Lecture 2 - Sampling of one-dimensional signals

Outline

- Convolution and correlation
- Sampling
- Undersampling and aliasing
- Nyquist-Shannon theorem

[1]:
```
import numpy as np
import matplotlib.pylab as plt
import scipy.signal as sig
```

Convolution Theorem

The convolution theorem states that the Fourier transform of a convolution of two signals is the pointwise product of their Fourier transforms:

$$\mathcal{F}\{f \otimes g\} = \mathcal{F}\{f\} \odot \mathcal{F}\{g\} \tag{1}$$

where $\mathcal{F}$ denotes the Fourier transform of the functions and $\odot$ is the element-wise product (also known as Hadamard product). In other words, convolution in one domain (e.g. time or spatial domain) equals element-wise multiplication in the other domain (e.g. frequency domain).

We can invert the convolution theorem:

$$\mathcal{F}\{f \odot g\} = \mathcal{F}\{f\} \otimes \mathcal{F}\{g\}\,. \tag{2}$$

This will become useful when we study the sampling of signals.

The Dirac delta function

Definition

The Dirac delta function $\delta(x)$ is zero everywhere except at the origin, where it is infinite. Its integral over the entire real line is equal to 1:

$$\delta(x) = \begin{cases} \infty & x = 0 \\ 0 & x \neq 0 \end{cases} \quad \text{with} \quad \int_{-\infty}^{\infty} \delta(x)\, dx = 1 \tag{3}$$

One way to define the delta function is to regard it as the limit of a sequence of zero-centered rectangular pulses:

$$\delta(x) = \lim_{a \to 0} \text{rect}_a(x) \quad \text{with} \quad \text{rect}_a(x) = \begin{cases} \frac{1}{a} & |x| \leq a/2 \\ 0 & \text{else} \end{cases} \tag{4}$$

Another way to define the delta function is to regard it as the limit of a sequence of zero-centered normal distributions:

$$\delta(x) = \lim_{a \to 0} g_a(x) \quad \text{with} \quad g_a(x) = \frac{1}{a\sqrt{2\pi}} \exp\left\{-\frac{1}{2a^2}x^2\right\} \tag{5}$$

See the gif for an illustration of the latter limit.

Integral with another function

The integral of a Dirac delta with another function $f(x)$ is

$$\begin{aligned}
\int_{-\infty}^{\infty} \delta(x)\, f(x)\, dx &= \int_{-\infty}^{\infty} \lim_{a \to 0} \operatorname{rect}_a(x)\, f(x)\, dx \\
&= \lim_{a \to 0} \int_{-\infty}^{\infty} \operatorname{rect}_a(x)\, f(x)\, dx \\
&= \lim_{a \to 0} \frac{1}{a} \int_{-a/2}^{a/2} f(x)\, dx \\
&= \lim_{a \to 0} \frac{F(a/2) - F(-a/2)}{a} \\
&= f(0)
\end{aligned}$$

where $F(x) = \int_{-\infty}^{x} f(x')\, dx'$ is the antiderivative of $f(x)$.

Therefore:

$$\int_{-\infty}^{\infty} \delta(x)\, f(x)\, dx = f(0)\,. \tag{6}$$

The integral of a shifted δ with another function $f(x)$ results in a shift of $f(x)$:

$$\int_{-\infty}^{\infty} \delta(x-a)\, f(x)\, dx = \int_{-\infty}^{\infty} \delta(x)\, f(x+a)\, dx = f(a) \tag{7}$$

Approximation of the delta function by Fourier Series

Since the delta function is an even function, it can be approximated by a sum of cosines:

$$\delta(x) = a_0 + a_1 \cos(x) + a_2 \cos(2x) + a_3 \cos(3x) + \ldots \tag{8}$$

The coefficients can be calculated as usual (see lecture 11):

$$\begin{aligned}
a_0 &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \delta(x)\, dx = \frac{1}{2\pi} \\
a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} \cos(nx)\delta(x)\, dx \overset{\text{Eq. (6)}}{=} \frac{1}{\pi}\cos(0) = \frac{1}{\pi}
\end{aligned} \tag{9}$$

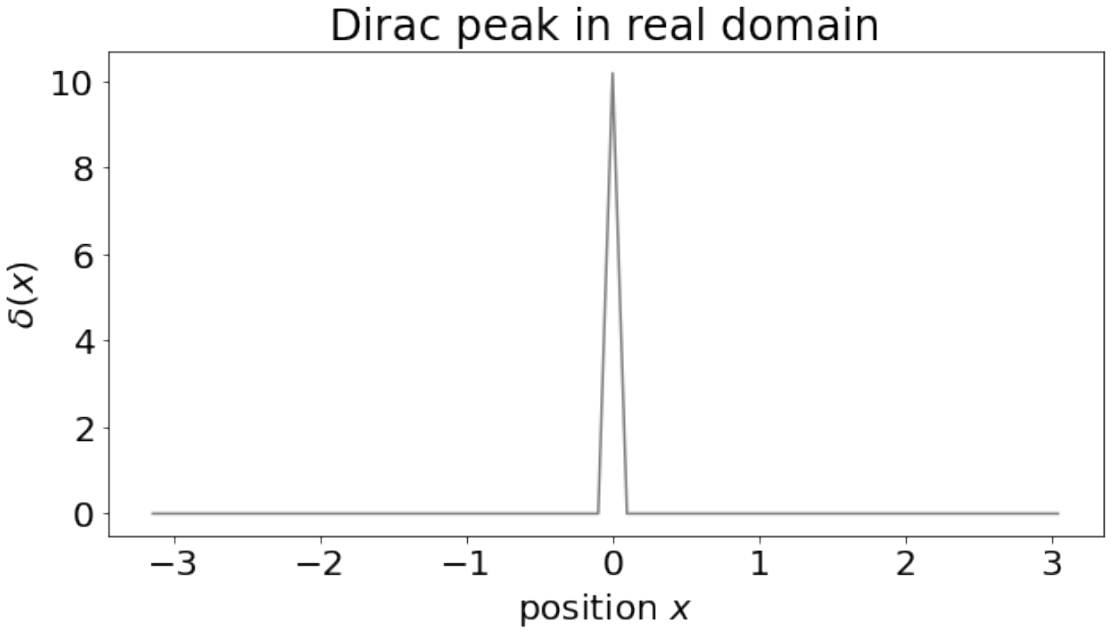
Therefore we can represent $\delta(x)$ as the series

$$\delta(x) = \frac{1}{2\pi} + \frac{1}{\pi}\cos(x) + \frac{1}{\pi}\cos(2x) + \frac{1}{\pi}\cos(3x) + \ldots \tag{10}$$

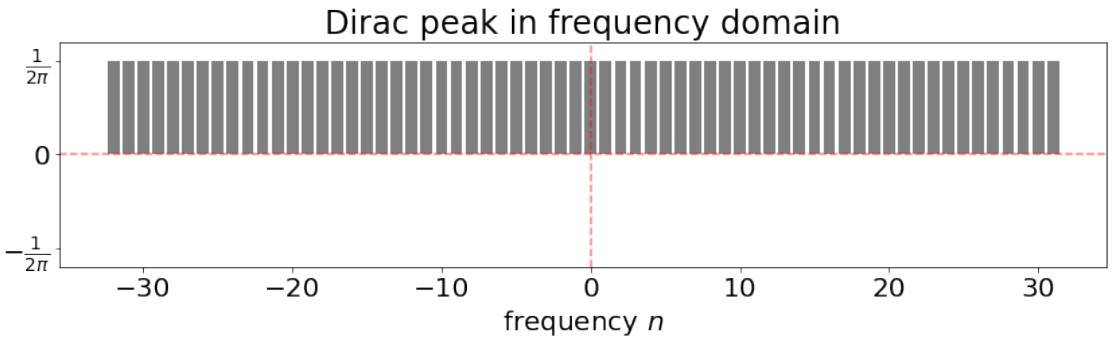
In the complex representation, we get:

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} \delta(x)\, e^{-inx}\, dx = \frac{1}{2\pi} \tag{11}$$

[2]:
```
# illustration of Fourier representation of Dirac delta
def ft(f):
    """Fourier transform with shifting and scaling. """
    return np.fft.fftshift(np.fft.fft(f))/len(f)

N = 2**6
x = np.linspace(-np.pi, np.pi, N, endpoint=False)
dx = x[1]-x[0]
delta = np.array(np.fabs(x) < dx/2, dtype=float)/dx
n = np.arange(-N//2, N//2)
c = 1/(2*np.pi)

plt.rc('font', size=20)
plot_kw = dict(color='k', alpha=0.5)
fig, ax = plt.subplots(figsize=(10, 5))
ax.set_title('Dirac peak in real domain')
ax.plot(x, delta, **plot_kw)
ax.set_xlabel(r'position $x$')
ax.set_ylabel(r'$\delta(x)$');
```



