1c VanLeer Advection (Bodenheimer et al, Section 6.3.2)

At the end of **1b Advection**, we saw that a straight evaluation of the advection equation leads to problems, in the form of "ringing" at sharp transitions in the advected quantity, so let's pick up the definitions from there:

Standard libraries, and a bunch of definitions. We have a new notebook and a new python kernel now, so nothing is remembered from **1b Advection**:

In [23]:
```
import numpy as np
import matplotlib.pyplot as plt
```

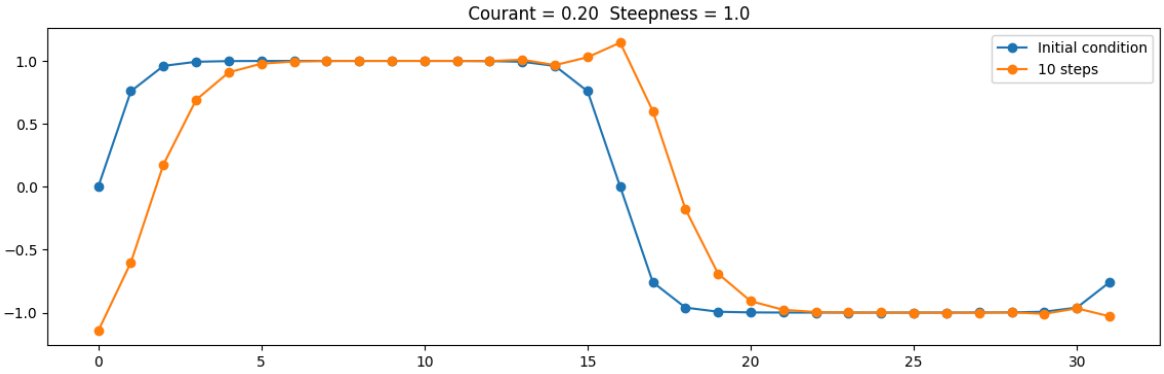
In [24]:
```
def adams_bashforth(f0, f1, dfdt, dt):
    return f1+dt*(1.5*dfdt(f1)-0.5*dfdt(f0))

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

def Courant(C, v, ds):
    """The courant number is a dimensionless number that is used to deter
    dt = C/np.max(v/ds)
    return dt

def deriv(f, ds, axis = 0, order = 2):
    return (np.roll(f,-1,axis)-np.roll(f,+1,axis))/(2.0*ds)

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

def square_wave(x, v, ds, t):
    f = IC(x, v, t)
    return np.tanh((S/ds)*f)
```

Now we can repeat the plot showing the problem, in a wide figure:

In [25]:
```
def wide_fig(title='',s=''):
    if title=='':
        title='{}  Courant = {:.2f}  Steepness = {}'.format(s,C,S)
    plt.figure(figsize=(14,4))
    plt.title(title);
```

In [26]:
```
n = 32
C = 0.2
S = 1.0
V = 1.0
nt = 10
```

```
ds, x = coordinates(n)
dt = Courant(C,V,ds)
sq = square_wave(x,V,ds,0.0)
f0 = sq
f1 = square_wave(x,V,ds,dt)
wide_fig()
plt.plot(f0, '-o', label = 'Initial condition')
for it in range(nt-1):
    f2 = adams_bashforth(f0,f1,dfdt,dt)
    f0 = f1
    f1 = f2
plt.plot(f2, '-o', label = '10 steps')
plt.legend();
```



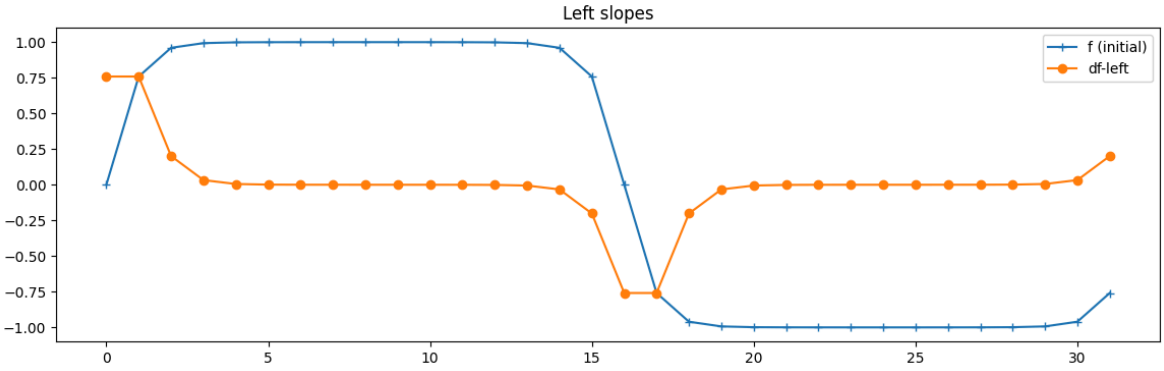
This figure above, is basically where we ended in exercise 1b. An overshoot, or ringing, due to the discontinuity that is a composite of all Fourier modes, with the problem that we cannot efficiently evolve features at the grid scale.

