1b Advection

Background

Advection is a central phenomenon in all kinds of fluid dynamics. To quote Wikipedia: "*In the field of physics, engineering, and earth sciences, advection is the transport of a substance by bulk motion*".

Any property of a gas or fluid, such as color, temperature, density, or even velocity or momentum, may be "advected" by the flow.

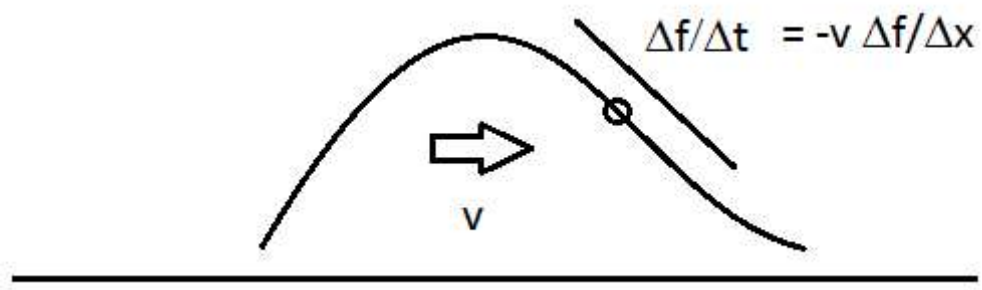
One way to think of and understand advection, and its central importance in fluid dynamics, is to consider the difference between studying the behavior of some system in a coordinate system moving with the flow (a *co-moving* coordinate system), and a stationary one (a *lab-frame* coordinate system).

Since we are looking at the same phenomenon, the terms that describe the physics (forces, heating / cooling, etc) are the same, and the only difference is apparent, and is due to the motion of the coordinate system. If what happens is described by $f({\bf r},t)$, then in a system moving with velocity $\bf v$, the same thing is described by $f({\bf r}-{\bf v}t,t)$.

From the derivative chain-rule, the motion gives rise to a difference in partial time derivate $\partial f/\partial t$, which is

\begin{equation}\tag{1} - v_x \partial f/\partial x - v_y \partial f/\partial y - v_z \partial f/\partial z = -\,{\bf v} \cdot \nabla f \end{equation}

If, for example, the shape of a function is stationary in the co-moving coordinate system, then in the lab-frame one sees the function shape passing by, with the local value changing slowly where the function is steep.



It is customary to write the partial time derivative in the lab-frame $\partial/\partial t$, while the one in the co-moving frame is written ${\rm D}/{\rm D}t$, so

\begin{equation}\tag{2} {\rm D}f/{\rm D}t = \partial f/\partial t + {\bf v \cdot \nabla} f \end{equation}

Numerical Advection

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Advecting a sine-wave

We can test numerical advection by introducing a time stepping scheme, for example the second order Adams-Bashforth scheme (this can be thought of as an extrapolation in the derivative, see also the useful wikipedia entry: https://en.wikipedia.org/wiki/Linear_multistep_method). The method can be derived in many ways, for the second order scheme we can simply match Taylor expansions forward and backwards in time:

\begin{equation} f(t + \Delta t) = f(t) + f'(t) \Delta t + {1\over 2}f''(t) \Delta t^2 + \mathcal{O}(\Delta t^3) \end{equation}

\begin{equation} f'(t - \Delta t) \Delta t = f'(t) \Delta t - f''(t) \Delta t^2 + \mathcal{O}(\Delta t^3) \end{equation}

