Assuming the following fixed parameters,

Nbar = 1.0

Nstar = 0.1432394487827058

tau\_eq = 1 microsecond

sigma0 = 0.2

when the “baseline” scenario is perturbed a little to the following:

L = 30 micrometer

D = 0.000365 micrometer \*\* 2 / microsecond

nu\_kin = 105 micrometer / second

nu\_kin\_mlyperus = 0.26992287917737784 / microsecond

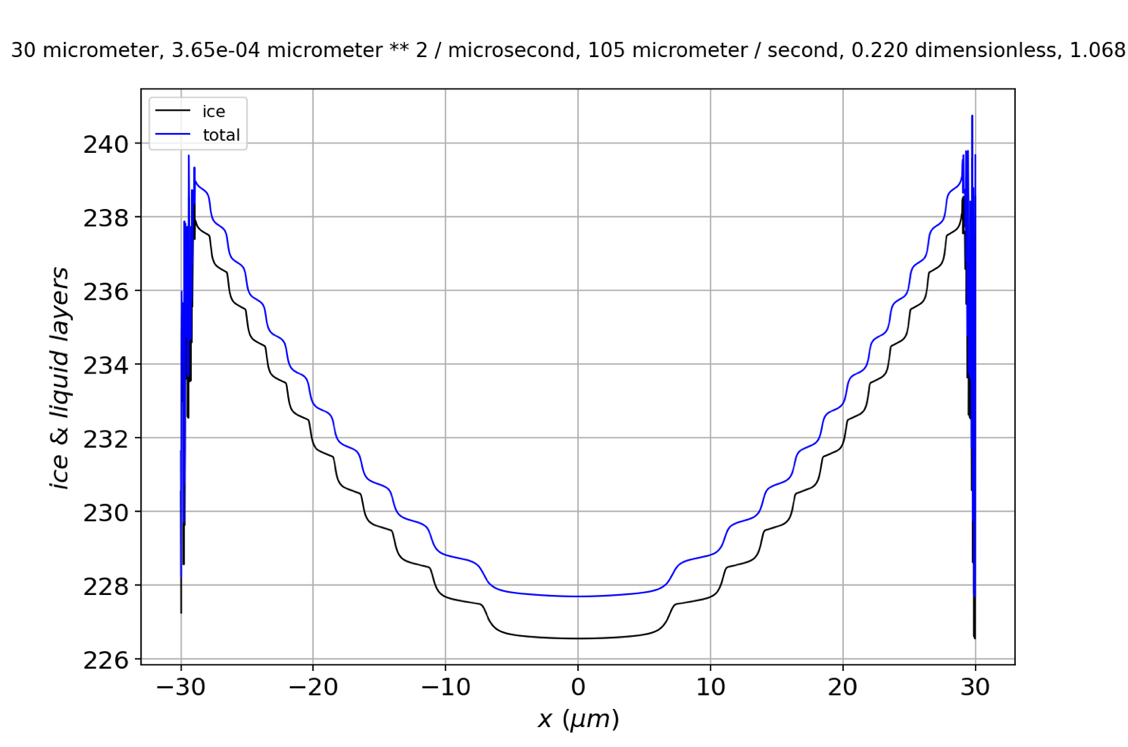
sigmaI\_corner = 0.22 dimensionless

c\_r\_percent = 1.068 dimensionless

nx (crystal) = 2401

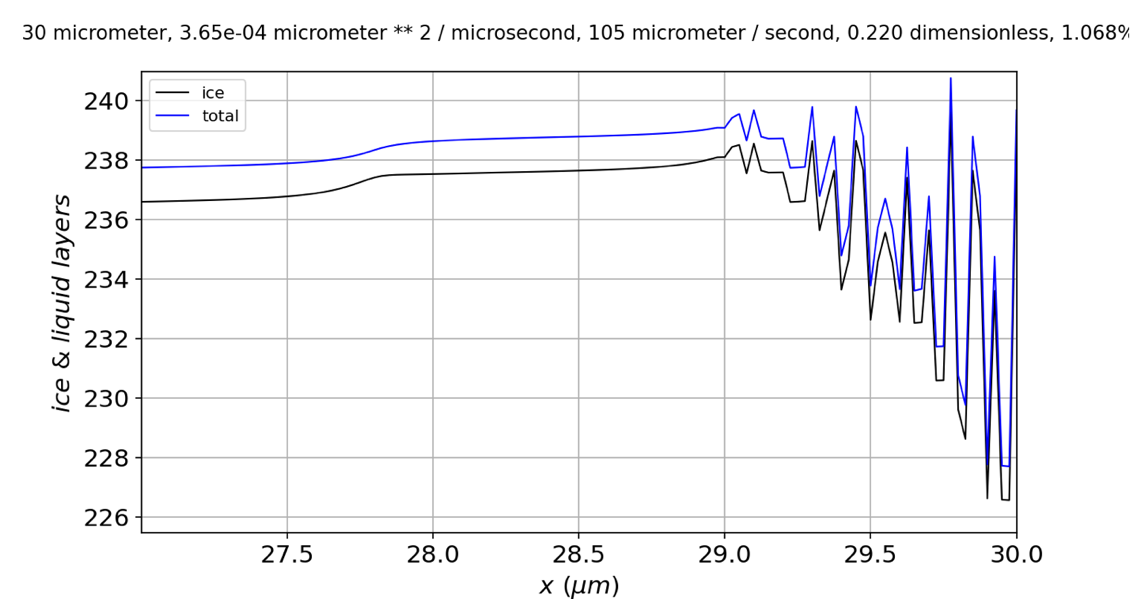
Spacing of points on the ice surface = 0.02499999999999858 micrometer

we get instability:



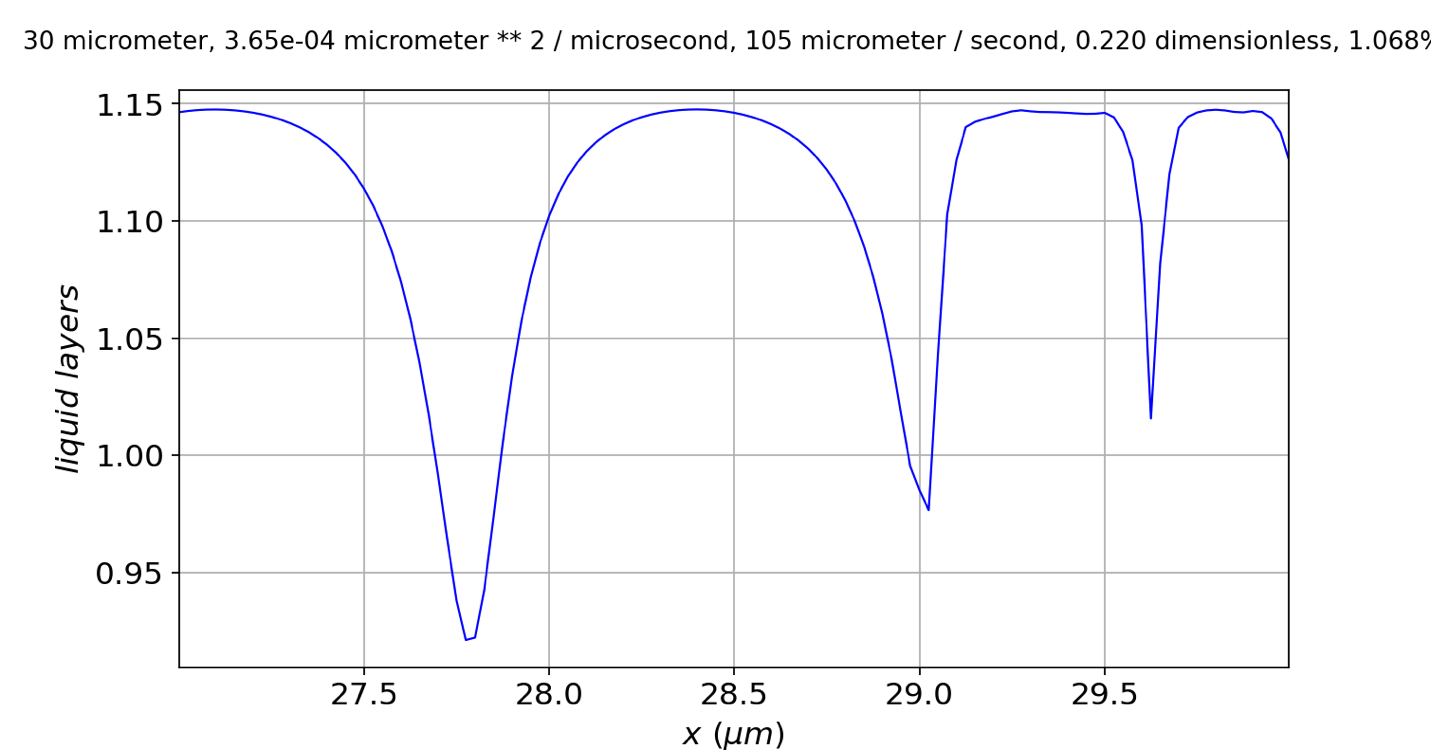
**Figure 1**. Profile showing unstable growth.

If we expand in the corner, it’s clear that instabilities have crept in, by this time, from the edge at to .



**Figure 2**. Expanded view of the profile in Fig. 1.

If we examine the alone,



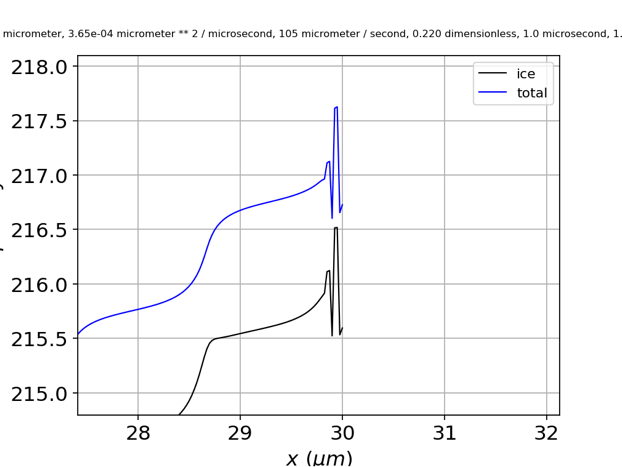
**Figure 2**. Quasiliquid thickness after onset of instability.

we see that the instability seems to originate on a *step*, rather than at a *riser* (solid arrow). The fact that tops out at risers generally (dashed arrows) with a value close to is significant, because the parameterization means that when is at its **maximum** value (), . That means, at that time step, deposition from the vapor phase has slightly exceeded the ability of the quasiliquid to **freeze** or **diffuse away**. If this thinking is correct, then we can predict that stability can be favored by bigger (which is correct) and smaller (which is incorrect).

OK, more data. **Stability is favored by**

* Smaller (from to )
* Bigger (from to )
* Bigger (from to )
* Smaller (from to )