Implementing the VanLeer advection scheme from Section 6.3.2 in the book

The derivative of a function is the central object we are concerned with in this exercise, and much of the course. The atomic operation related to the discretized derivative is obtaining the difference between two adjacent points. We therefore define a procedure computing the "left slopes" -- the slopes connecting a point with the point on the left hand side and look at the representation for a square wave:

In [27]:
```
def left_slope(f):
    return f-np.roll(f,1)
```

In [28]:
```
f0 = square_wave(x,V,ds,0.0)
ls = left_slope(f0)
wide_fig('Left slopes')
plt.plot(f0,'-+',label='f (initial)')
plt.plot(ls,'-o',label='df-left')
plt.legend();
```

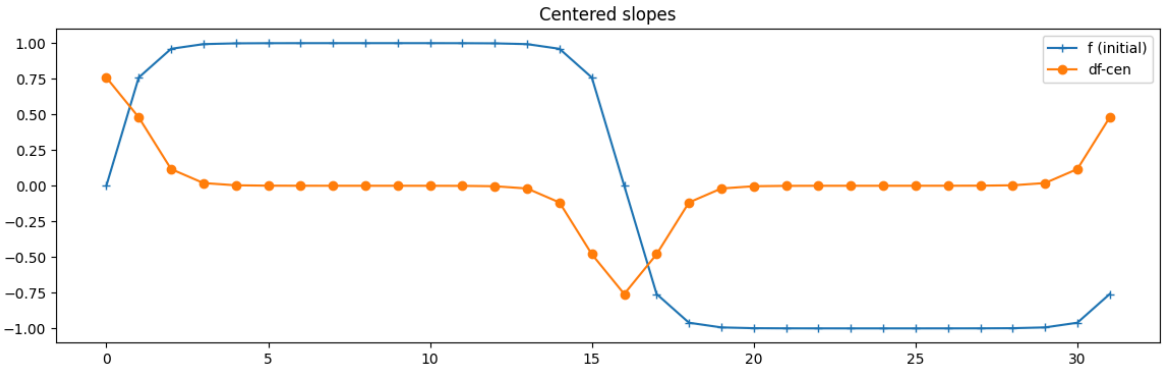


Cell-centered slopes, with no limiter

Next, we make a function that implement cell-centered slopes, equal to the average of the left and right slopes (the "right slope" connects a point with the next point to the right, and we can get them by simply rolling the left slopes one step):

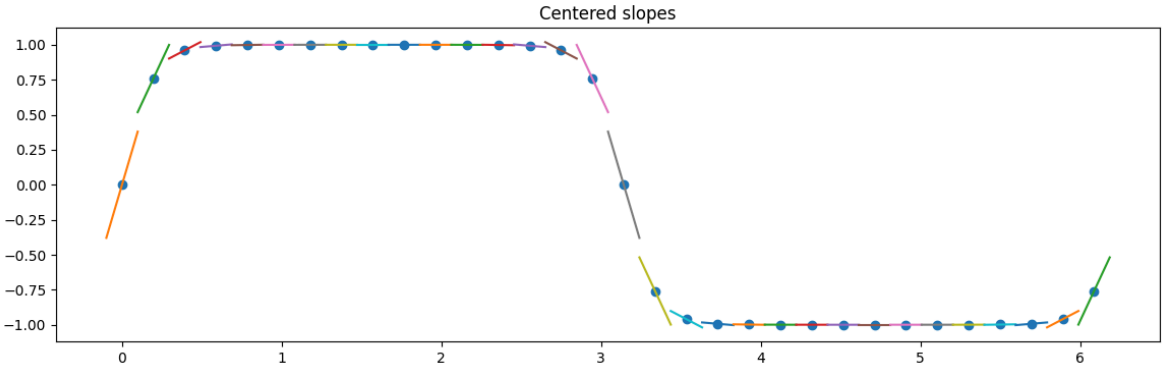
$$\Delta f_{\rm Cen} = f(x+\Delta x) - f(x-\Delta x) = \frac{1}{2}[(f(x+\Delta x) - f(x)) + (f(x) - f(x -$$

