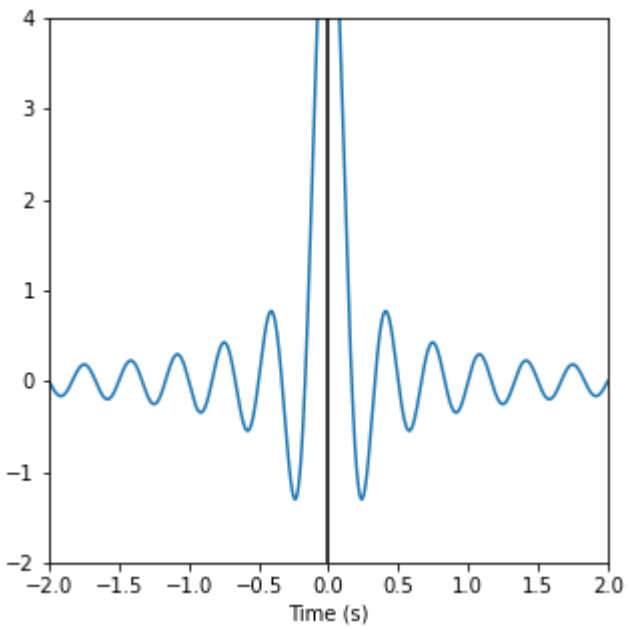
Why not just take the Fourier transform of our signal, chop it off at W, and inverse FT it? Expensive and would require the whole signal...even then, may incur computational error etc.

Prefer to filter using the convolution theorem: instead of multiplying by our "brick wall" in the frequency domain, we can convolve by its impulse response in the time domain.

We can obtain the impulse response of the "brick wall" by taking the inverse Fourier transform thereof. This yields:~

In [17]:

```
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.figure()
plt.rcParams['figure.figsize']=[5,5]

t = np.linspace(-2,2,1000)


# plt.plot([-20,-10.01,-10,10,10.01,20],[0,0,1,1,0,0],color=co
lour_palette[1])
h = np.sin(2 * np.pi * t * 3)/(np.pi * t)
plt.plot(t,h)
plt.xlabel("Time (s)")
plt.xlim(-2,2)
plt.ylim(-2,4)
plt.autoscale(False)
plt.plot([0,0],[-1000,1000],color="black")

#plt.plot([x,x],[0,y],color='black')
#plt.fill_between([x,x+1],0,[y,y],alpha=0.5,color=colour_palett
e[1])

# plt.grid(True)
print()
```



It turns out that the inverse Fourier transform of our "brick wall" is a "sinc function":

$$h(t) = \frac{1}{\pi t} \times sin(2\pi W t)$$

This is the impulse response of a filter with this property! Hurray!

- In order to get the same effect as cutting off the spectrum outside $(-W, W)$, just convolve with $h(t)$.
- A bonus: 'sinc interpolation' (as described by Shannon) allows us to recover the signal at non-sampled moments from a sampled signal (see the Shannon paper linked earlier).

Always a snag

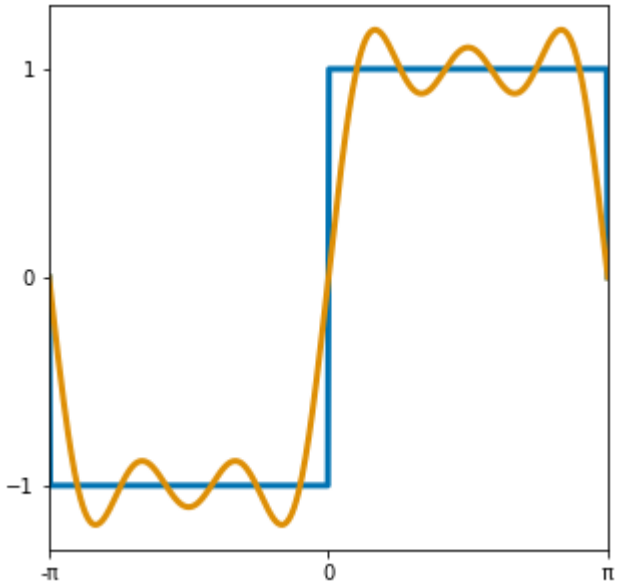
- Non-causal
- Infinite

Practical Solutions

- Shift h(t) so that the peak occurs in the causal part (in other words, delay it), then truncate the impulse response to make it causal and finitely long.
- Truncating leads to some issues...remember the Gibbs phenomenon from earlier, where sharp transitions led to "ringing"?