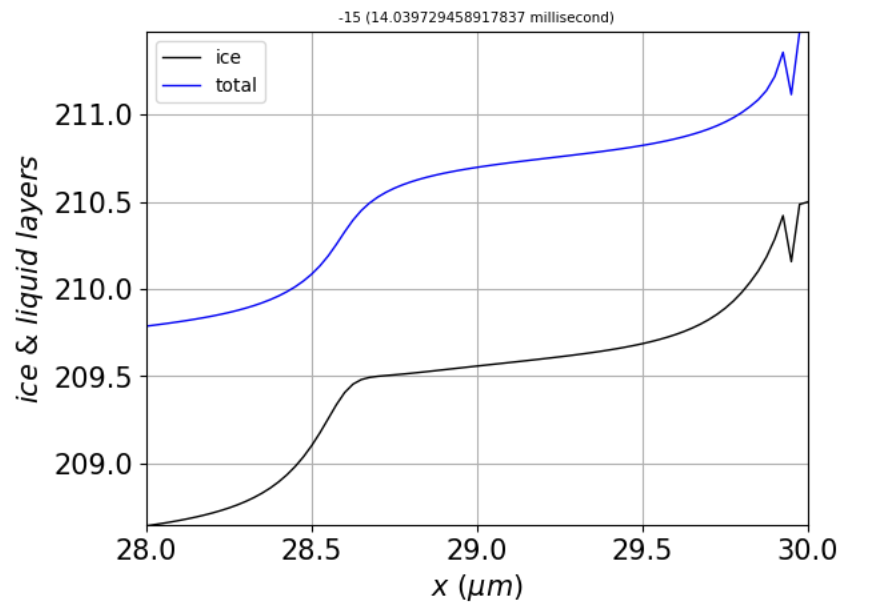
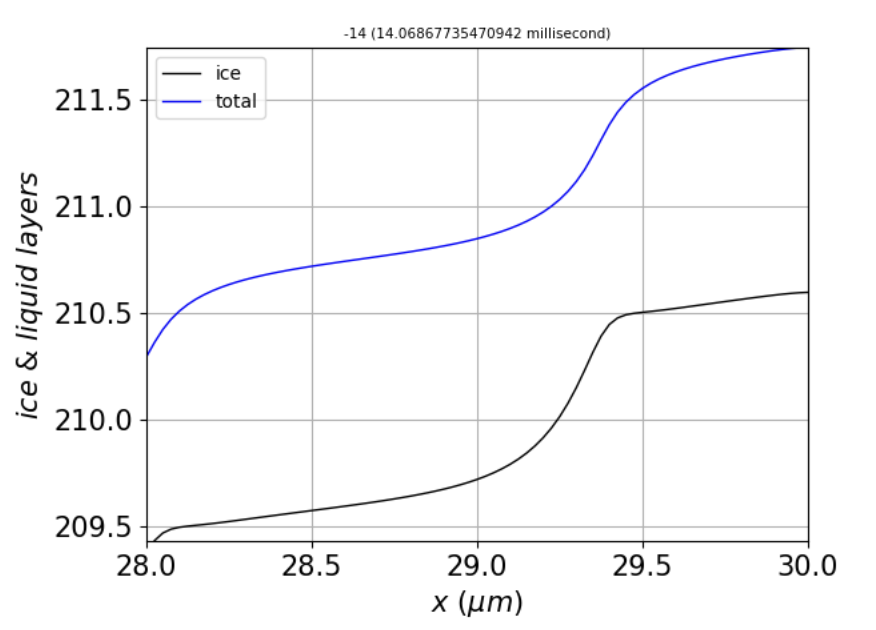
At times 14.445 and 14.446, I get small jumps in :

**Figure 3**. Evolution of a small-amplitude instability showing one glitch, then two.

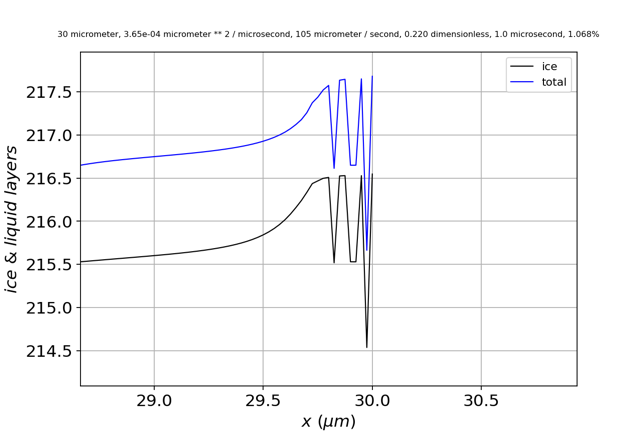
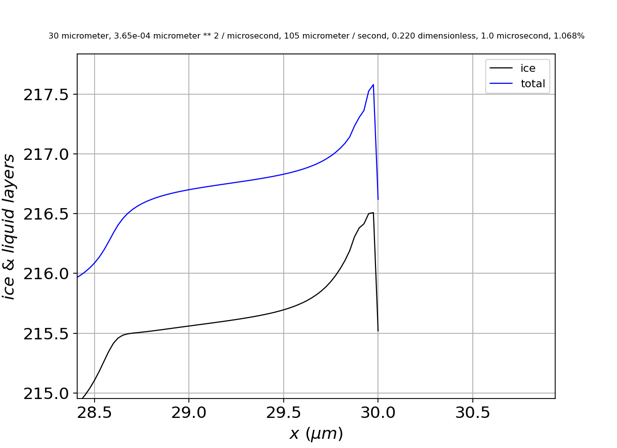
Conclusions …

1. The glitches first appear close to **facet corners**. It’s still not clear whether this has something to do with the imposition of periodic boundary conditions at facet corners, or some other property, like lambda, which happens to be small at facet corners. I think I could explore this with **different supersaturation** profiles, or with **different boundary conditions** (see below).
2. The fact that glitches (when they first appear) are smaller in than in suggests that they originate in **diffusion** or **deposition**, but **not freezing**. (See f1d\_solve\_ivp for a more detailed argument.)
3. In some cases, **glitches disappear in subsequent time steps**. An example is shown in Fig. 4: the glitch where the number of ice layers equals 210.5 (i.e., close to the riser at 30 micrometers) has disappeared at time step -14 (the riser now located closer to 29.4 micrometers).

**Figure 4**. Left: Profile at time step -15. Right: Profile one time step later, showing removal of the glitch.

1. **When glitches persist** over time, however, they grow until they reach increments of layer, then layers, and so on. An example is shown in Fig. 5.