In [29]:
```
def Cen(ls):
    rs = np.roll(ls,-1)
    return (ls+rs)*0.5

wide_fig('Centered slopes')
f0 = square_wave(x, V, ds, 0.0)
ls = left_slope(f0)
cs = Cen(ls)
plt.plot(f0,'-+',label='f (initial)')
plt.plot(cs,'-o',label='df-cen')
plt.legend();
```



To better appreciate these slopes, we can plot short lines with these slopes, through the points. The main thing to notice here is that in places where the slope change rapidly, e.g. where the higher derivatives are large, the central slope over / under estimates the slope to either the left or right hand side to such a degree that any transport of a feauture will start to create wiggles -- ringing. This is very apparent when plotting slopes as lines exactly matching how a central slope would extrapolate the function values to the interface between two cells / the mid point bewteen two points.

In [30]:
```
wide_fig('Centered slopes')
f = f0
cs = Cen(ls)
plt.plot(x,f,'o')
for i in range(n):
    xx=[x[i]-ds*0.5,x[i]+0.5*ds]
    yy=[f[i]-0.5*cs[i],f[i]+0.5*cs[i]]
    plt.plot(xx,yy)
```

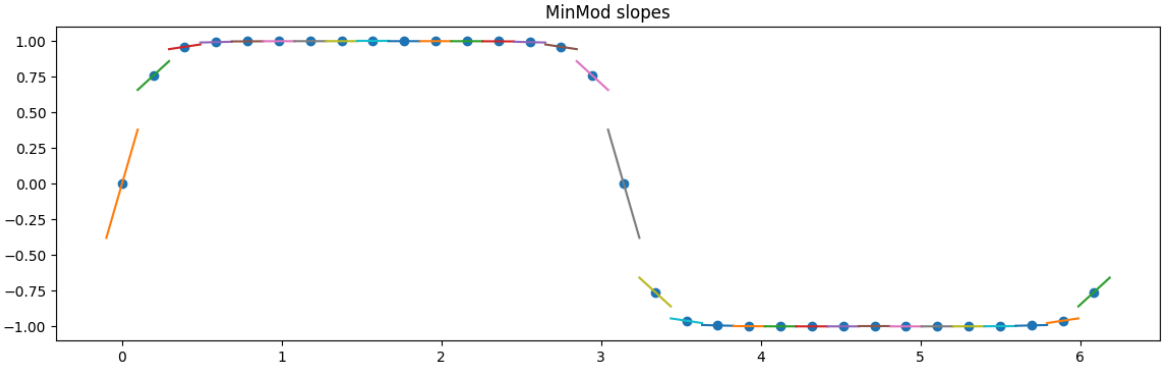


MinMod cell-centered slopes

One of the first improvements that was invented (cf. the VanLeer 1977 reference in the book) was to use the so called *MinMod* slope, which is the smallest of the left and right slopes ("smallest" disregarding sign). In addition, one sets (rather arbitrarily) the slope to zero, when the left and right slopes differ in sign:

In [31]:
```
def MinMod(ls):
    rs = np.roll(ls,-1)
    sign = (np.sign(ls) + np.sign(rs)) // 2     # Gives 0 if sign differs
    return np.minimum(abs(ls),abs(rs))*sign
```

In [32]:
```
wide_fig('MinMod slopes')
plt.plot(x,f,'o')
cs = MinMod(ls)
for i in range(n):
    xx=[x[i]-ds*0.5,x[i]+0.5*ds]
    yy=[f[i]-0.5*cs[i],f[i]+0.5*cs[i]]
    plt.plot(xx,yy)
```