[3]:
```
fig, ax = plt.subplots(figsize=(12, 4), sharex='all', sharey='all')
ax.axvline(0., ls='--', color='r', alpha=0.5)
ax.set_title('Dirac peak in frequency domain')
ax.bar(n, np.abs(ft(delta)), **plot_kw)
ax.axhline(0., ls='--', color='r', alpha=0.5)
ax.set_ylim(-1.2*c, 1.2*c)
ax.set_yticks([-c, 0, c])
ax.set_yticklabels([r'$-\frac{1}{2\pi}$', r'$0$', r'$\frac{1}{2\pi}$'])
ax.set_xlabel(r'frequency $n$')
fig.tight_layout()
```



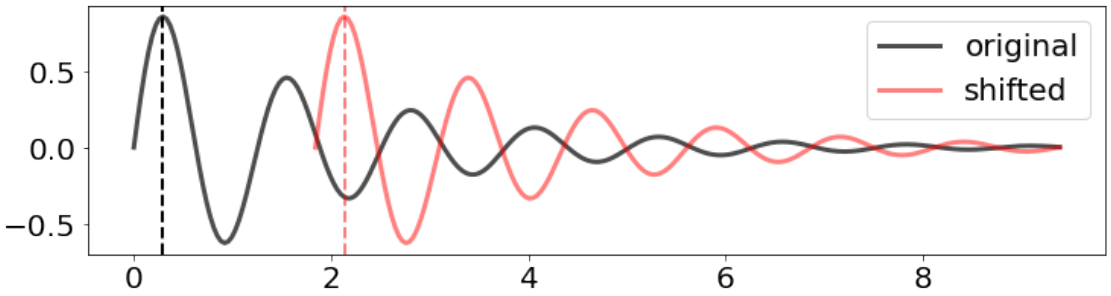
[4]:
```
delta_hat = np.full(N, c)
delta_hat[1::2] *= -1
delta_hat = np.fft.fftshift(delta_hat)
print(np.allclose(ft(delta), delta_hat))
```

```
True
```

[5]:
```
# Convolution with a delta function shifts a signal

N = 2**8
x = np.linspace(0., 3*np.pi, N, endpoint=False)
f = np.sin(5*x) * np.exp(-x/2)

shift = 50
peak = np.zeros_like(x)
peak[shift-1] = 1
peak = np.fft.ifftshift(peak)
shifted = np.convolve(f, peak, 'same')

fig, ax = plt.subplots(figsize=(12, 3))
ax.plot(x, f, lw=3, color='k', label='original', alpha=0.7)
```

```
ax.plot(x[shift:], shifted[shift:], color='r', lw=3, alpha=0.5, label='shifted')
ax.axvline(x[f.argmax()], ls='--', color='k', lw=2)
ax.axvline(x[shifted.argmax()], ls='--', color='r', lw=2, alpha=0.5)
ax.legend()
print(shifted.argmax()-f.argmax())
```

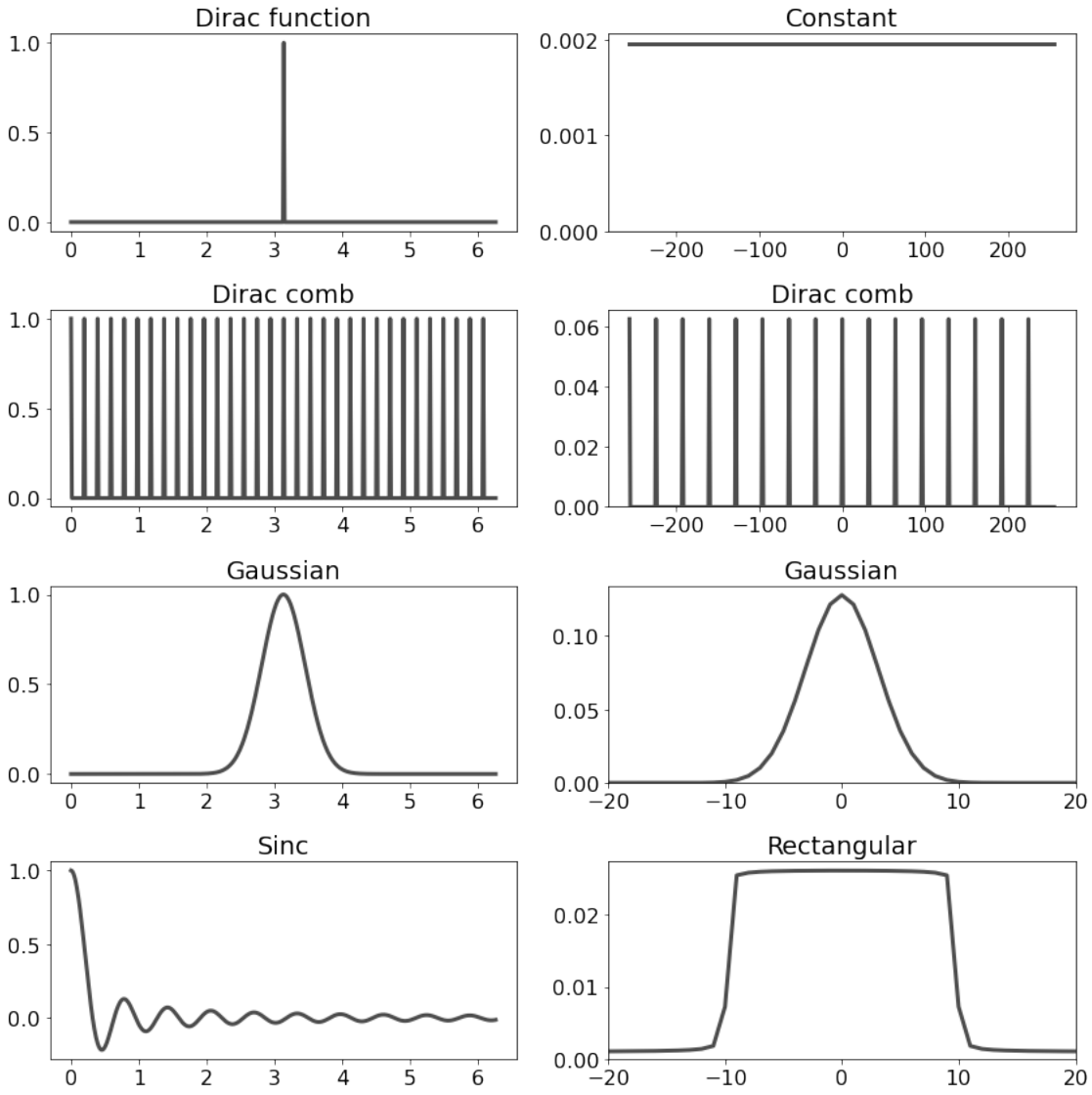
```
50
```



Fourier pairs

[6]:
```
# Fourier pairs
def ft(f):
    return np.fft.fftshift(np.fft.fft(f)) / len(f)

# number of sampling points and frequencies
N = 2**9
n = np.arange(-N//2, N//2)

kw = dict(lw=3, color='k', alpha=0.7)
plt.rc('font', size=16)
fig, axes = plt.subplots(4, 2, figsize=(12, 12))

# Dirac peak
x = np.linspace(0., 2*np.pi, N, endpoint=False)
f = np.zeros_like(x)
f[0] = 1.

index = 0
ax = axes[index]
ax[0].set_title('Dirac function')
ax[0].plot(x, np.fft.ifftshift(f), **kw)
ax[1].set_title('Constant')
ax[1].plot(n, np.real(ft(f)), **kw)
```

```
# Dirac comb
dx = N // 2**5
x = np.linspace(0., 2*np.pi, N, endpoint=False)
f = np.zeros_like(x)
f[::dx] = 1.

index += 1
ax = axes[index]
ax[0].set_title('Dirac comb')
ax[0].plot(x, f, **kw)
ax[1].set_title('Dirac comb')
ax[1].plot(n, np.abs(ft(f)), **kw)

# Gaussian
index += 1
ax = axes[index]
var = 1 / 2**4
f = np.exp(- 1/var * (x-x.mean())**2 / x.std()**2)
ax[0].set_title('Gaussian')
ax[0].plot(x, f, **kw)
ax[1].set_title('Gaussian')
ax[1].plot(n, np.real(ft(np.fft.ifftshift(f))), **kw)
ax[1].set_xlim(-20., 20.)

# sinc function: sin(x)/x
index += 1
ax = axes[index]
f = np.sinc(np.pi*x)
ax[0].set_title('Sinc')
ax[0].plot(x, f, **kw)
ax[1].set_title('Rectangular')
ax[1].plot(n, np.real(ft(f)), **kw)
ax[1].set_xlim(-20., 20.)

for ax in axes[:, 1]: ax.set_ylim(0., None)
fig.tight_layout()
```