**Figure 5**. Evolution of the small-amplitude instability in Fig. 3 to full layer instability.

Examining **different boundary conditions**: If I replace **periodic boundary conditions** with **reflection** (no-flux) boundary conditions,

# Ntot diffusion

dy = np.empty(np.shape(NQLL0))

for i in range(1,len(NQLL0)-1):

dy[i] = DoverdeltaX2\*(NQLL0[i-1]-2\*NQLL0[i]+NQLL0[i+1])

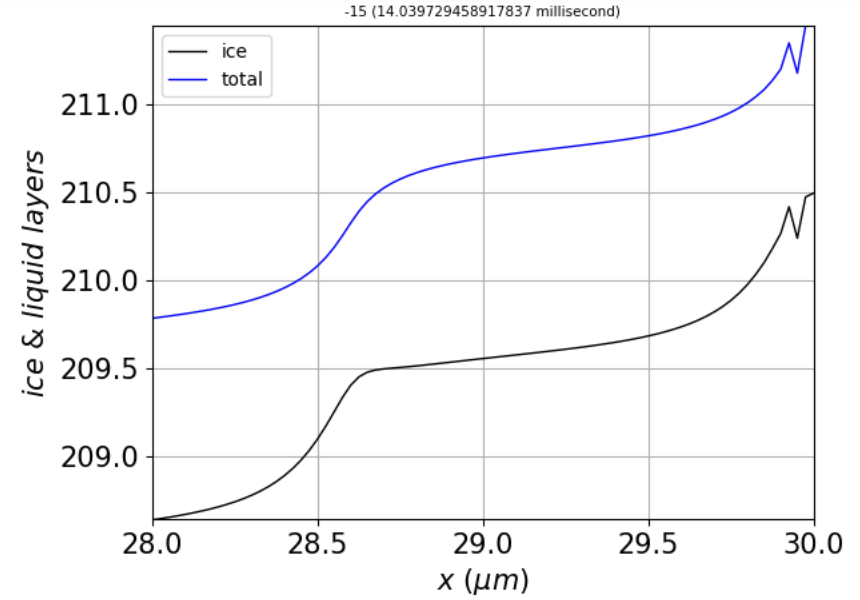
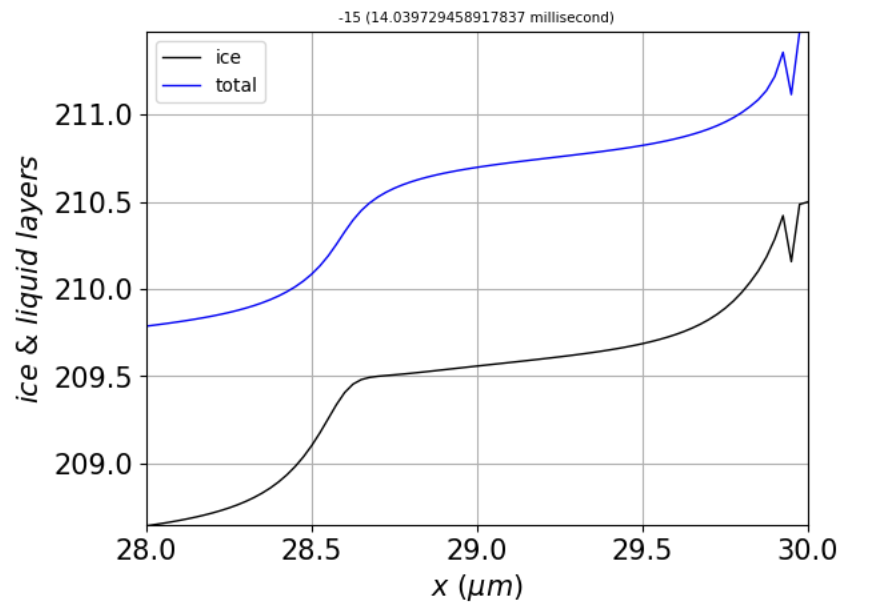
# dy[0] = DoverdeltaX2\*(NQLL0[-1] -2\*NQLL0[0] +NQLL0[1]) # Periodic BC

# dy[-1] = DoverdeltaX2\*(NQLL0[-2] -2\*NQLL0[-1]+NQLL0[0])

dy[0] = DoverdeltaX2\*(NQLL0[0] -2\*NQLL0[0] +NQLL0[1]) # No-flux BC

dy[-1] = DoverdeltaX2\*(NQLL0[-2] -2\*NQLL0[-1]+NQLL0[-1])

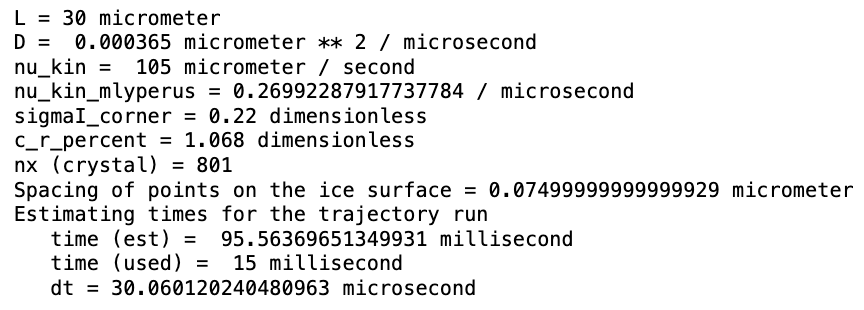
the results are a little different, but the **glitches still occur**:



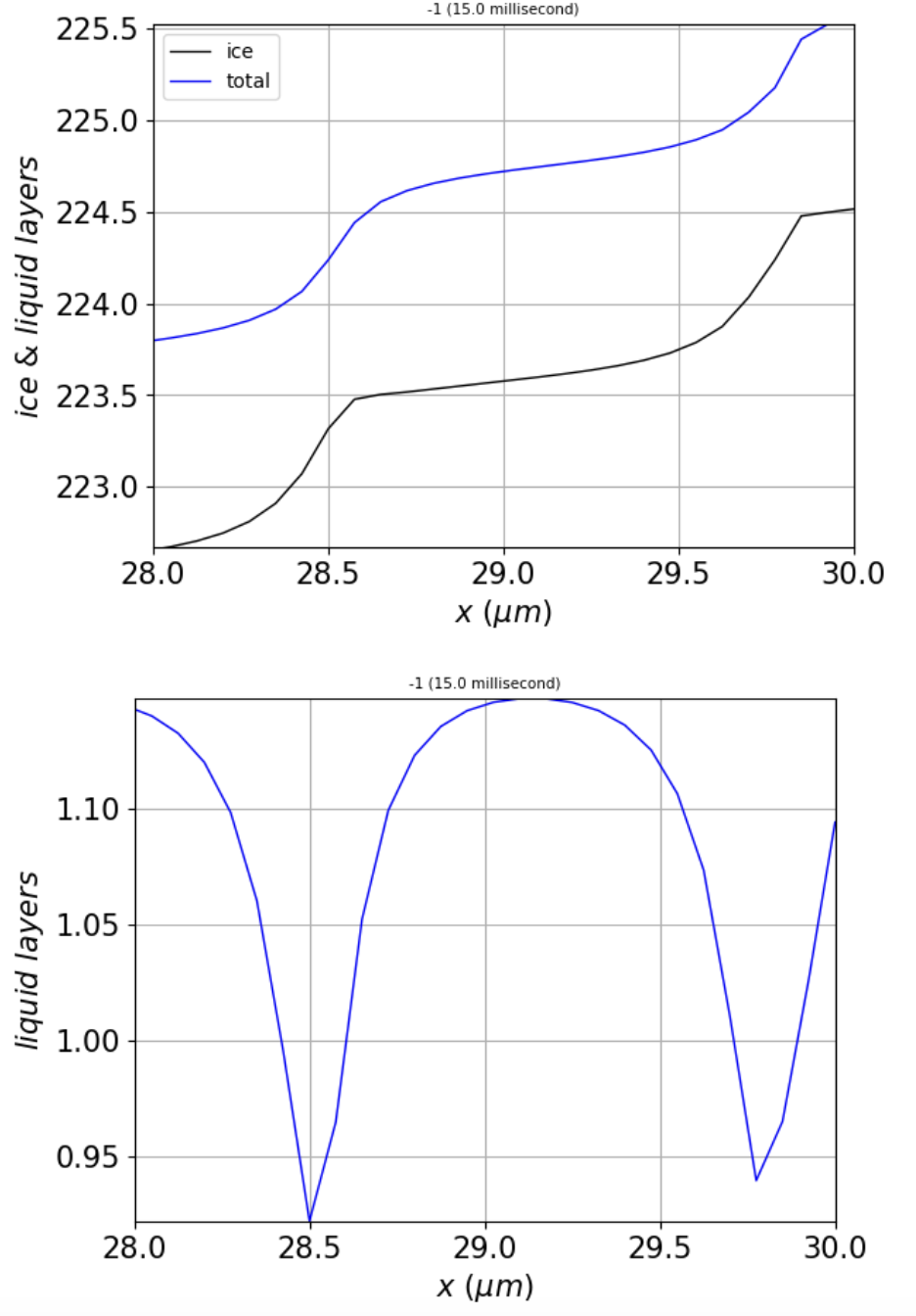
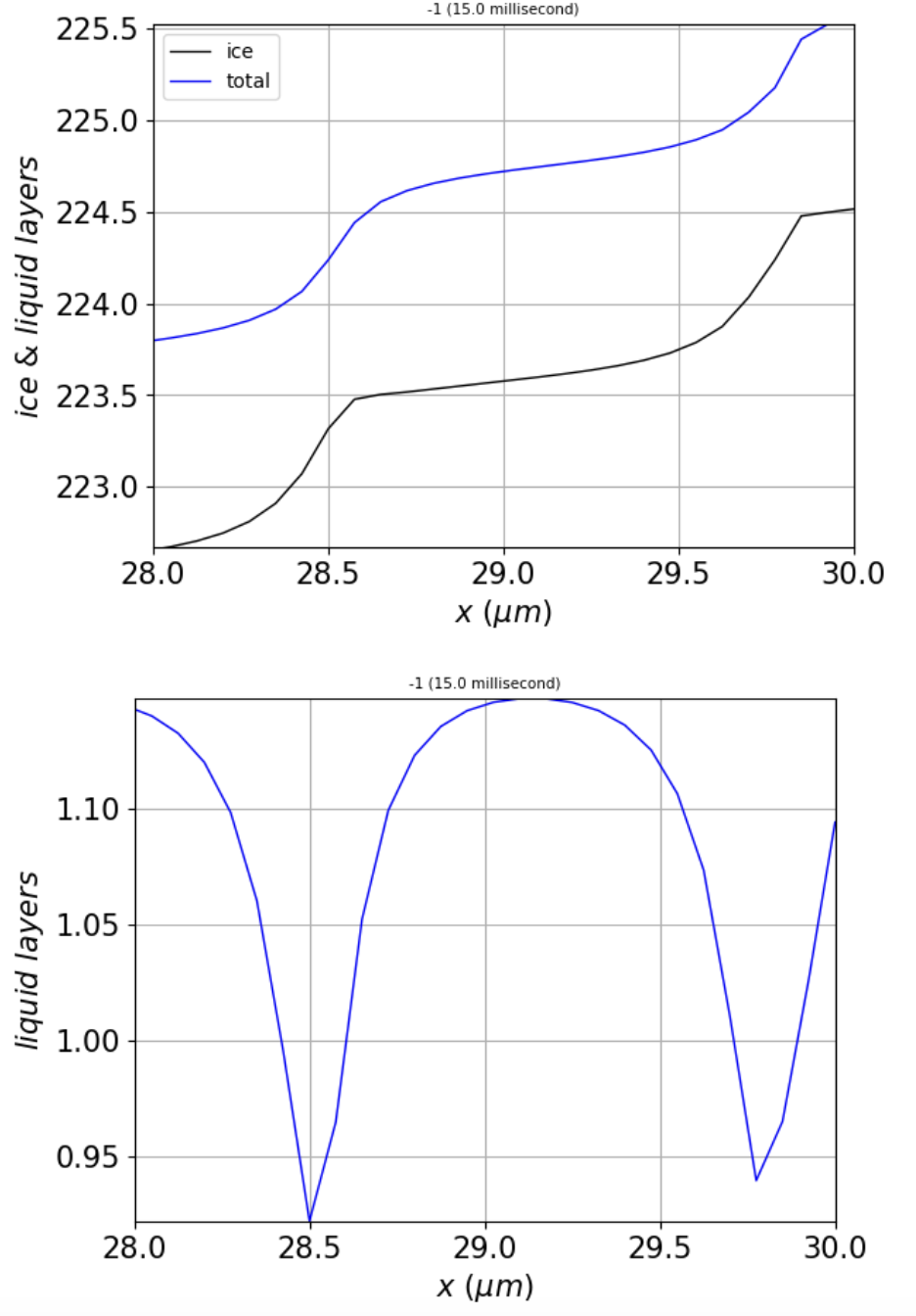
Left: **periodic** boundary conditions. Right: **reflection** (no-flux) boundary conditions.