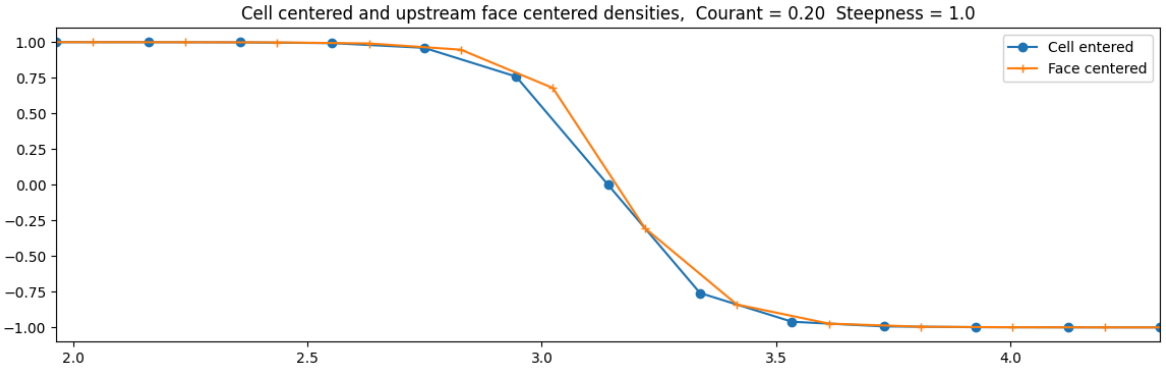


Upstream cell face values of the density

The advection method, by Van Leer, uses slopes inside the upstream cell to estimate the time averaged interface density `d_lf`, which is then multiplied by the (negative) velocity, to get the time derivative. Below we plot a small subsection centered on the transition region, instead of the full 2π interval.

In [33]:
```
dt = Courant(C,V,ds)
d_lf = np.roll(f + (ds-V*dt)*cs/ds*0.5,1)       # The flux through the le
```

In [34]:
```
wide_fig(s='Cell centered and upstream face centered densities,')
plt.plot(x, f0, '-o', label = 'Cell entered')
plt.plot(x-0.5*ds-0.5*V*dt, d_lf, '-+', label = 'Face centered')
plt.legend()
n1 = int(6.0/S)
n2 = n//2
xl = (x[n2-n1],x[n2+n1])
plt.xlim(xl);
```



Take a moment to figure out, with the help of the plot, what is meant by "upstream": If one has such a piecewise linear representation ("reconstruction" is a term often used), then one can

1. figure out the value at the interface between two cells, at time t, and
2. figure out the value at the interface at time t+dt (by following the slope)

But what one really wants is the average value over time, and since the slope is linear, it is the mean of the two, and that -- again -- is the same as the value at the time `dt/2`, if the problem is linear (as it is, in this approximation).

Alternatively, one can think of it as figuring out what is the slope inside the cell, and then see how much is transported across the cell interface in a timestep `dt`. This is a *triangle*, and the average height of a triangle is exactly ad the midpoint -- `dt/2`

Got it? Otherwise, revisit the book and the slides.

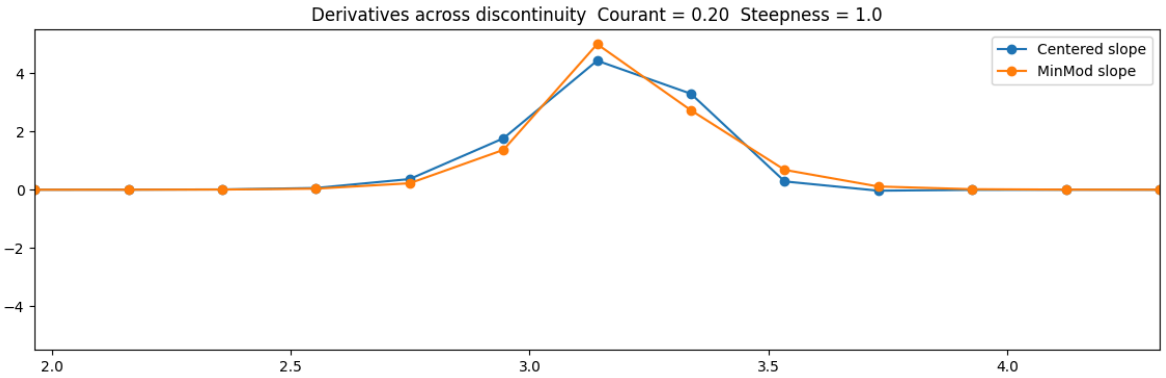
Collect steps into a time derivative function

The slope is computed by the function `Slope`, which may be set to `Cen` or `MinMod`. It takes as an input the left slope `ls`. Notice how further down, when computing the flux across the interface this slope is divided by ds to get the derivative, and therefore the flux. The flux is the (positive) change of values through

an interface per length. To find the change in a cell, we have to add what is coming in from the left and subtract what is leaving to the right.