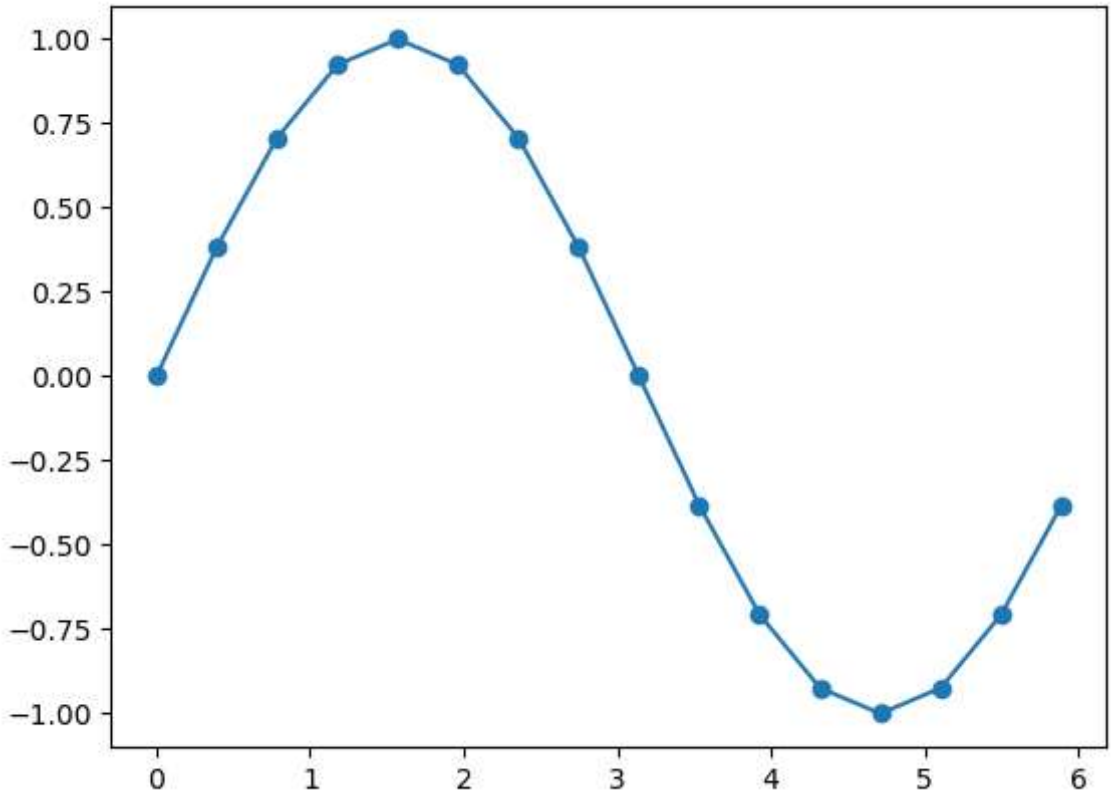
Multiplying the second equation by $1/2$ and adding them together we find after rearrangement \begin{equation} f(t + \Delta t) = f(t) + {3\over 2}f'(t) \Delta t - {1\over 2}f'(t-\Delta t) \Delta t + \mathcal{O}(\Delta t^3) \end{equation}

In [22]:
```
def adams_bashforth(f0,f1,dfdt,dt):
    """ 2nd order Adams-Bashforth time stepping scheme:
        f1: function at time t
        f0: function at time t-dt
        dfdt: procedure giving the time derivative
    """
    return f1+dt*(1.5*dfdt(f1)-0.5*dfdt(f0))
```

Let us use this to advect the function f(x) with velocity v, using a time step equal to `courant*ds/v`, where `ds` is the cell size, and `v` is the velocity amplitude:

Initial condition

In [23]:
```
n=16
v=1.0
def coordinates(n):
    ds=2.0*np.pi/n
    x=ds*np.arange(n)
    return ds,x
def IC(x,v,t):
    f=np.sin(x-v*t)
    return f
ds,x=coordinates(n)
f0=IC(x,v,0.0)
plt.plot(x,f0,'-o');
```



Courant condition

This routine computes the maximum *wavespeed* -- simply v for advection -- and sets the timestep such that features in the solution do not propagate more than $C\times\Delta x$, a C fraction of a cell, from one iteration to the next.

In [24]:
```
v=1.0
def courant(C,v,ds):
    dt=C/np.max(v/ds)
    return dt
dt=courant(0.2,v,ds)
print('dt={:.4f}'.format(dt))
```

```
dt=0.0785
```

Time derivative procedure:

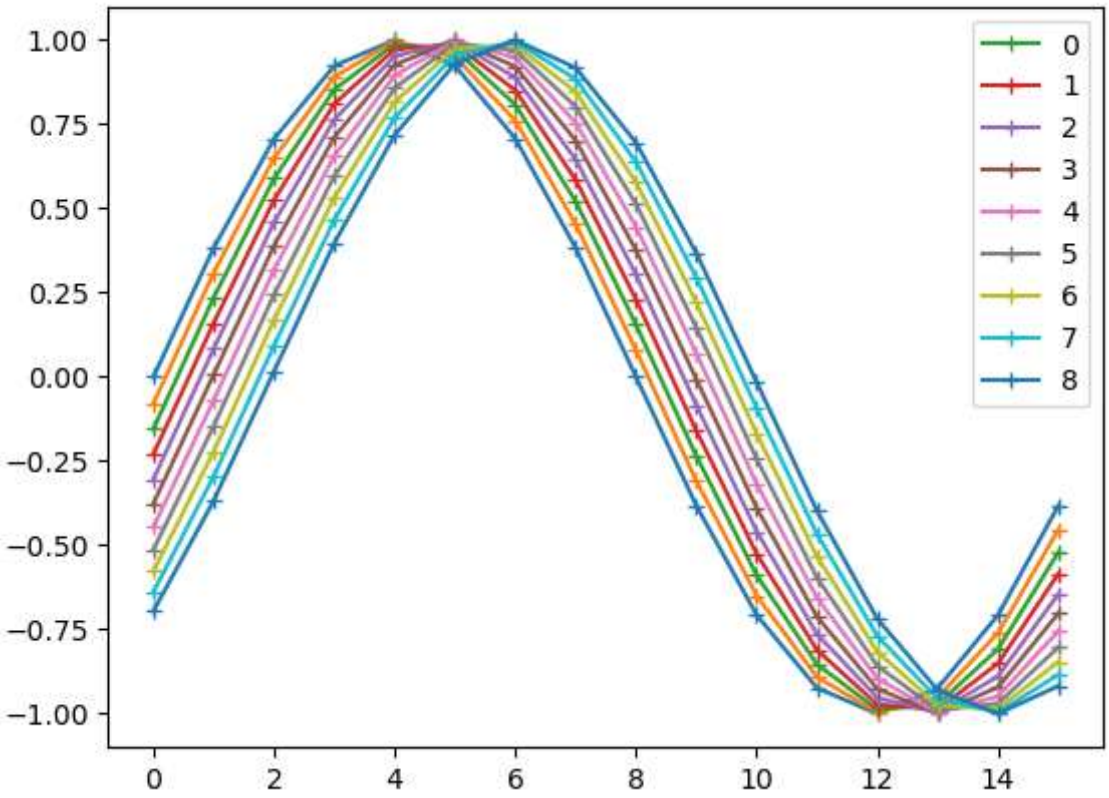
Define the derivative operator and use it to construct the time-derivative with a *second order in space* correct result

In [25]:
```
def deriv(f,ds,axis=0):
    return (np.roll(f,-1,axis)-np.roll(f,+1,axis))/(2.0*ds)

def deriv4(f,ds,axis=0):
    rollf = lambda n: np.roll(f, n, axis = axis)
    return -(8*(rollf(1) - rollf(-1)) - (rollf(2) - rollf(-2)))/(12.0*ds)

def dfdt4(f):
    return -v*deriv4(f,ds,0)

def dfdt(f):
    return -v*deriv(f,ds,0)
```

10 time steps

Let us construct an initial condition, and a *time evolution loop* that updates the function values according to our PDE describing advection

In [26]:
```
n=16
v=1.0
C=0.2
ds,x=coordinates(n)
dt=courant(C,v,ds)
f0=IC(x,v,0.0)                          # set "old time" f0 = f(0)
f1=IC(x,v,dt)                           # set "current time" f1 = f(dt)
plt.plot(f0,'-+')
plt.plot(f1,'-+')
for it in range(9):
    f2=adams_bashforth(f0,f1,dfdt,dt) # calculate new time f(t+dt)
    f0=f1                               # set "old time" f0 == f(t)
    f1=f2                               # set "current time" f1 == f(t+dt) = f2
    plt.plot(f1,'-+',label=it)
plt.legend();
```



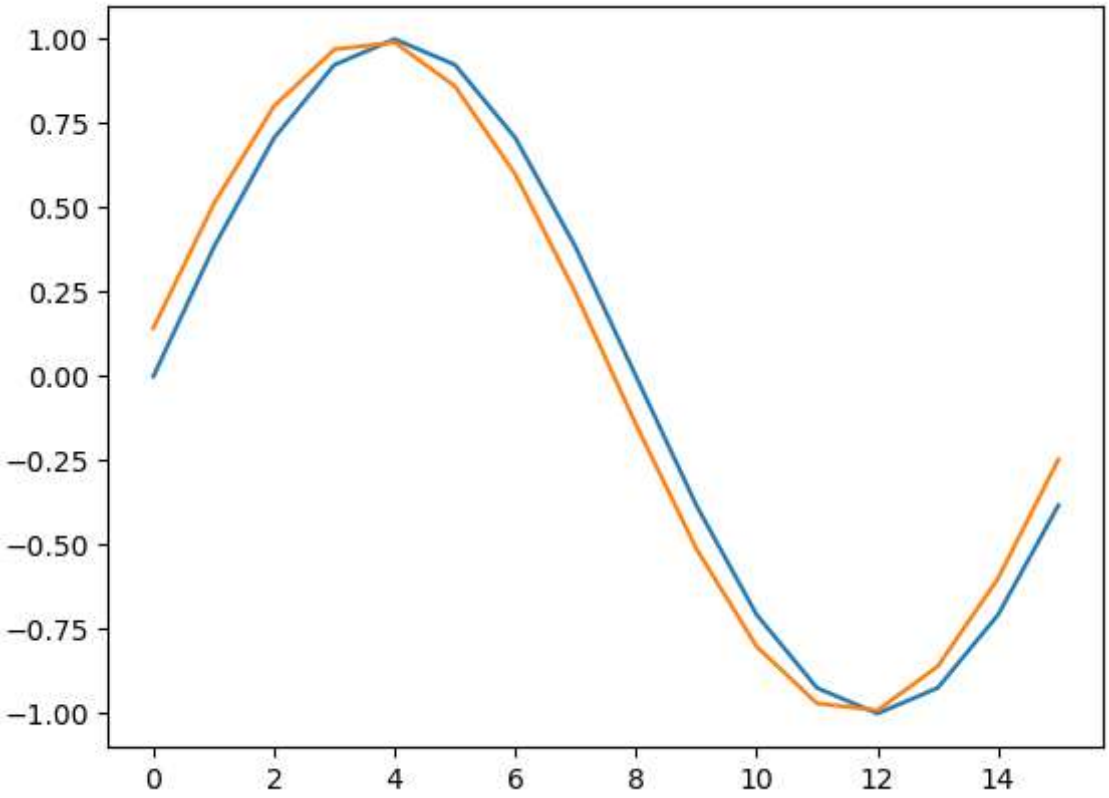
So, this seems to work pretty well, right? Looking at the plus symbols, we can see that the sine wave has moved about 2 cells, which is consistent with 10 time steps with 0.2 of a cell per time step.