In [18]:

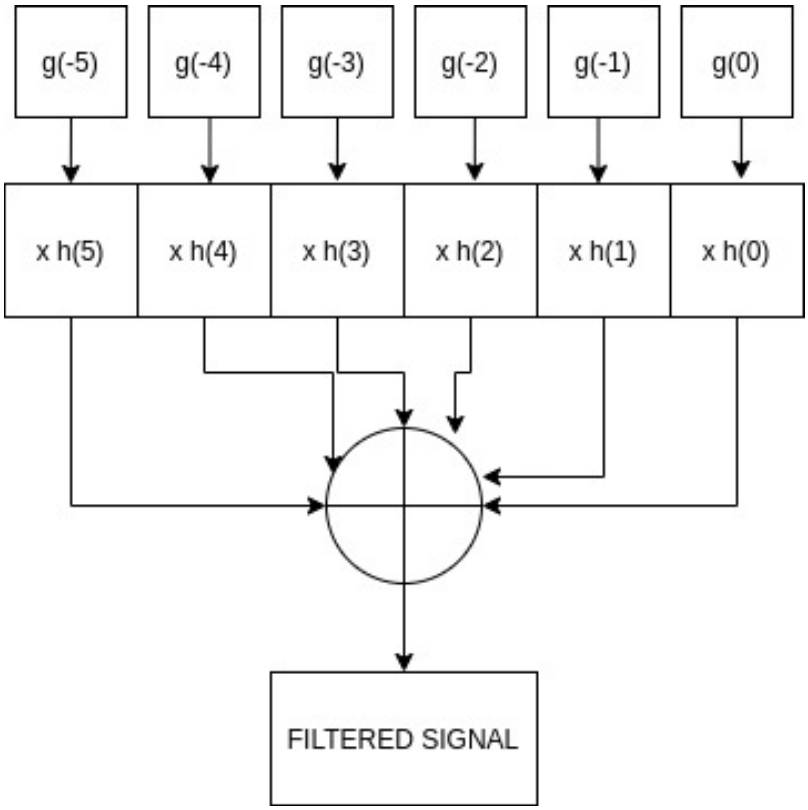
```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]


fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
#     m = m_slider.val
    plt.cla()
    y_1 = np.where(x<0,-1.0,1.0)
    y_1[0] = 0
    y_1[-1]=0
    plt.plot(x,y_1,color=colour_palette[0],lw=3.0)
#     plt.plot([-100,100],[0,0],color='black')
    bases = np.linspace(1,m,m)

    sin_coefficients = np.where(bases%2==0,0,4 /(bases*np.pi))
    sinusoids = np.sin(x*np.reshape(bases,(-1,1)))
    y_2 = np.sum(np.reshape(sin_coefficients,(-1,1))*sinusoids,a
xis=0)
    plt.plot(x,y_2,color=colour_palette[1],lw=3.0)

    plt.xlim(-np.pi,np.pi)

    plt.yticks([-1,0,1])
    plt.xticks([-np.pi,0,np.pi],labels=["-π",0,"π"])
    #plt.title('$sin(x)sin(kx)$',fontsize=30)


m = 2

# fig,ax = plt.subplots(1,1)
x = np.linspace(-np.pi,np.pi,500)
y_1 = np.sin(x)

update_m_value(5)

#add slider for m

    # mpld3.display(fig)
print()
```



Digital Filters

- Not a practical talk: tools for playing around include GNURadio, scipy.signal
- Digital filters just apply the above property: use an impulse response in the time domain to process the signal
- FIR: **F**inite **I**mpulse **R**esponse. Simple, but often expensive.
- IIR: **I**nfinite **I**mpulse **R**esponse. Complex and more computationally efficient, but potentially explosive...feedback!

- We've now come full circle: sample a signal, filter it, and generate the spectrum should you so desire
- FIR with "6 taps" (i.e. 6 sample long impulse response): each sample, we multiply this and the 5 previous samples by some weight (the value of the impulse response) and add up the results.