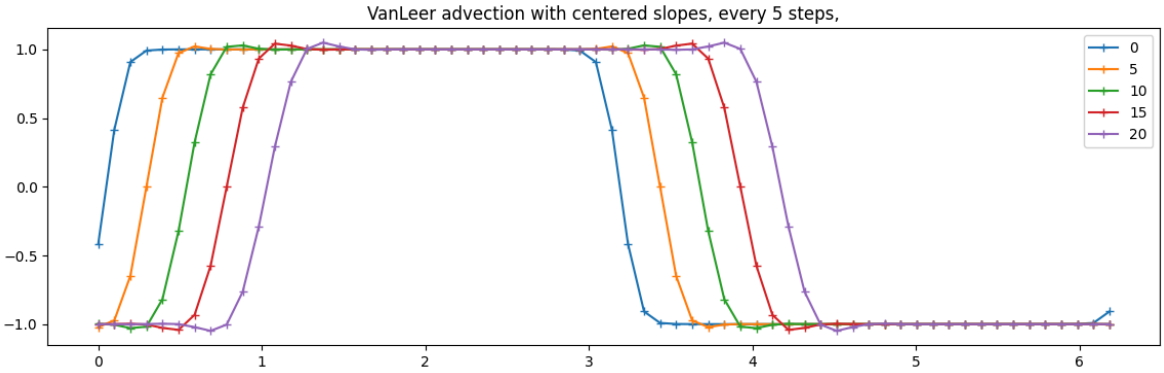
In [35]:
```
def dfdt(f):
    ls = left_slope(f)
    cs = Slope(ls)
    flux = V*(f + (ds-V*dt)*cs/ds*0.5)
    return (np.roll(flux,1)-flux)/ds
```

In [36]:
```
S = 1.0
sq=square_wave(x, V, ds, 0.0)
wide_fig(s='Derivatives across discontinuity')
for Slope in (Cen,MinMod):
    if (Slope==MinMod):
        label='MinMod slope'
    if (Slope==Cen):
        label='Centered slope'
    plt.plot(x, dfdt(sq), '-o', label = label)
    plt.xlim(xl);
plt.legend();
```



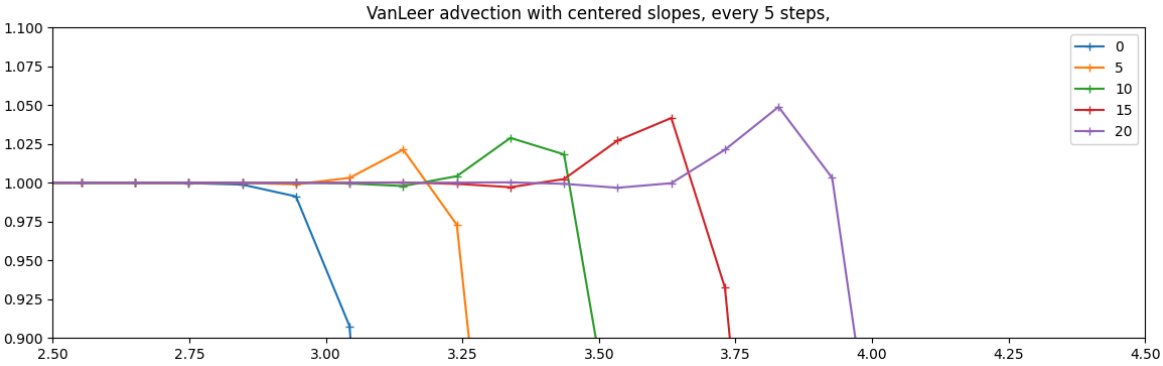
First order time stepping

In [37]:
```
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

wide_fig('VanLeer advection with centered slopes, every 5 steps,')
Slope = Cen
for i in range(21):
    f = f+dt*dfdt(f)
    if i%5==0:
        plt.plot(x,f,'-+',label=i)
plt.legend();
```



It doesn't look all that bad, but one can see a weak tendency for "overshoot", especially after zooming in:

In [38]:
```
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

wide_fig('VanLeer advection with centered slopes, every 5 steps,')
Slope=Cen
for i in range(21):
    f = f+dt*dfdt(f)
    if i%5==0:
        plt.plot(x,f,'-+',label=i)
plt.xlim(2.5,4.5); plt.ylim(0.9,1.1)
plt.legend();
```