So, let's try as many steps as would be needed for a full period; i.e, `n` cells:

In [27]:
```
n=16
ds,x=coordinates(n)
dt=courant(C,v,ds)
f0=IC(x,v,0.0)                          # set "old time" f0 = f(0)
f1=IC(x,v,dt)                           # set "current time" f1 = f(dt)
```

```
plt.plot(f0)
nt=int(n/C+0.5)
print('We need {} steps'.format(nt))
t = dt
for it in range(nt-1):                # do one step less because we start at t =
    f2=adams_bashforth(f0,f1,dfdt,dt)
    f0=f1
    f1=f2
    t += dt
print("End time / 2 pi:", t / (2 * np.pi))
plt.plot(f2);
```

```
We need 80 steps
End time / 2 pi: 1.000000000000002
```



Close, but not exact, so let's compute a measure of the error (slightly different from last exercise, namely the RMS error):

In [28]:
```
f0 = IC(x,v,0.0)
RMS2 = np.average((f0-f2)**2)**0.5
print('The root-mean-square error is {:.6f}'.format(RMS2))
```

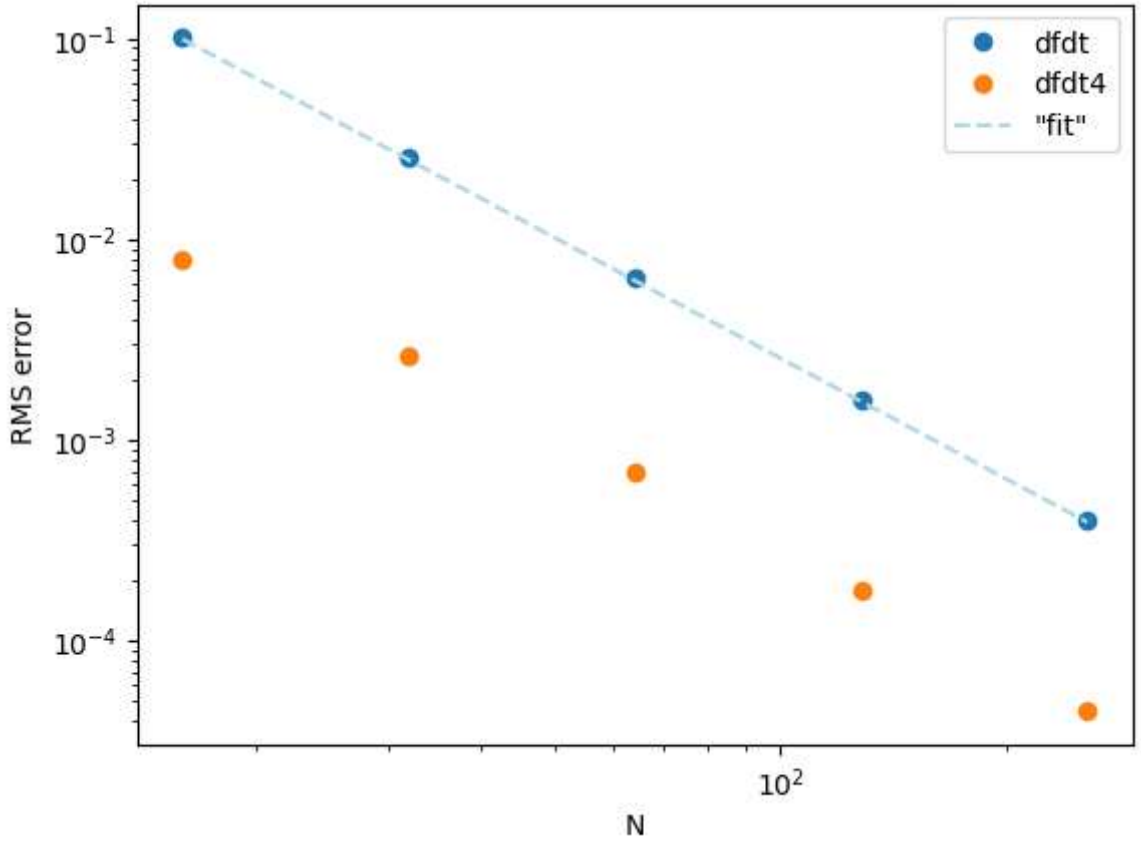
```
The root-mean-square error is 0.101364
```