The most this can mean, however, is that the imposition of periodic boundary conditions has been coded up correctly.

**Another thought**: what if we switch to an **implicit** integration method? They’re slower, so I had to back off on the spatial resolution (nx=801, instead of 2401 points).



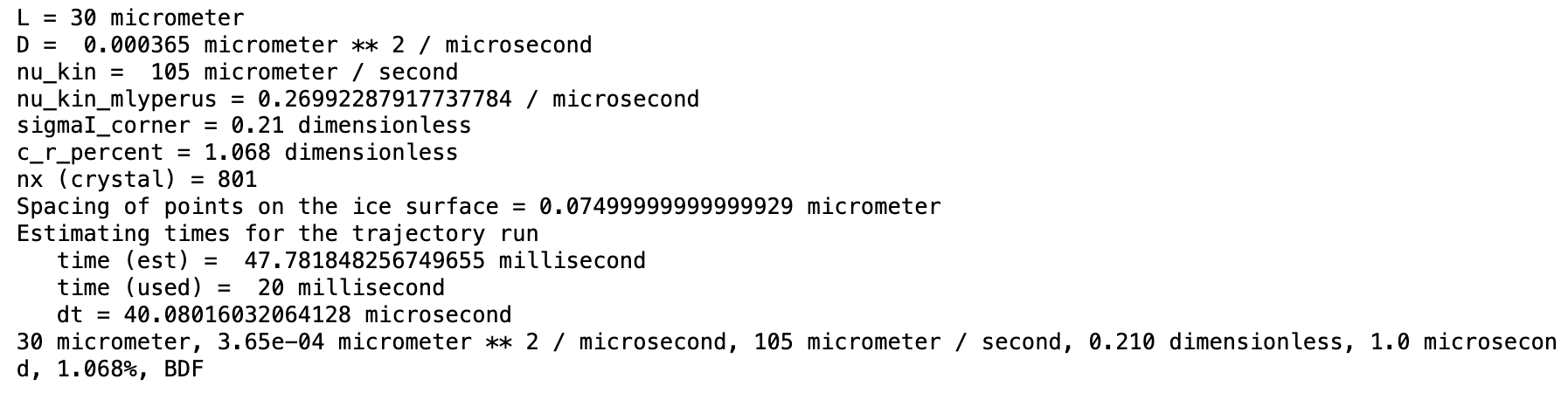
The results are encouraging, in that whereas all the explicit methods exhibited instability by 15 milliseconds, **BDF** remained stable:

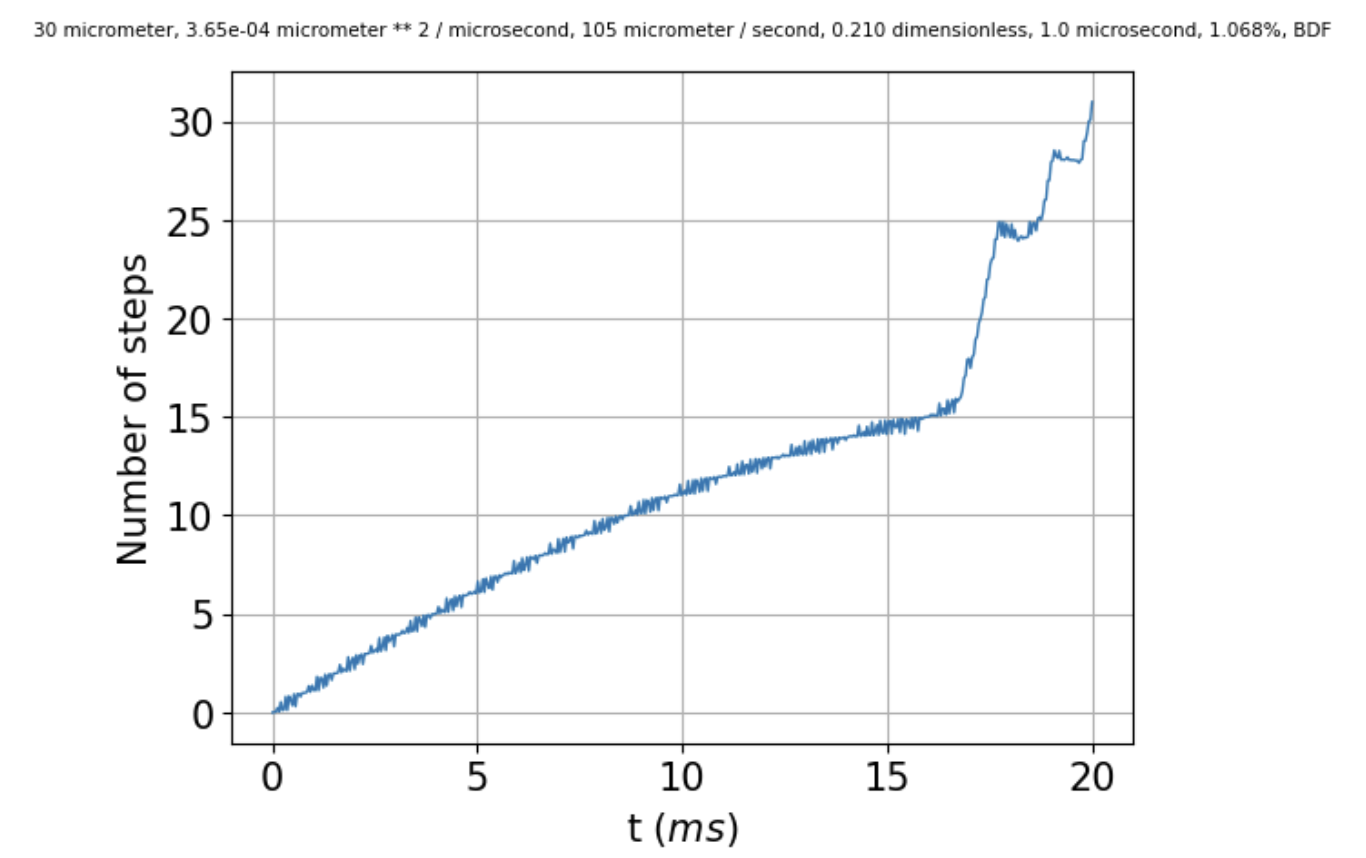
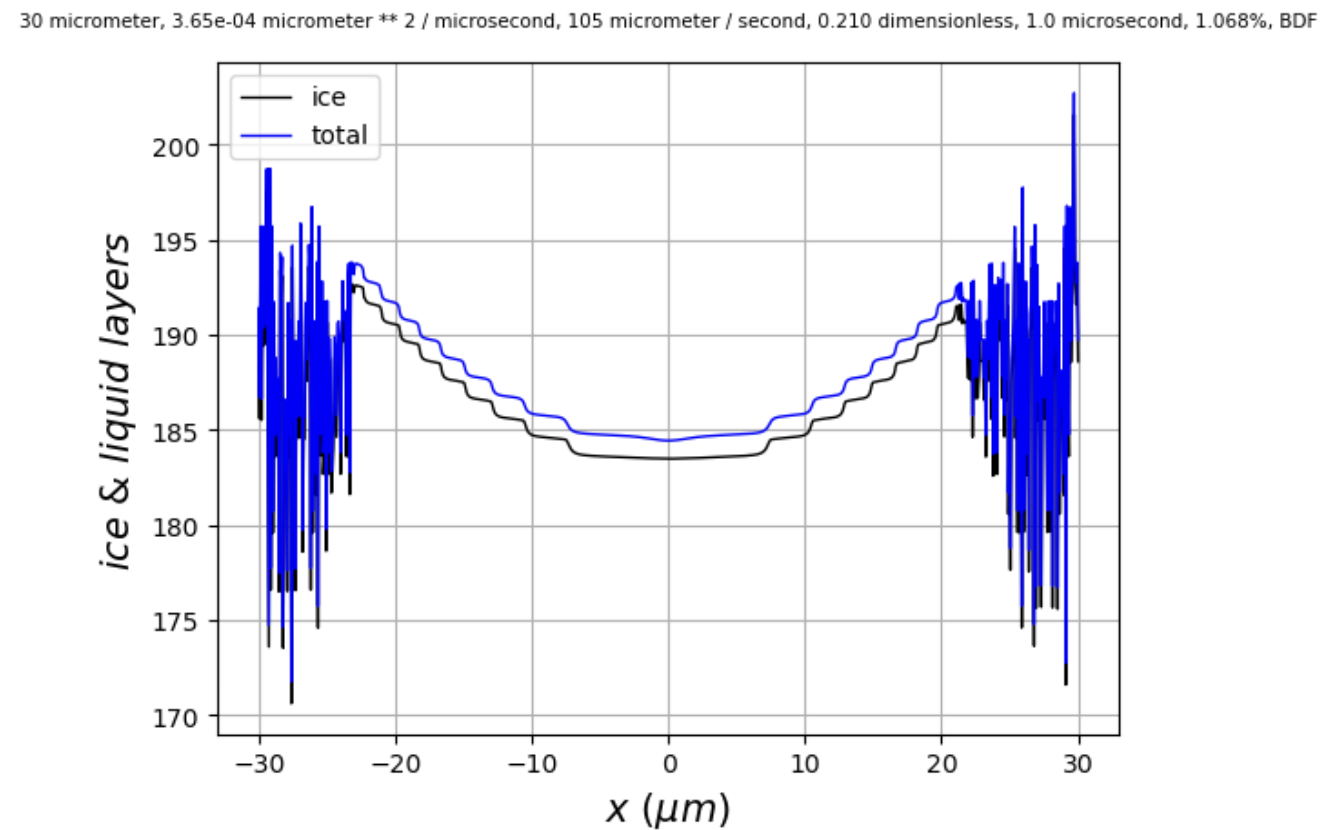


The step width is about (at the corners).