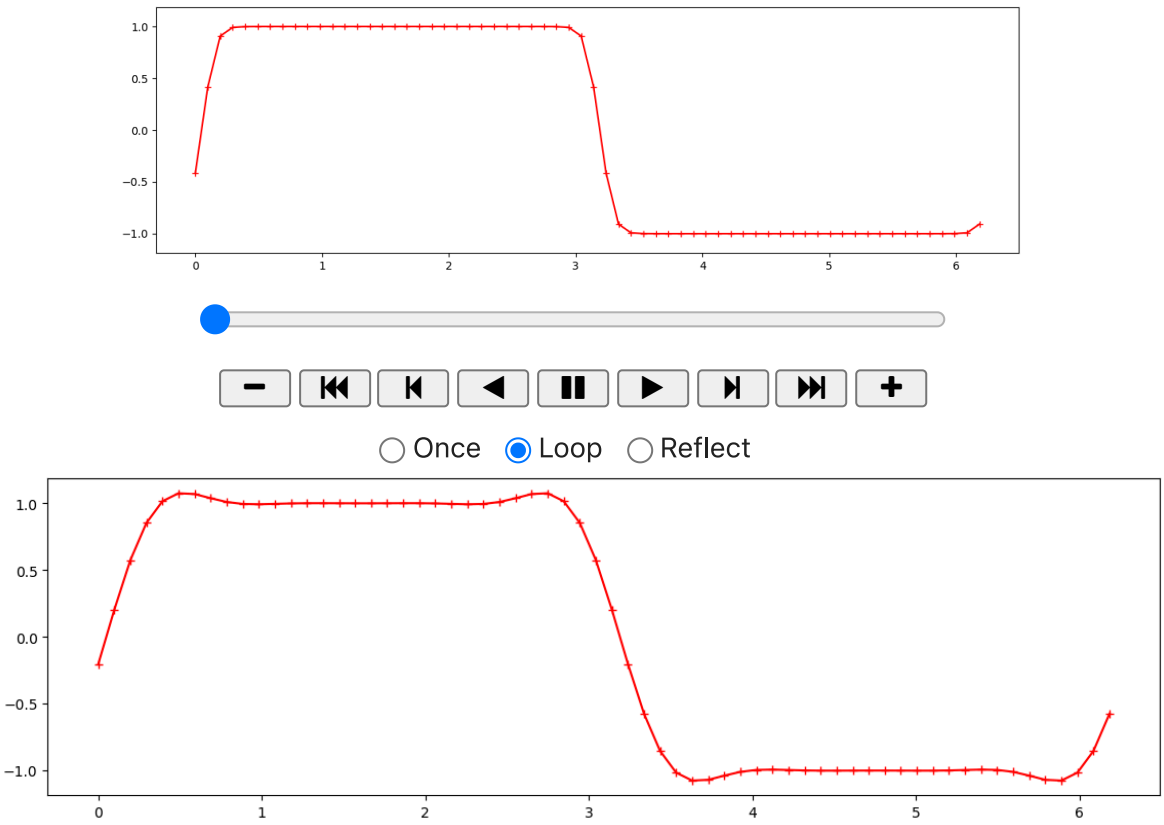


We can also try to make a movie which illustrates the progressive degradation of the square wave with time

In [39]:
```
from matplotlib import animation
from IPython.display import HTML

n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

fig = plt.figure(figsize=(14,4))

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
```

```
# each frame
ims = []

#wide_fig('VanLeer advection with Cen slopes, every 5 steps,')
Slope=Cen
for i in range(2*n+1):
    f=f+dt*dfdt(f)
    # plot returns a list of plots (one per (x,y) pair)
    # writing "im, " unpacks this one element list to a single element
    im,  = plt.plot(x,f,'-+',animated=True,color='r')
    ims.append([im])

# create animation object
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# render animation javascript widget
HTML(anim.to_jshtml())
```

Out[39]:



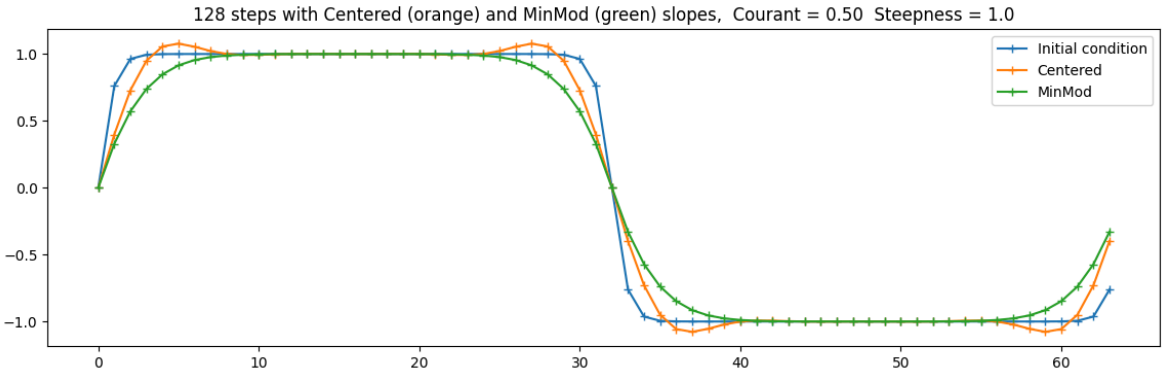
Compare Cen and MinMod slopes after a full period