Now it's interesting to see how the error varies with the number of points `n`:

In [65]:
```
for df_fn in (dfdt, dfdt4):
    N=[]
    RMS2=[]
    for n in (16,32,64,128,256):
        ds,x=coordinates(n)
        dt=courant(C,v,ds)
        f0=IC(x,v,0.0)
        f1=IC(x,v,dt)
        nt=int(n/C+0.5)
```

```
        for it in range(nt-1):
            f2=adams_bashforth(f0,f1,df_fn,dt)
            f0=f1
            f1=f2
        f0=IC(x,v,0.0)
        rms=np.average((f0-f2)**2)**0.5
        N.append(n)
        RMS2.append(rms)
    plt.loglog(N,RMS2,'o', label=df_fn.__name__);

N=np.array(N)
plt.ylabel('RMS error')
plt.xlabel('N')

plt.loglog(N,0.1/(N/16.0)**2, '--',color = "lightblue",label='"fit"');

plt.legend()
```

Out[65]: <matplotlib.legend.Legend at 0x1bb0344f610>



We conclude that the error drops with $1/n^2 \sim dx^2 \sim dt^2$; i.e., the prodedure is *second order accurate in the resolution and in the timestep*.

Second order accuracy means that after advecting a function a fixed amount of time, with a timestep that is reduced according to the cell size, the error scales as $1/n^2$.

Task 1:

Replace the `deriv()` procedure with your 4th order accurate version, and repeat the computation, but store the error as `RMS4` (so you can plot both in the same plot).

- What is the accuracy order of the improved procedure?

- The answer may seem surprising (or not). Discuss!

ANSWER:

The plot can be seen above.

Hmm it seems like it is also second order!?

At first this seemed super odd, but after discussing, we have come to the conclusion that it is the time resolution that is dominating. Meaning, that even though we have an estimate that is fourth-order in space, the error is still bounded by the Δt^2-resolution