Correlation

The ***(cross-)correlation*** of two continuous complex-valued functions $f(x)$ and $g(x)$ is defined as:

$$(f * g)(x) = \int_{-\infty}^{\infty} f^*(y)\, g(y+x)\, dy \tag{12}$$

where $f^*(y)$ denotes the complex conjugate of $f(y)$, and y is the displacement. An equivalent definition is:

$$(f * g)(x) = \int_{-\infty}^{\infty} f^*(y-x)\, g(y)\, dy \tag{13}$$

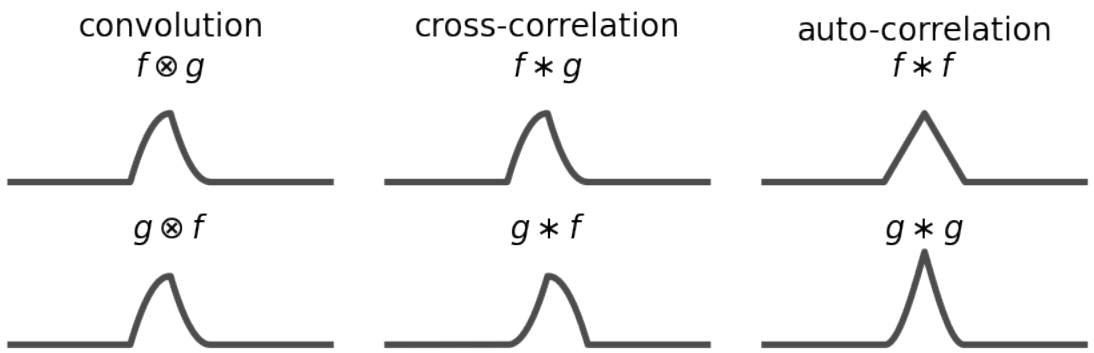
Equation (12) can be interpreted as the *convolution* of $f^*(-x)$ and $g(x)$. Therefore, to compute the correlation of two signals we can use the same techniques as those used to compute convolutions. The correlation assesses the *similarity* of two signals.

The ***auto-correlation*** is the correlation of a signal with itself.

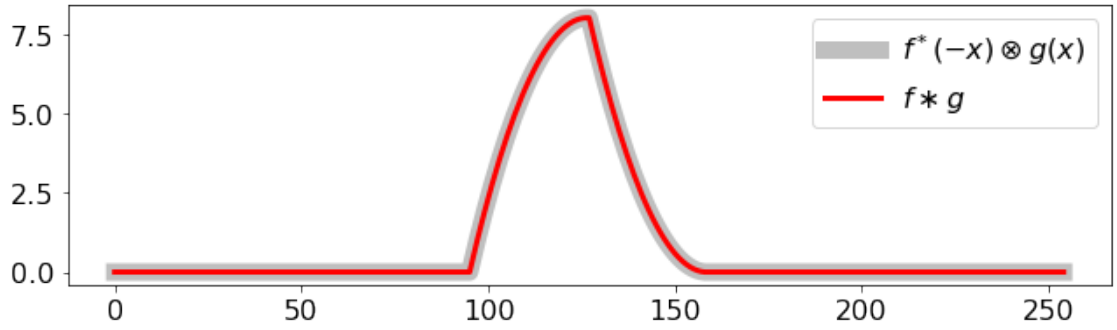
[7]:
```
ops = (
    # convolution
    np.convolve,
    # cross-correlation
    lambda f, g: np.convolve(np.conjugate(f[::-1]), g),
    # auto-correlation
    lambda f, g: np.convolve(np.conjugate(f[::-1]), f),
)

N = 128
x = np.linspace(-1., 1., N)
f, g = np.zeros((2, N))
mask = (-0.25<=x) & (x<=0.25)
f[mask] = 0.5
g[mask] = np.linspace(1., 0., mask.sum())

kw = dict(lw=5, color='k', alpha=0.7)
plt.rc('font', size=16)
fig, axes = plt.subplots(1, 2, figsize=(12, 3), sharex='all', sharey='row')
for ax, h, name in zip(axes, [f, g], ['f', 'g']):
    ax.set_title(r'${}(x)$'.format(name))
    ax.plot(h, **kw)
    ax.axis('off')
    ax.set_ylim(None, 1.5)
fig.tight_layout()
```

[8]:
```
fig, axes = plt.subplots(2, 3, figsize=(12, 4), sharex='all', sharey='all')
titles = ('convolution\n' + r'$f \otimes g$',
          'cross-correlation\n' + r'$f\ast g$',
          'auto-correlation\n' + r'$f\ast f$')
titles2 = (r'$g \otimes f$', r'$g\ast f$', r'$g\ast g$')
for ax, t, ax2, t2, op in zip(axes[0], titles, axes[1], titles2, ops):
```

```
    ax.set_title(t, fontsize=24)
    ax.plot(op(f, g), **kw)
    ax2.set_title(t2, fontsize=24)
    ax2.plot(op(g, f), **kw)
for ax in axes.flat:
    ax.axis('off')
fig.tight_layout()
```



[9]:
```
# simple Python implementation

def correlate(f, g):
    """Returns full cross-correlation.

    Parameters
    ----------
    f, g : np.ndarray
        Two signals

    Returns
    -------
    Full cross-correlation stored in an array of size 'len(f)+len(g)-1'
    """
    N, M = len(f), len(g)
    g = np.pad(g, M-1)
    return np.array([np.conjugate(f) @ g[i:i+M] for i in range(N+M-1)])

fig, ax = plt.subplots(figsize=(9, 3))
ax.plot(ops[1](f, g), color='k', lw=10, alpha=0.25,
        label=r'$f^*(-x) \otimes g(x)$')
ax.plot(correlate(f, g), color='r', lw=3, label=r'$f\ast g$')
ax.legend()
fig.tight_layout()
```



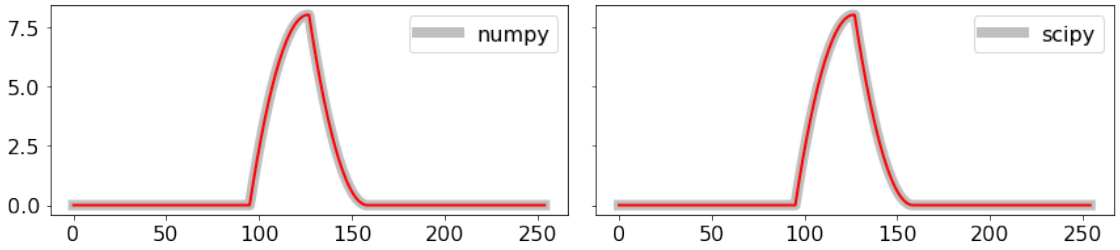
Python implementations

Beware that the implementations offered by NumPy and SciPy

- 1D signals: `np.correlate` and `scipy.signal.correlate`
- 2D signals: `scipy.signal.correlate2d`

use a different definition where the role of f and g is interchanged:

[10]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 3), sharex=True, sharey=True)
# using NumPy with f and g swapped
ax[0].plot(np.correlate(g, f, 'full'), color='k',
       lw=8, label='numpy', alpha=0.25)
ax[0].plot(ops[1](f, g), color='r', lw=2)
ax[0].legend();
# using SciPy with f and g swapped
ax[1].plot(sig.correlate(g, f, 'full'), color='k',
       lw=8, label='scipy', alpha=0.25)
ax[1].plot(ops[1](f, g), color='r', lw=2)
ax[1].legend();
fig.tight_layout()
```



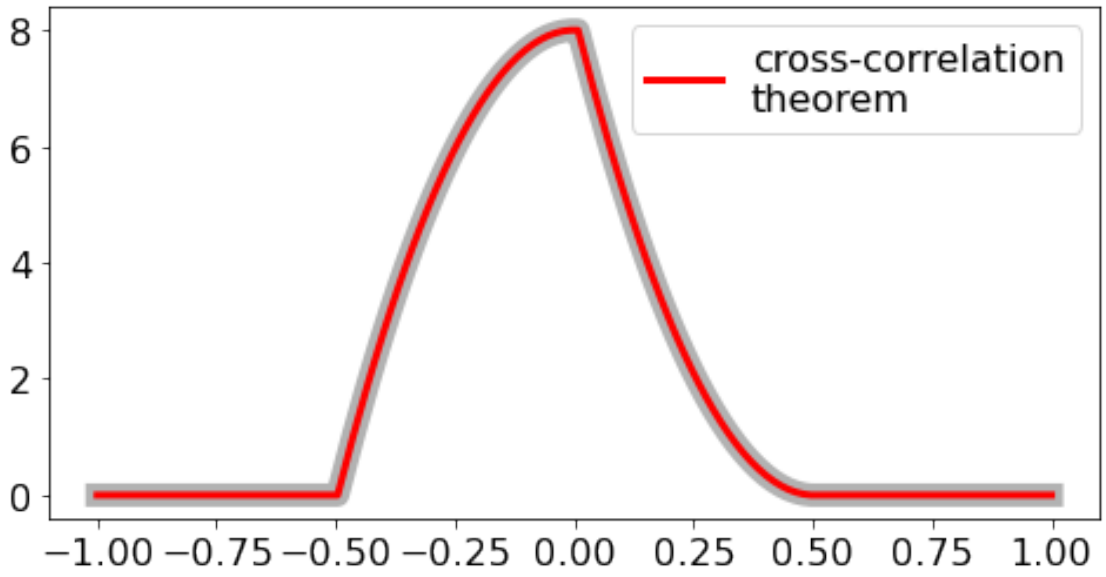
Cross-correlation theorem

The ***cross-correlation theorem*** is the analog of the convolution theorem

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\}^* \odot \mathcal{F}\{g\} \tag{14}$$

In case of the auto-correlation ($g = f$), the correlation theorem is called the *Wiener-Khinchin theorem*.

[11]:
```
# testing of the cross-correlation theorem
F = np.fft.fft(f)
G = np.fft.fft(g)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, np.correlate(g, f, 'same'), color='k', lw=10, alpha=0.3)
ax.plot(x, np.fft.ifftshift(np.fft.ifft(F.conj() * G).real), lw=3, color='r',
        label='cross-correlation\ntheorem')
ax.legend();
```



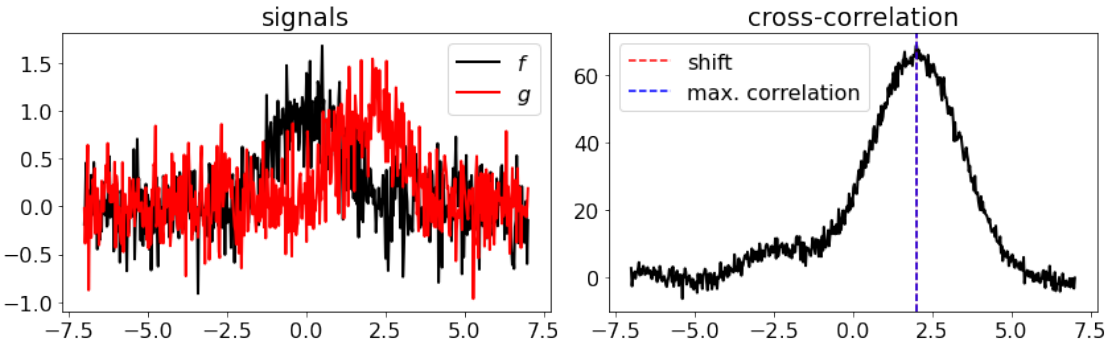
Correlation as similarity measure

The cross-correlation can be used to measure the similarity of two signals and to determine the optimal shift so as to maximize their overlap. This is commonly used in image registration methods.

In the following example, we look at two noisy versions of the same signal where one version has been shifted.

[12]:
```
# Gaussian signal
N = 2**9
x = np.linspace(-7., 7., N)
shift = 2
```

```
f = np.exp(-0.5*x**2) # zero-centered Gaussian
g = np.exp(-0.5*(x-shift)**2) # shifted Gaussian
# add some noise
f += np.random.randn(N) * 0.3
g += np.random.randn(N) * 0.3

xcorr = np.correlate(g, f, 'same')
fig, ax = plt.subplots(1, 2, figsize=(12, 4), sharex=True)
ax[0].set_title('signals')
ax[0].plot(x, f, label=r'$f$', lw=2, color='k')
ax[0].plot(x, g, label=r'$g$', lw=2, color='r')
ax[0].legend()
ax[1].set_title('cross-correlation')
ax[1].plot(x, xcorr, lw=2, color='k')
ax[1].axvline(shift, color='r', ls='--', label='shift')
ax[1].axvline(x[xcorr.argmax()], color='b', ls='--', label='max. correlation')
ax[1].legend()
fig.tight_layout()
```



Sampling

In signal processing, ***sampling*** refers to the reduction of a continuous to a discrete signal. Functions that vary in time, space, or other dimensions can be sampled.

Let $f(x)$ be a continuous time-domain signal that we want to sample, and let sampling be performed by acquiring $f(x)$ every Δx seconds. The increment Δx is called the **sampling interval** (for spatial signals) or the **sampling period** (for time-domain signals). Then the sampled function is given by the sequence:

$$f_n = f(n\Delta x), \;\; n = 0, 1, 2, \ldots, N-1$$

Mathematical description

Sampling is equivalent to a point-wise multiplication of the signal with a ***Dirac comb***:

$$f(x) \overset{sampling}{\rightarrow} f(x) \odot \underbrace{\sum_n \delta(x - n\Delta x)}_{\text{Dirac comb}} = \sum_n f(n\Delta x)\, \delta(x - n\Delta x) = \sum_n f_n\, \delta(x - n\Delta x)$$

Practical considerations

In computer programs for image processing, we never deal with continuous signals but finite discrete samples. We can use the samples, for example, to generate a plot that gives us the impression that we were looking at a continuous signal.

As usual let the period of our signal be 2π and N the number of fine sampling points (typically we choose N to be rather large). Let the size-N array `f` store a finely sampled signal:

```
f = [signal(2*np.pi*0/N), signal(2*np.pi*1/N), ..., signal(2*np.pi*(N-1)/N)]
```

If the Python function `signal` implements our signal, then we can easily sample the signal by calling

```
x = np.linspace(0., 2*np.pi, N, endpoint=False)
f = signal(x)
```

or equivalently

```
x = 2 * np.pi * np.arange(N) / N
f = signal(x)
```

The sampling increment is $\Delta x = 2\pi/N$. Using N sampling points, we can cover all frequencies in the interval $[-N/2, N/2)$.

If we now sample the signal using a coarser sampling interval $\Delta x = 2\pi/M$ where $M < N$, and if $s = N/M > 1$ (assuming N is a multiple of M), then we can use s as a step and generate the coarser sample by slicing

```
f_coarse = f[::s]
```

So the coarse sampling interval is now $\Delta x = 2\pi s/N = 2\pi/M$ in real space. This sampling density allows us to cover a reduced frequency interval $[-M/2, M/2) = [-N/(2s), N/(2s))$.

In the following test code, we will identify the slicing step s with the sampling interval Δx and call the variable `dx`.

Example

Let the signal be

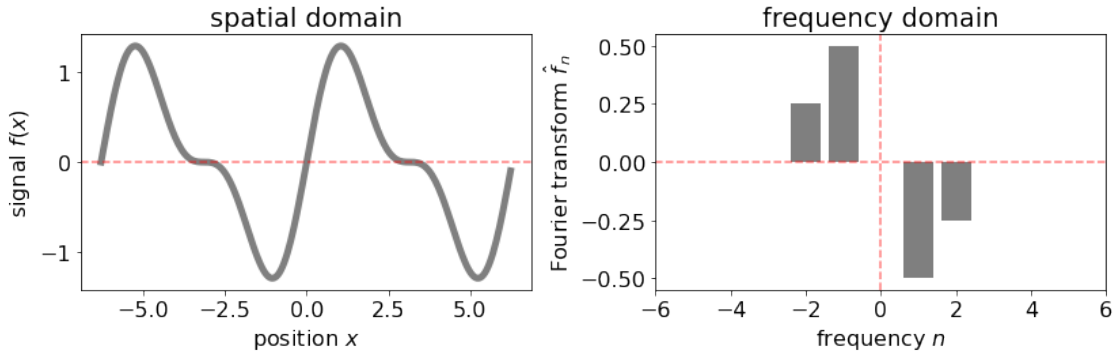
$$\begin{aligned} f(x) &= \sin(x) + \frac{1}{2}\sin(2x) = i\frac{e^{-ix} - e^{ix}}{2} + i\frac{e^{-2ix} - e^{2ix}}{4} \\ &= \frac{i}{4}e^{-2ix} + \frac{i}{2}e^{-ix} - \frac{i}{2}e^{ix} - \frac{i}{4}e^{2ix} \end{aligned}$$

The only non-zero Fourier coefficients are:

$$c_{-2} = \frac{i}{4},\ c_{-1} = \frac{i}{2},\ c_1 = -\frac{i}{2},\ c_2 = -\frac{i}{4}$$

[13]:
```
def signal(x):
    """Superposition of two sine waves. """
    return np.sin(x) + 0.5 * np.sin(2*x)