After a full period, the overshoot has become clearly visible for the *Cen* (central, simple derivative) slopem while the *MinMod* suppress ringing at the cost of diffusion:

In [40]:

```
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
sq = square_wave(x,V,ds,0.0)

nt = int(n/C+0.5)
wide_fig(s='{} steps with Centered (orange) and MinMod (green) slopes,'.f
```

```
plt.plot(sq,'-+',label='Initial condition')
for Slope in (Cen, MinMod):
    f = np.copy(sq)
    for i in range(nt):
        f = f+dt*dfdt(f)
    if (Slope==MinMod):
        label = 'MinMod'
    if (Slope==Cen):
        label = 'Centered'
    plt.plot(f,'-+',label = label)
plt.legend();
#plt.savefig("advection_vl.png")
```



The `savefig` command write the figure out to an `advection_vl.png` file and is useful if we would like to export the plot to e.g. a document.

Task:

Implement the `MonCen` slope limiter (also called the monotonized cell-centered slope), mentioned in Section 6.3.2 in the book, and extend the comparison, so all three results are compared

Absalon turn-in

Upload the notebook with the advection figure, and a pdf of the notebook, and write a short text -- inside the notebook -- commenting on the figure

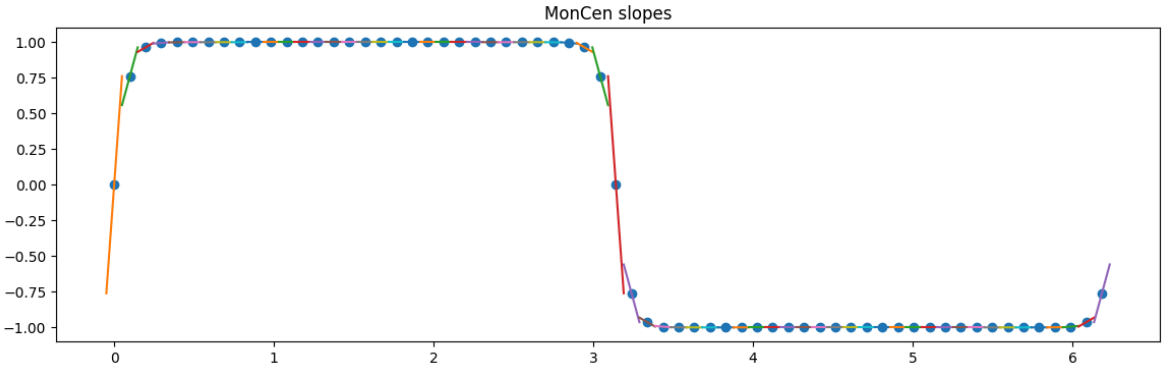
Implementation of MonCen slope limiter

The comment on the final figure can be found at the end of the notebook.

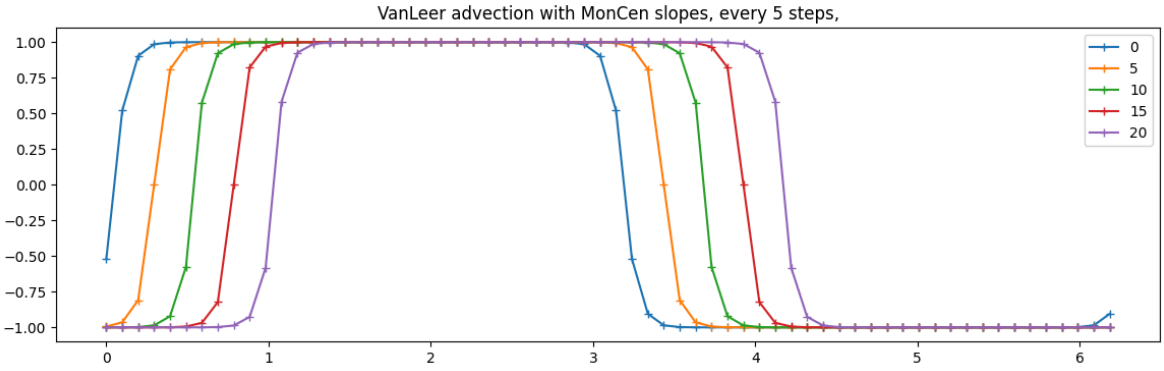
In [41]:
```
# MonCen (monotonized central-difference) slope limiter
def MonCen(ls):
    """
    Monotonic Central (MonCen) slope limiter function.

    Parameters:
        ls (array-like): Left slope values.

    Returns:
        array-like: Limited slope values.
    """
    # Right slope (roll left slope by -1)
```

```
    rs = np.roll(ls, -1)

    # Sign compatibility: zero out if signs differ
    sign = (np.sign(ls) + np.sign(rs)) // 2  # 1 if same sign, 0 otherwis

    cen = Cen(ls)

    # Compute the limited slope
    limited_slope = np.minimum(2*np.abs(ls), 2*np.abs(rs), cen)

    # Restore the sign
    return limited_slope * sign
```