Errors become visible when advecting a square wave

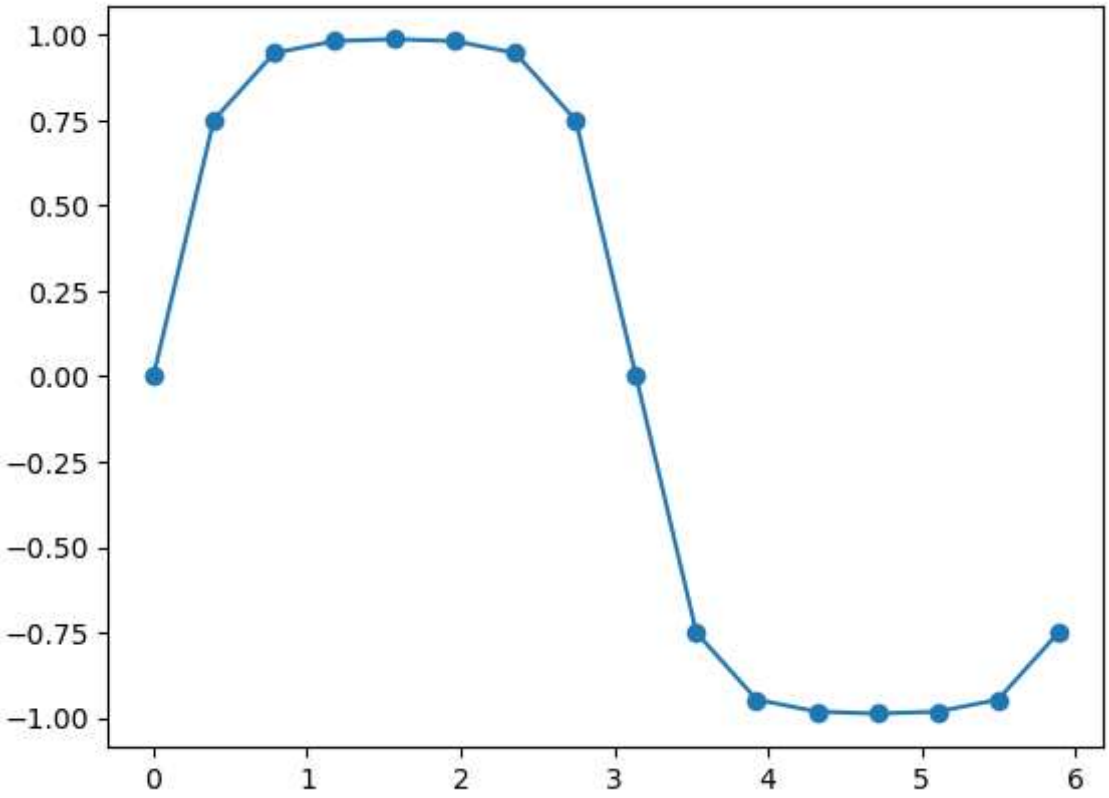
So, this seems rather good, then -- one can get very small advection errors, by choosing high spatial order. But the problem is the dependence of the errors on the number of points in the sine wave. One can think of the error as an error in *phase* of the sine wave (there is also an error in amplitude, but the error in phase is more apparent in this example).

Errors in amplitude and phase of sin/cos waves translate to *dispersion* of shapes that has "overtones"; i.e., functions whose Fourier transform contains higher wave numbers because the relative resolution of the different wave numbers.

To illustrate *dispersion*, let's make a square(-ish) wave, for testing:

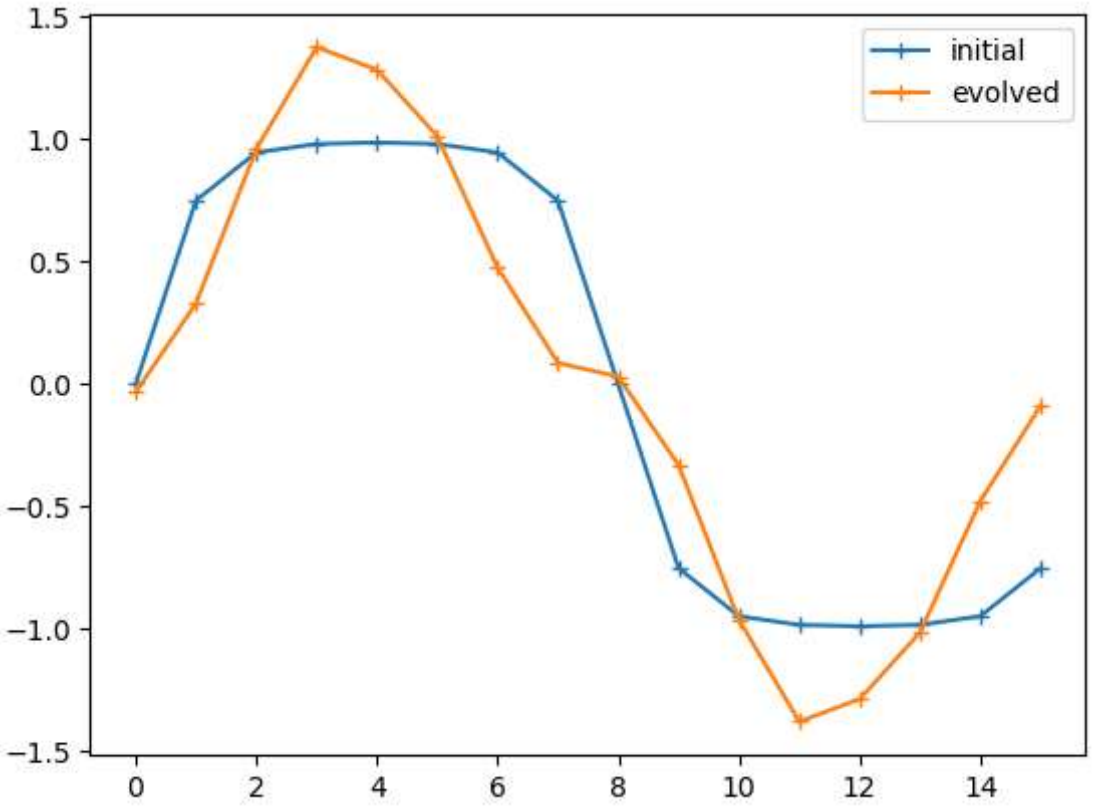
Such a wave may be though of as a superposition of sine-waves. A *Fourier transform* would give the individual amplitudes