# number of (fine) samples per period
N = 2**7

# we plot the signal over two periods
x = np.linspace(-2*np.pi, 2*np.pi, 2*N, endpoint=False)

# evaluating signal over two periods
f = signal(x)

# single frequency period
n = np.arange(-N//2, N//2)
```

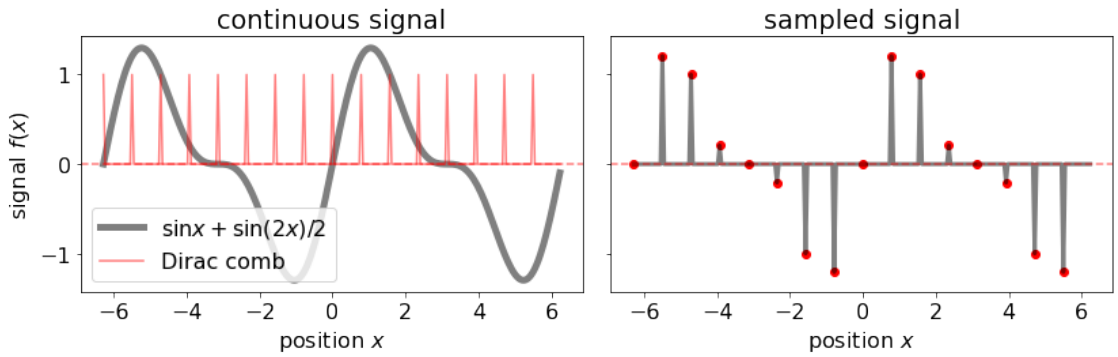
[14]:
```
plt.rc('font', size=16)
fig, ax = plt.subplots(1, 2, figsize=(12, 4))

# plotting two periods of the signal
ax[0].set_title('spatial domain')
ax[0].plot(x, f, lw=5, color='k', alpha=0.5)
ax[0].set_xlabel(r'position $x$')
ax[0].set_ylabel(r'signal $f(x)$')
ax[0].axhline(0., ls='--', color='r', alpha=0.5)

# evaluating the Fourier transform for a single period (i.e. the
# first half of the signal)
ax[1].set_title('frequency domain')
ax[1].bar(n, np.imag(ft(f[:N])), color='k', alpha=0.5)
ax[1].set_xlabel(r'frequency $n$')
ax[1].set_ylabel(r'Fourier transform $\hat f_n$')
ax[1].axhline(0., ls='--', color='r', alpha=0.5)
ax[1].axvline(0., ls='--', color='r', alpha=0.5)
ax[1].set_xlim(-6, 6)

fig.tight_layout();
```



We sample every 16-th element of the signal by multiplying it with a Dirac comb whose peaks are spaced by $\Delta x = 2\pi 16/N = \pi/4$ (since $N = 128$ in our test code), so we have $128/16 = 8$ samples per period:

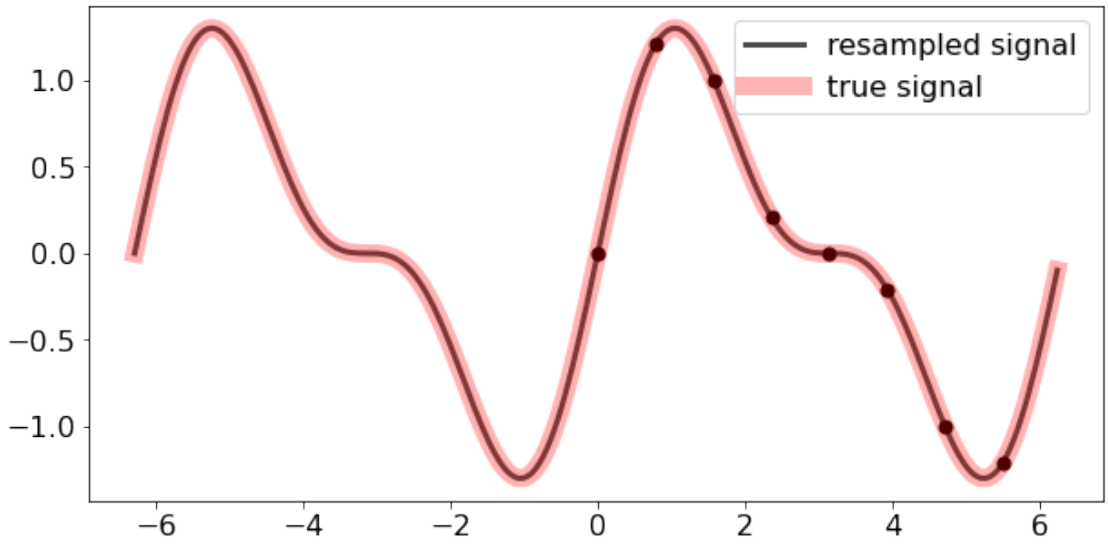
[15]:
```
# coarse sampling of the signal
dx = 2**4
dirac_comb = np.zeros(2*N)
dirac_comb[::dx] = 1.

fig, ax = plt.subplots(1, 2, figsize=(12, 4), sharex=True, sharey=True)

ax[0].set_title('continuous signal')
ax[0].plot(x, f, lw=5, color='k', alpha=0.5, label=r'$\sin x + \sin(2x)/2$')
ax[0].plot(x, dirac_comb, ls='-', color='r', alpha=0.5, label='Dirac comb')
ax[0].set_xlabel(r'position $x$')
ax[0].set_ylabel(r'signal $f(x)$')
ax[0].axhline(0., ls='--', color='r', alpha=0.5)
ax[0].legend()

ax[1].set_title('sampled signal')
ax[1].plot(x, f * dirac_comb, lw=3, color='k', alpha=0.5)
ax[1].scatter(x[::dx], f[::dx], color='r')
ax[1].set_xlabel(r'position $x$')
ax[1].axhline(0., ls='--', color='r', alpha=0.5)
fig.tight_layout();

# verify that sampling interval is indeed pi/4
print(np.allclose(np.diff(x[dirac_comb>0]), np.pi/4))
```

```
True
```



If the continuous signal is sampled properly (we still need to find out what "properly" means in this context), it can be reconstructed via interpolation / resampling:

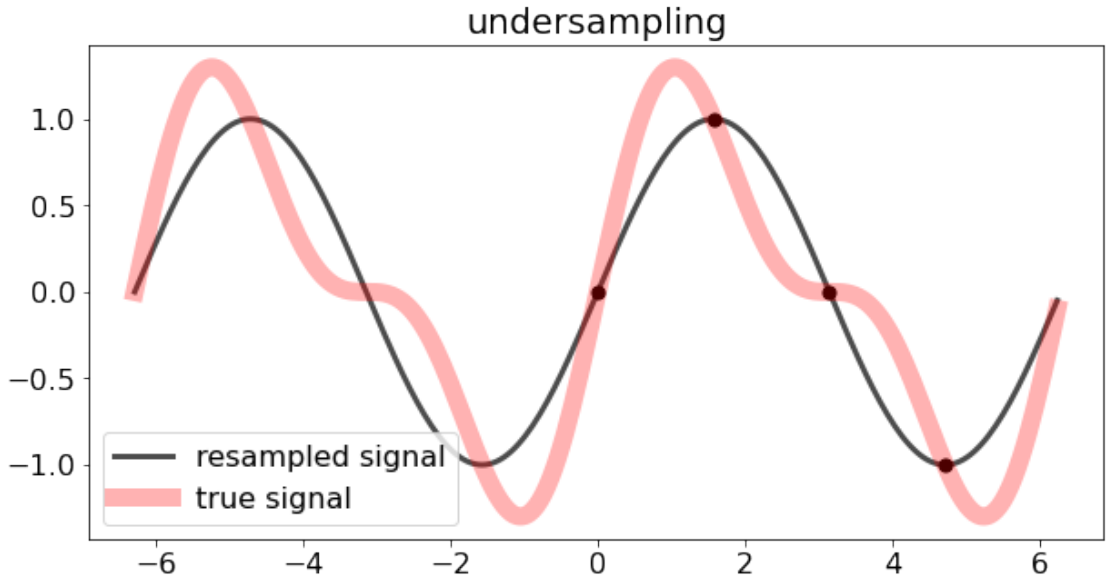
[16]:
```
# (coarse) sampling of signal
dx = 2**4
f_sampled = f[N::dx]

# Fourier transform of samples and corresponding frequencies
F_sampled = ft(f_sampled)
n_sampled = np.arange(-len(f_sampled)//2, len(f_sampled)//2)

# inverse discrete Fourier transform where spatial domain is
# sampled 16x more densely (resampling)
F_inv = np.exp(1j*np.multiply.outer(x, n_sampled))
f_reconstruct = F_inv.dot(F_sampled).real

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(x, f_reconstruct, lw=3, color='k', alpha=0.7,
        label='resampled signal')
ax.plot(x, f, lw=10, color='r', alpha=0.3, label='true signal')
ax.scatter(x[N::dx], f_sampled, color='k', s=50)
ax.legend();
```



However if we do not sample the signal densely enough (*undersampling*), problems will occur: The reconstructed signal differs from the true signal. We need to find out when this happens and how to avoid it.

[17]:
```
# showing the effect of undersampling

def resample(f_sampled, x):
    """Convenience function for resampling a signal. """
    # compute Fourier transform of samples
    F_sampled = ft(f_sampled)
    n_sampled = np.arange(-len(f_sampled)//2, len(f_sampled)//2)