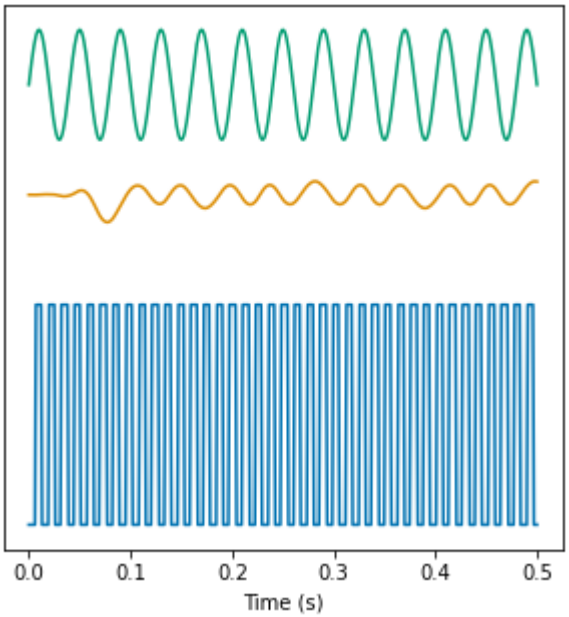


Below, we start with a 25Hz square wave and decide to filter out all but the fundamental, with an FIR filter centred at 30Hz. The number of taps (i.e. the length of the impulse response) is m; as we increase m, the filter becomes better in the frequency sense (sharper cutoff), but we pay for this by doing more computation and accepting a longer delay. Here $m = 150$. The top signal on the plot (green) is at the frequency of the fundamental; the middle signal (orange) is the result of applying the m-tap filter to the bottom signal, which is the square wave.

The second plot shows a frequency-domain view of the same filter.

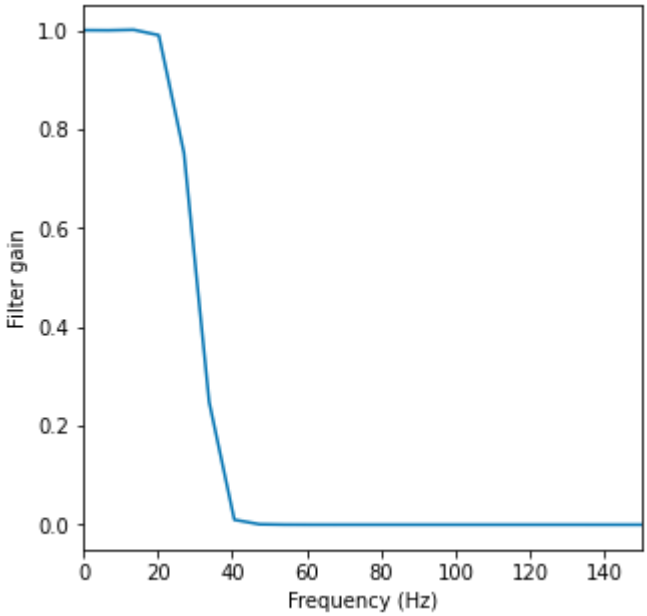
In [19]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]


import scipy
import scipy.signal

colour_palette = sns.color_palette("colorblind",3)
fs = 1000
f_fund = 25
time_end = 0.5

fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
    plt.cla()
    t = np.linspace(0.0,time_end,np.int(fs*time_end))
    y = np.where((np.trunc(t*f_fund*2*np.pi)%2==0),-1,1)*0.2
    h = scipy.signal.firwin(m,30,fs=fs)
    y_filt = np.convolve(y,h) +0.4
    plt.plot(t,y,color=colour_palette[0])
    plt.plot(t,y_filt[:len(t)],color=colour_palette[1])
    plt.yticks([])
    plt.plot(t,np.sin(2*np.pi*f_fund*t)*0.1+0.6,color=colour_p
alette[2])
    plt.xlabel("Time (s)")


m = 2

# fig,ax = plt.subplots(1,1)

update_m_value(150)

#add slider for m


# mpld3.display(fig)
print()
```


0.0
0.1
0.2
0.3
0.4
0.5
Time (s)

In [20]:

```
%reset -f
import warnings
from __future__ import print_function
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import seaborn as sns
colour_palette = sns.color_palette("colorblind",3)
plt.rcParams['figure.figsize']=[5,5]

import scipy
import scipy.signal

colour_palette = sns.color_palette("colorblind",3)
fs = 1000
f_fund = 25
time_end = 0.5

fig,ax=plt.subplots(1,1)
# ax.plot([0,10],[1,1])
# fig.set_size_inches((0.1,0.1),forward=True)

def update_m_value(m):
    plt.cla()
    h = scipy.signal.firwin(m,30,fs=fs)
    plt.plot(np.linspace(0,fs/2,np.int(len(h)/2)),np.abs(np.ff
t.fft(h))[:np.int(len(h)/2)],color=colour_palette[0])
    plt.xlim(0,150)
#     plt.yticks([])
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Filter gain")


m = 2

# fig,ax = plt.subplots(1,1)

update_m_value(150)

# mpld3.display(fig)
print()
```



The End

- Sample a signal: proved we can reproduce it
- Given a signal, we can take the Fourier transform & find out about the frequency content
- LTI systems represented by their impulse response
- Convolution in time domain = multiplication in frequency domain and vice-versa
- Use this to filter a digital signal

Questions to m0+rsgb2020@wje.io

(please try Google first!)