In [31]:
```
n=16
ds,x=coordinates(n)
def square_wave(x,v,ds,t,steepness=1.0):
    f=IC(x,v,t)
    return np.tanh((steepness/ds)*f)
f0=square_wave(x,v,ds,0.0)
plt.plot(x,f0,'-o');
```



We try to evolve it for 10 time steps again, using a copy & paste of what we did before, just with the `square_wave()` as the initial condition, and plotting only the initial and final states:

In [32]:
```
n=16
C=0.2
ds,x=coordinates(n)
dt=courant(C,v,ds)
nt=int(n/C+0.5)
f0=square_wave(x,v,ds,0.0)
f1=square_wave(x,v,ds,dt)
plt.plot(f0,'-+',label='initial')
for it in range(nt-1):
    f2=adams_bashforth(f0,f1,dfdt,dt)
    f0=f1
    f1=f2
plt.plot(f2,'-+',label='evolved')
plt.legend();
```

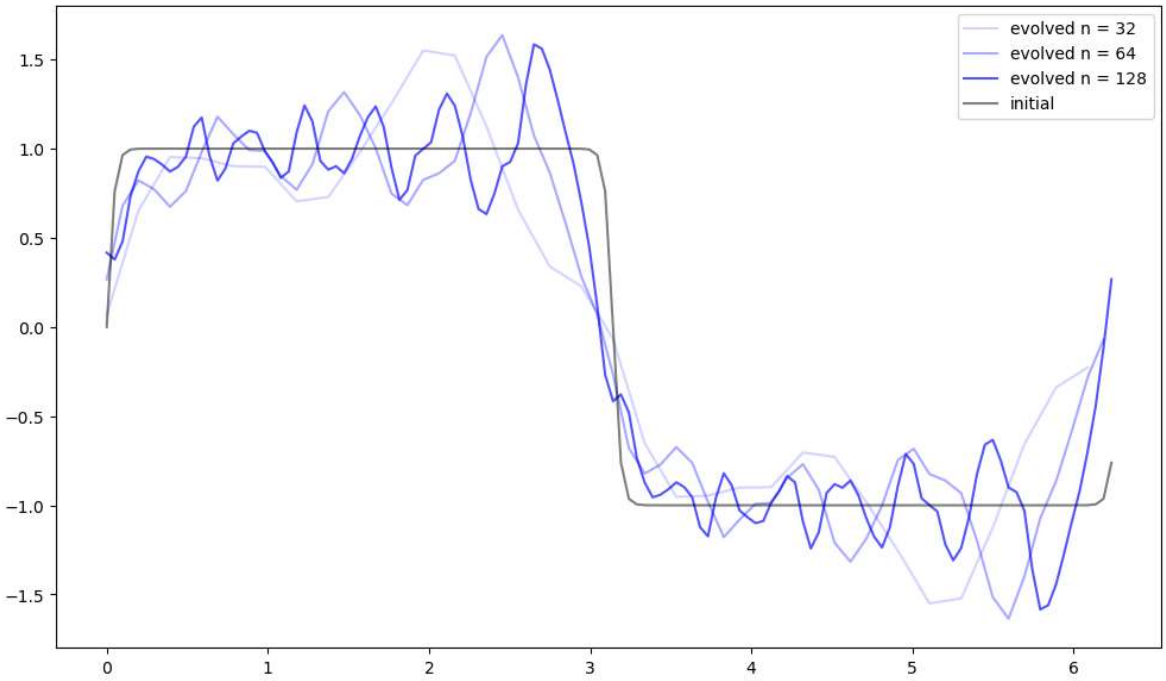


The "square" wave has already become pretty distorted!

Let's repeat this for n=32

The error at the front of the square wave looks exactly the same! Try with `n=64` and `n=128`, to confirm!