    # resample by evaluating the inverse Fourier transform
    F_inv = np.exp(1j*np.multiply.outer(x, n_sampled))

    return F_inv.dot(F_sampled).real


def resample2(f_sampled, N):
    """Alternative implementation using the FFT but covering only a
    single period. """
    assert len(f_sampled) < N

    # compute FFT and shift zero-frequency component to center
    # account for factor `len(f_sampled)` by dividing by it
    F = np.fft.fftshift(np.fft.fft(f_sampled)) / len(f_sampled)

    # fill in higher frequencies by padding zeros
    F = np.pad(F, (N-len(F)) // 2) * N
```

```
    return np.fft.ifft(np.fft.ifftshift(F)).real


# very coarse sampling corresponding to a sampling interval of pi/2
dx = 2**5
f_sampled = f[N::dx]

# resampling
if not True:
    f_reconstruct = resample(f_sampled, x)
else:
    # FFT-based method
    f_reconstruct = resample2(f_sampled, N)
    # to cover two periods we need to repeat the resampled signal
    f_reconstruct = np.append(f_reconstruct, f_reconstruct)

fig, ax = plt.subplots(figsize=(10, 5))
ax.set_title('undersampling')
ax.plot(x, f_reconstruct, lw=3, color='k', alpha=0.7,
        label='resampled signal')
ax.plot(x, f, lw=10, color='r', alpha=0.3, label='true signal')
ax.scatter(x[N::dx], f_sampled, color='k', s=50)
ax.legend();
```



Sampling in the Fourier domain

In the spatial or time domain, sampling is equivalent to a point-wise multiplication of the signal with a Dirac comb:

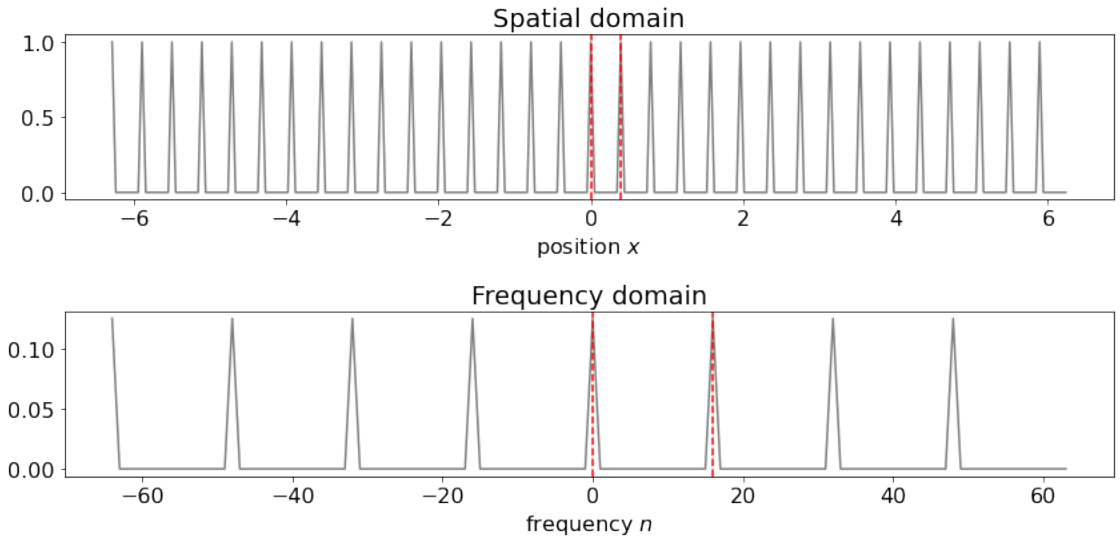
$$f(x) \overset{sampling}{\rightarrow} f(x) \odot \underbrace{\sum_n \delta(x - n\Delta x)}_{\text{Dirac comb}}$$

In the frequency domain, we can understand the sampling process as the *convolution* of the Fourier transforms of the signal and the Dirac comb thanks to the convolution theorem (Eq. 2). The Fourier transform of a Dirac comb is again a Dirac comb. Therefore, in the Fourier domain, sampling boils down to the convolution of the Fourier transform of the signal with the Dirac comb:

$$\mathcal{F}\left[f(x) \odot \sum_k \delta(x - k\Delta x)\right](n) = \hat{f}(n) \otimes \sum_k \delta(n - k\Delta n)$$

where the sampling interval in the frequency domain is $\Delta n = 2\pi/\Delta x = 2\pi/(2\pi s/N) = N/s$.

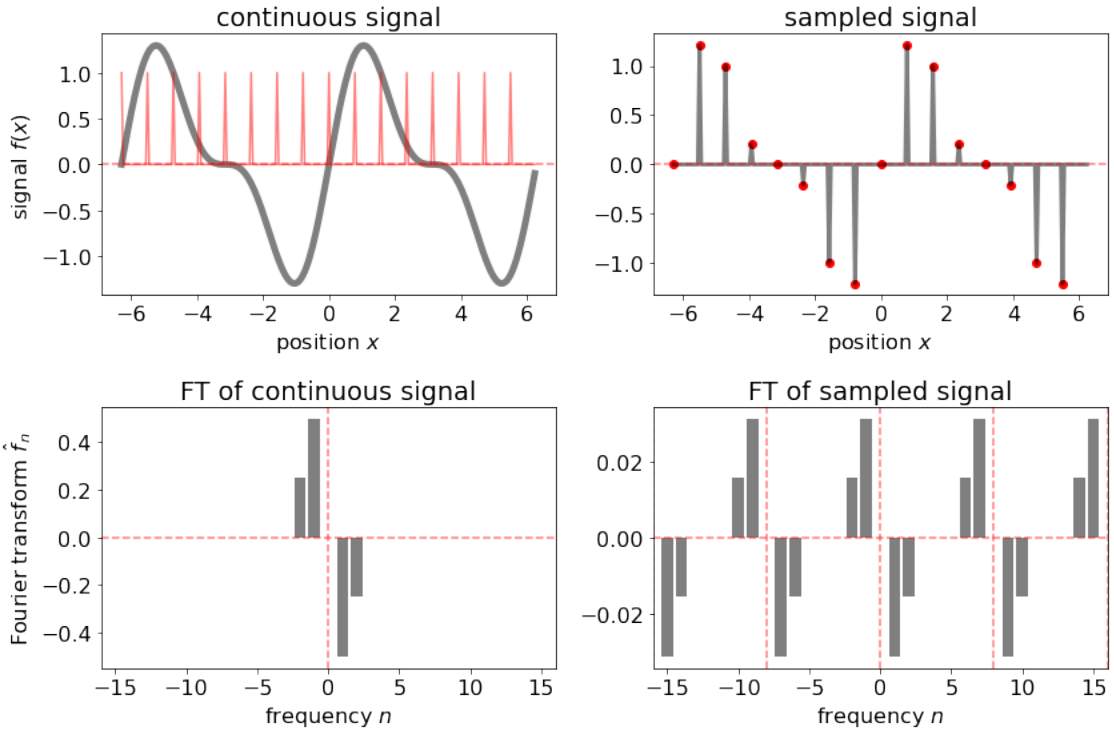
[18]:
```
# relation between Dirac combs in real and frequency space
dx = 2**3
dirac_comb = np.zeros(2*N)
dirac_comb[::dx] = 1.

delta_x = 2*np.pi/N*dx
delta_n = 2*np.pi/delta_x
assert int(delta_n) == N // dx

fig, ax = plt.subplots(2, 1, figsize=(12, 6))

ax[0].set_title('Spatial domain')
ax[0].plot(x, dirac_comb, color='k', alpha=0.5)
ax[0].set_xlabel(r'position $x$')
ax[0].axvline(0., ls='--', color='r')
ax[0].axvline(delta_x, ls='--', color='r')

ax[1].set_title('Frequency domain')
ax[1].plot(n, np.abs(ft(dirac_comb[N:])), color='k', alpha=0.5)
ax[1].set_xlabel(r'frequency $n$')
ax[1].axvline(0., ls='--', color='r')
ax[1].axvline(delta_n, ls='--', color='r')

fig.tight_layout()
```



Since the convolution of a signal with a delta peak shifts the signal, the convolution of a signal (or its Fourier transform) with a Dirac comb results in generating many *copies* of the signal at different locations:

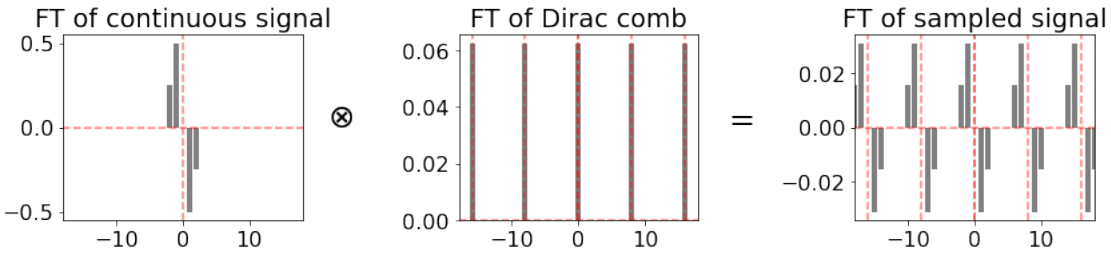
[20]:
```
dx = 2**4
dirac_comb = np.zeros(2*N)
dirac_comb[::dx] = 1.