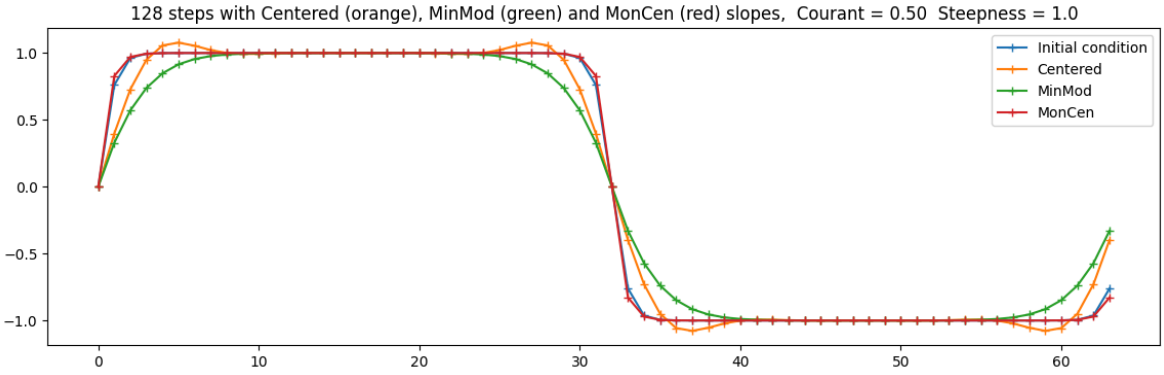
In [42]:
```
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

wide_fig('MonCen slopes')
plt.plot(x,f,'o')
ls = left_slope(f)
cs = MonCen(ls)
for i in range(n):
    xx=[x[i]-ds*0.5,x[i]+0.5*ds]
    yy=[f[i]-0.5*cs[i],f[i]+0.5*cs[i]]
    plt.plot(xx,yy)
```



In [43]:
```
# VanLeer advection with MonCen slopes
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

wide_fig('VanLeer advection with MonCen slopes, every 5 steps,')
Slope = MonCen
for i in range(21):
    f = f+dt*dfdt(f)
    if i%5==0:
        plt.plot(x,f,'-+',label=i)
plt.legend();
```



In [44]:
```
# Comparison of MinMod, Centeren and MonCen slopes
n = 64
C = 0.5
ds, x = coordinates(n)
dt = Courant(C,V,ds)
f = square_wave(x,V,ds,0.0)

nt = int(n/C+0.5)
wide_fig(s='{} steps with Centered (orange), MinMod (green) and MonCen (r
plt.plot(f,'-+',label='Initial condition')
for Slope in (Cen, MinMod, MonCen):
    f = np.copy(sq)
    for i in range(nt):
        f = f+dt*dfdt(f)
    if (Slope==MinMod):
        label = 'MinMod'
    if (Slope==Cen):
        label = 'Centered'
    if (Slope==MonCen):
        label = 'MonCen'
    plt.plot(f,'-+',label = label)
plt.legend();
```



Comment of the plot above: In the plot, we see that the *centered scheme* shows significant oscillations near discontinuities which indicates that it is not suitable for problems with sharp gradients like this. The *MinMod limiter* seems to better suppress oscillations, but then introduces noticable numerical diffusion leading to overly smoothed transitions. Finally, the *MonCen limiter* seems to give the best "balance" - like the MinMod limiter it avoids oscillations, while also persevering the sharper gradients like the centered scheme. However, the MonCen scheme seems to overshoot slightly in regions where the derivative changes more, though it does not create new extrema. Despite this, the MonCen limiter clearly performs the best overall out of the three schemes.