In [55]:
```
fig = plt.figure(figsize=(12, 7))

for n in (32, 64, 128):
    C=0.2
    ds,x=coordinates(n)
    dt=courant(C,v,ds)
    nt=int(n/C+0.5)
    f0=square_wave(x,v,ds,0.0)

    f1=square_wave(x,v,ds,dt)
    for it in range(nt-1):
        f2=adams_bashforth(f0,f1,dfdt,dt)
        f0=f1
        f1=f2
    plt.plot(x, f2,'-',label='evolved n = {}'.format(n), c = "blue", alpha = (n)

    if n == 128:
        f0=square_wave(x,v,ds,0.0)
        plt.plot(x, f0,'-',label='initial', c = "black", alpha = 0.5)

plt.legend();
```

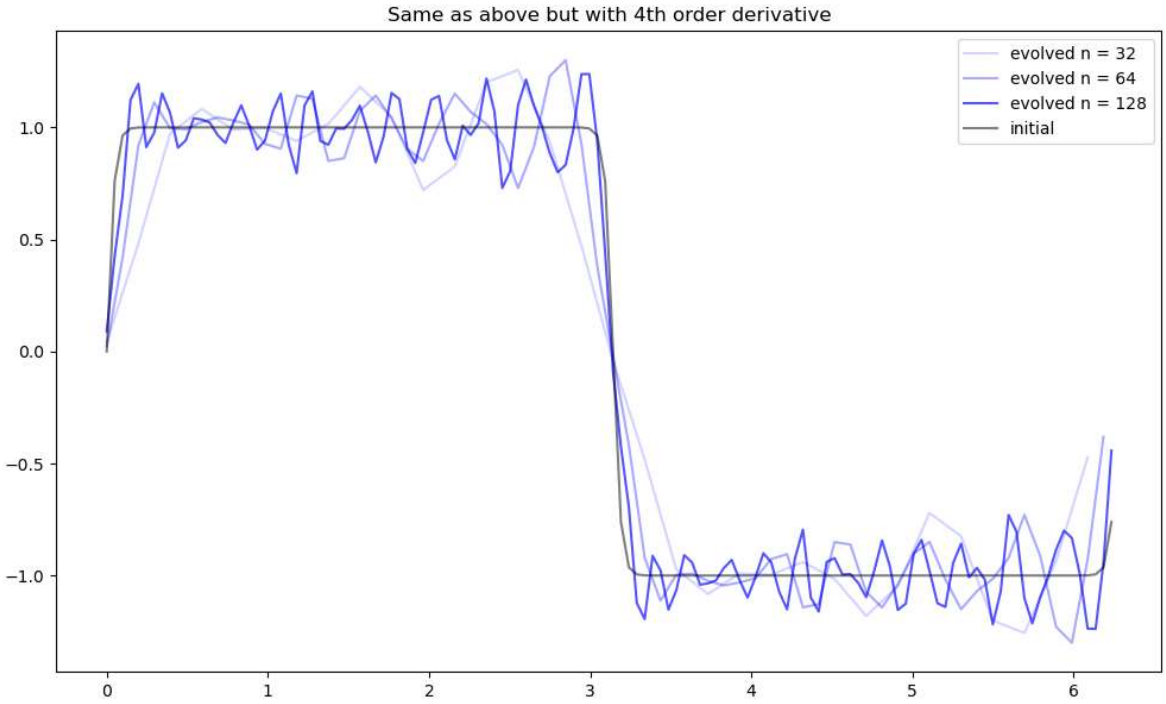


The result is not encouraging; the error remains essentially the same, which shows that the problem cannot be solved by increasing the number of points.

Task 2: Try using the 4th order derivative

Try, as above, to replace the 2nd order derivative with the 4th order derivative. Does the problem go away? Why did we have a problem in the first place (*hint*: think of a square wave as a sum of waves with many different wavelengths)

In [58]:
```
fig = plt.figure(figsize=(12, 7))

for n in (32, 64, 128):
    C=0.2
    ds,x=coordinates(n)
    dt=courant(C,v,ds)
    nt=int(n/C+0.5)
    f0=square_wave(x,v,ds,0.0)

    f1=square_wave(x,v,ds,dt)
    for it in range(nt-1):
        f2=adams_bashforth(f0,f1,dfdt4,dt)
        f0=f1
        f1=f2
    plt.plot(x, f2,'-',label='evolved n = {}'.format(n), c = "blue", alpha = (n)

    if n == 128:
        f0=square_wave(x,v,ds,0.0)
        plt.plot(x, f0,'-',label='initial', c = "black", alpha = 0.5)

plt.title("Same as above but with 4th order derivative")
plt.legend();
```



Thinking about fourier-transforming a horizontal line, it is clear to see, that a square wave is composed by many different waves with different wave lengths and frequencies. When the error goes into the specific waves, they might get different speeds, which gives a clear dispersion, and is the source of this wiggling we see above. We also figured, that the almost discontinous edges of the square wave might be additional sources of amplification for the nummerical error.

Absalon turn-in:

Answer the questions in Task 1 and 2 in the notebook and upload the notebook together with a PDF file of the notebook.