delta_x = 2*np.pi/N*dx
delta_n = 2*np.pi/delta_x
assert int(delta_n) == N // dx

fig, ax = plt.subplots(2, 2, figsize=(12, 8), sharex='row')
ax = list(ax.flat)

# plotting continuous and sampled signal
ax[0].set_title('continuous signal')
ax[0].plot(x, f, lw=5, color='k', alpha=0.5)
ax[0].plot(x, dirac_comb, ls='-', color='r', alpha=0.5, label='Dirac comb')
ax[0].set_xlabel(r'position $x$')
ax[0].set_ylabel(r'signal $f(x)$')
ax[0].axhline(0., ls='--', color='r', alpha=0.5)
ax[1].set_title('sampled signal')
ax[1].plot(x, f * dirac_comb, lw=3, color='k', alpha=0.5)
ax[1].scatter(x[::dx], f[::dx], color='r')
ax[1].set_xlabel(r'position $x$')
ax[1].axhline(0., ls='--', color='r', alpha=0.5)

# Fourier transforms of continuous and sampled signal
```

```
ax[2].set_title('FT of continuous signal')
ax[2].bar(n, np.imag(ft(f[N:])), color='k', alpha=0.5)
ax[2].set_xlabel(r'frequency $n$')
ax[2].set_ylabel(r'Fourier transform $\hat f_n$')
ax[2].axhline(0., ls='--', color='r', alpha=0.5)
ax[2].axvline(0., ls='--', color='r', alpha=0.5)
ax[3].set_title('FT of sampled signal')
ax[3].bar(n, np.imag(ft((dirac_comb*f)[N:])), color='k', alpha=0.5)
ax[3].axhline(0., ls='--', color='r', alpha=0.5)
for i in range(-2, 3):
    ax[3].axvline(i * delta_n, ls='--', color='r', alpha=0.5)
ax[3].set_xlabel(r'frequency $n$')
ax[3].set_xlim(-16, 16)
fig.tight_layout()
```



[21]:
```
# demonstration that in Fourier space, sampling is a convolution with a Dirac
# comb which generates multiple copies of the Frequency spectrum at different
# locations

kw = dict(color='k', alpha=0.5)
fig, ax = plt.subplots(1, 3, figsize=(12, 3), sharex='all')
ax[0].set_title('FT of continuous signal')
ax[0].bar(n, ft(f[N:]).imag, **kw)
```

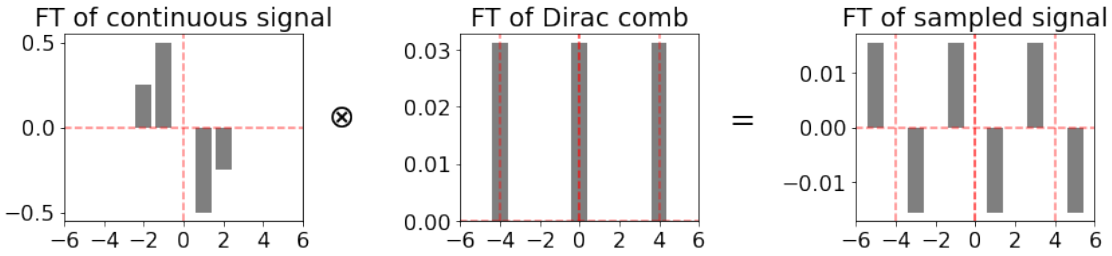
```
ax[0].annotate(r'$\otimes$', xy=(1.1, .5), fontsize=24, xycoords='axes fraction')
ax[1].set_title('FT of Dirac comb')
ax[1].bar(n, np.abs(ft(dirac_comb[N:])), **kw)
ax[1].annotate(r'$=$', xy=(1.1, .5), fontsize=24, xycoords='axes fraction')
ax[2].set_title('FT of sampled signal')
ax[2].bar(n[:-1],
          np.convolve(ft(f[N:]), ft(dirac_comb[N:]), mode='same').imag[1:],
          **kw)
ax[2].set_xlim(-18, 18)
for a in ax:
    a.axhline(0., ls='--', color='r', alpha=0.5)
    a.axvline(0., ls='--', color='r', alpha=0.5)
for a in ax[1:]:
    for i in range(-2, 3):
        a.axvline(i * delta_n, ls='--', color='r', alpha=0.5)
fig.tight_layout()
```



In the Fourier domain, undersampling means that the sampling interval Δx is so large, that the copies of the Fourier transform of the sampled signal start to overlap. The frequency threshold at which the overlap starts to occur is called ***Nyquist frequency***.

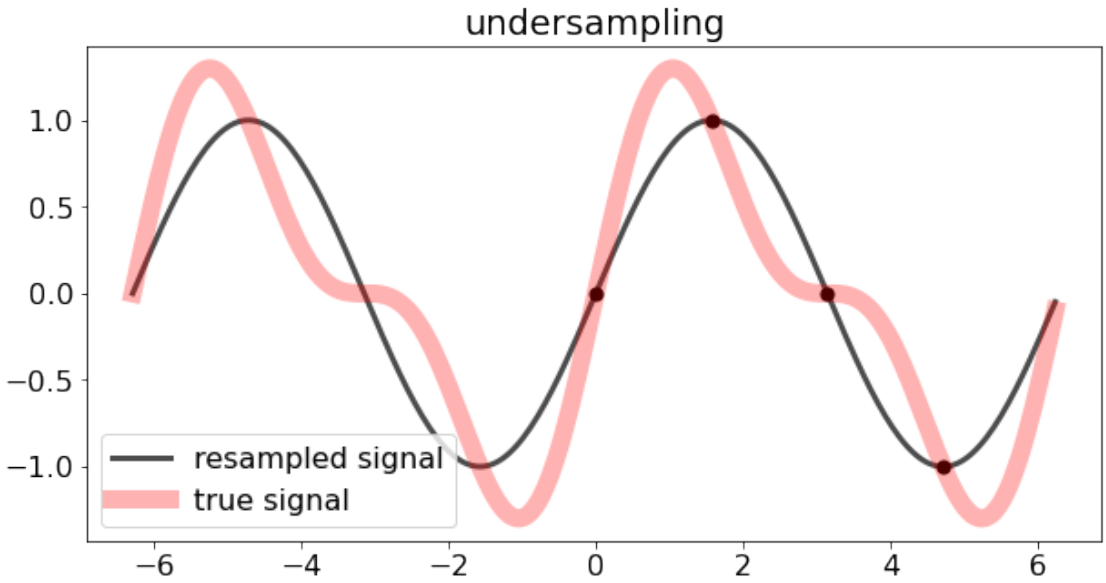
[22]:
```
# too coarse sampling corresponding to a sampling interval of pi/2
dx = 2**5
dirac_comb = np.zeros(2*N)
dirac_comb[::dx] = 1.

delta_x = 2*np.pi/N*dx
delta_n = 2*np.pi/delta_x
assert int(delta_n) == N // dx

kw = dict(color='k', alpha=0.5)
fig, ax = plt.subplots(1, 3, figsize=(12, 3), sharex='all')
ax[0].set_title('FT of continuous signal')
ax[0].bar(n, ft(f[N:]).imag, **kw)
ax[0].annotate(r'$\otimes$', xy=(1.1, .5), fontsize=24, xycoords='axes fraction')
ax[1].set_title('FT of Dirac comb')
ax[1].bar(n, np.abs(ft(dirac_comb[N:])), **kw)
```

```
ax[1].annotate(r'$=$', xy=(1.1, .5), fontsize=24, xycoords='axes fraction')
ax[2].set_title('FT of sampled signal')
ax[2].bar(n[:-1],
          np.convolve(ft(f[N:]), ft(dirac_comb[N:]), mode='same').imag[1:],
          **kw)
ax[2].set_xlim(-6, 6)
ax[2].set_xticks(np.arange(-6, 7, 2))
for a in ax:
    a.axhline(0., ls='--', color='r', alpha=0.5)
    a.axvline(0., ls='--', color='r', alpha=0.5)
for a in ax[1:]:
    for i in range(-2, 3):
        a.axvline(i * delta_n, ls='--', color='r', alpha=0.5)
fig.tight_layout()
```



The frequency contributions at $n = \pm 2$ erase each other if the sampling is too coarse, such that only a single contribution at $n = \pm 1$ is left. Therefore, the resampling procedure misrepresents the signal as a single sine function:

[23]:
```
f_sampled = f[N::dx]
f_reconstruct = resample(f_sampled, x)

fig, ax = plt.subplots(figsize=(10, 5))
ax.set_title('undersampling')
ax.plot(x, f_reconstruct, lw=3, color='k', alpha=0.7,
        label='resampled signal')
ax.plot(x, f, lw=10, color='r', alpha=0.3, label='true signal')
ax.scatter(x[N::dx], f_sampled, color='k', s=50)
ax.legend();
```



Aliasing

When an analog signal is digitized at an inadequate sampling frequency, a phenomenon known as ***aliasing*** occurs. Aliasing can take place in time (temporal aliasing) or space (spatial aliasing).

Aliasing causes continuous signals of different frequencies to become indistinguishable (or aliases of one another) in the course of sampling. When this happens, the original signal cannot be uniquely reconstructed from the sampled signal.

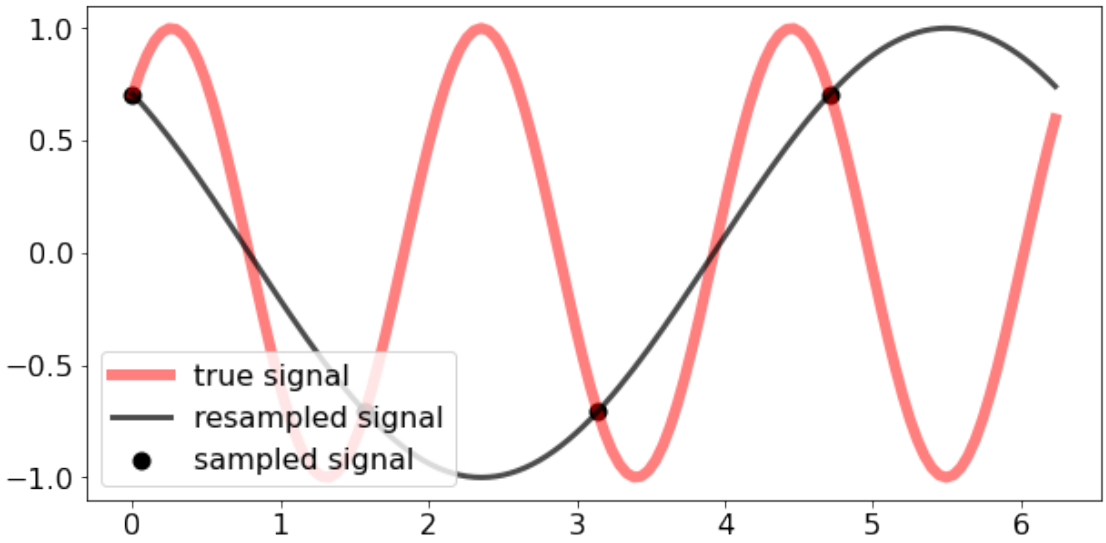
Aliasing can result not only in the loss of important high-frequency information, but also in the introduction of spurious lower-frequency features.

Example

In the following examples, the black dots are the digital samples taken to record the red signal. Apparently, these are not sufficient to reconstruct the original signal. Interpolation from the black dots yields the "wrong" black signal, but not the "true" red signal.

[24]:
```
N = 2**7
x = np.linspace(0., 2*np.pi, N, endpoint=None)
phase = np.pi / 4
f = np.sin(3*x + phase)
dx = 32
g = resample(f[::dx], x)

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(x, f, label='true signal', lw=6, color='r', alpha=0.5)
ax.plot(x, g, label='resampled signal', lw=3, color='k', alpha=0.7)
```

```
ax.scatter(x[::dx], f[::dx], lw=3, color='k', label='sampled signal', s=50)
ax.legend(loc=3);
```

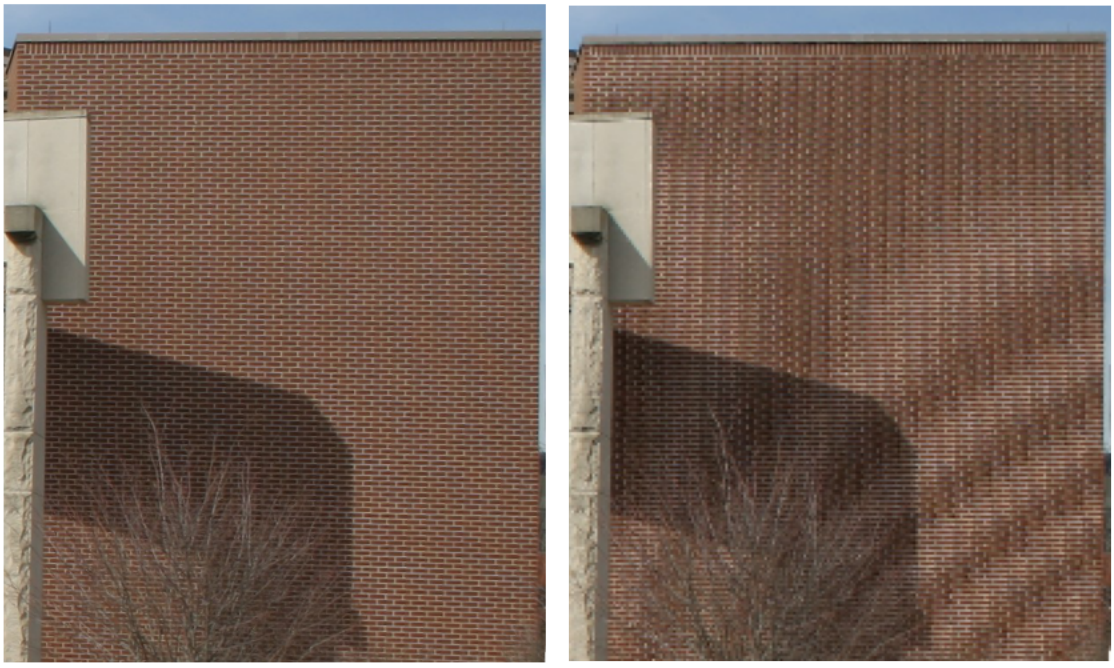


Aliasing effects in images

Aliasing can take place both in time (temporal aliasing) and space (spatial aliasing). In undersampled images, high spatial frequencies of finely textured objects appear to be "aliased" as low frequencies and can give rise to Moiré patterns.

[25]:
```
# example from wikipedia
from skimage import io
url = 'https://upload.wikimedia.org/wikipedia/commons/3/31/
 ↪Moire_pattern_of_bricks.jpg'
image = io.imread(url)
```

[26]:
```
dx = 3
fig, ax = plt.subplots(1, 2, figsize=(12, 9))
ax[0].imshow(image)
ax[1].imshow(image[::dx, ::dx])
for a in ax:
    a.axis('off')
fig.tight_layout()
```



The Nyquist-Shannon theorem

Shannon's version of the theorem states:

If a function $f(x)$ contains no frequencies higher than $n_{\max}$, it is completely determined by giving its ordinates at a series of points spaced $1/(2n_{\max})$ apart.

A sufficient sampling rate is therefore anything larger than $2n_{\max}$ samples (the Nyquist rate). Equivalently, for a given sampling rate s, a perfect reconstruction is guaranteed to be possible for band-limited signals with maximum frequency $n_{\max} < s/2$.

If the bandlimit is too high (or there is no bandlimit at all), frequencies higher than the *Nyquist frequency* $s/2$ will influence the samples in a way that is misinterpreted by the interpolation process. Aliasing occurs.

We can avoid aliasing by first low-pass filtering the signal at the expense of blurring the original image.

[27]:
```
image = image.astype(float)
image /= image.max()

# for filtering
size = 3
kernel = np.ones((size, size)) / size**2

# apply filter to each color channel
image_filtered = np.zeros_like(image)
```

```
for channel in range(image.shape[2]):
    image_filtered[:, :, channel] = sig.convolve2d(
        image[:, :, channel], kernel, 'same'
    )
fig, ax = plt.subplots(1, 3, figsize=(12, 6))
ax[0].set_title('Original')
ax[0].imshow(image)
ax[1].set_title('Filtered ({0}x{0})'.format(size))
ax[1].imshow(image_filtered)
ax[2].set_title('Filtered + downsampled')
ax[2].imshow(image_filtered[::dx, ::dx])
for a in ax:
    a.axis('off')
fig.tight_layout()
```