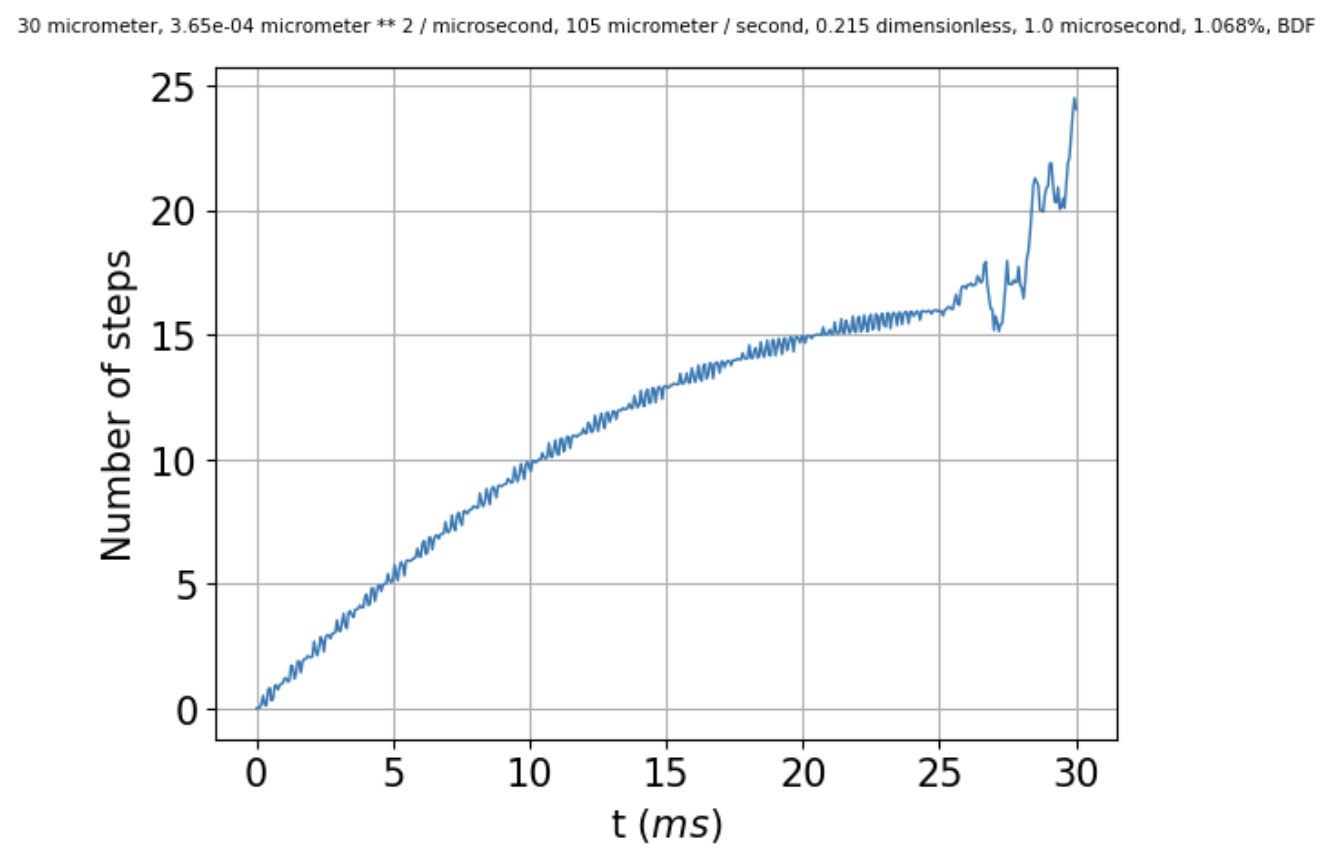
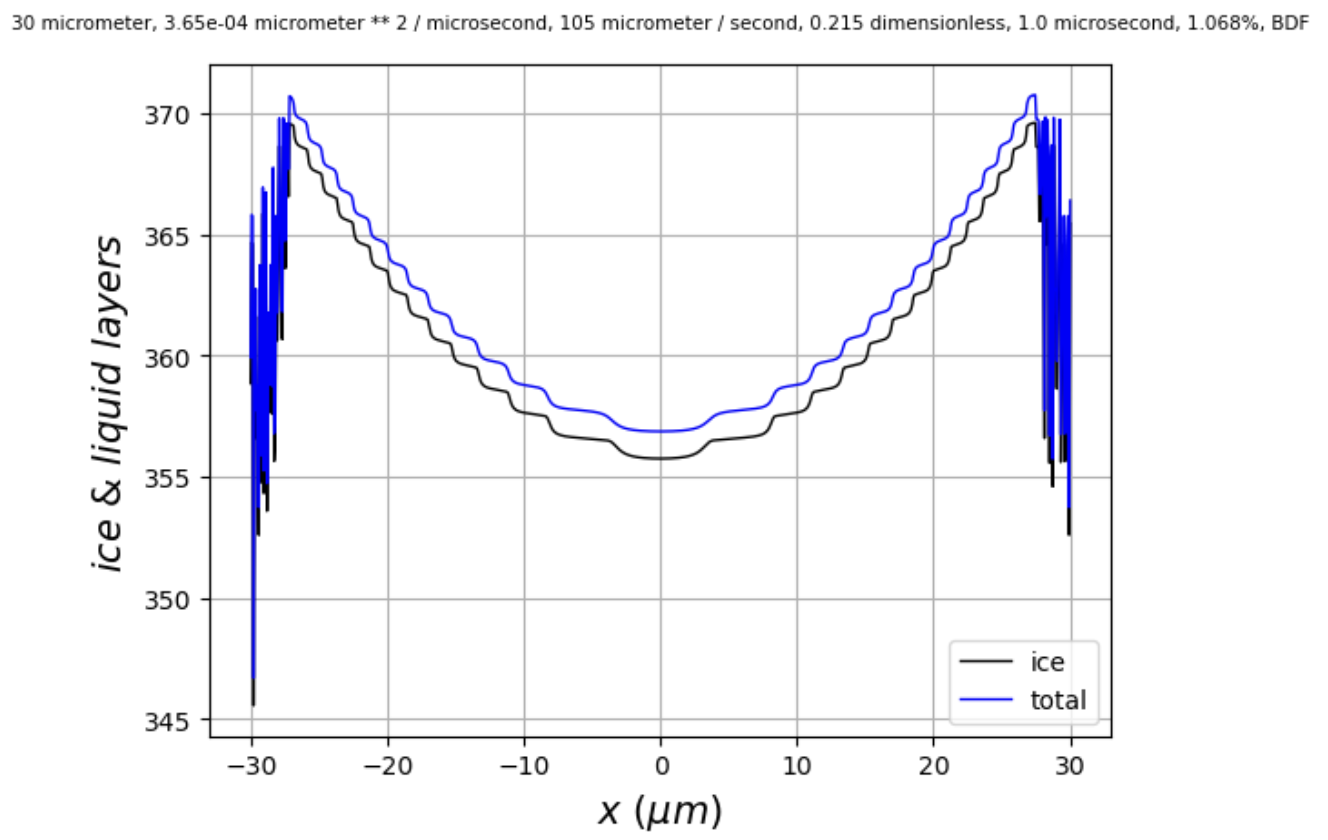
This **BDF** run took about **10 minutes** on my laptop, which is on the order of 100x longer than the explicit methods I’ve tried (**RK23**, **RK35**, **LSODA**, or **DOP853**). **Radau** seems to be even slower, taking about 2x as long as **BDF**.

BDF doesn’t prevent instability altogether, however. When I reduce the corner supersaturation to **0.21** (from 0.22), for example, I get an onset of stability after 17 ms, which is when the number of steps reaches about 16, and the critical step width is (at the corners).



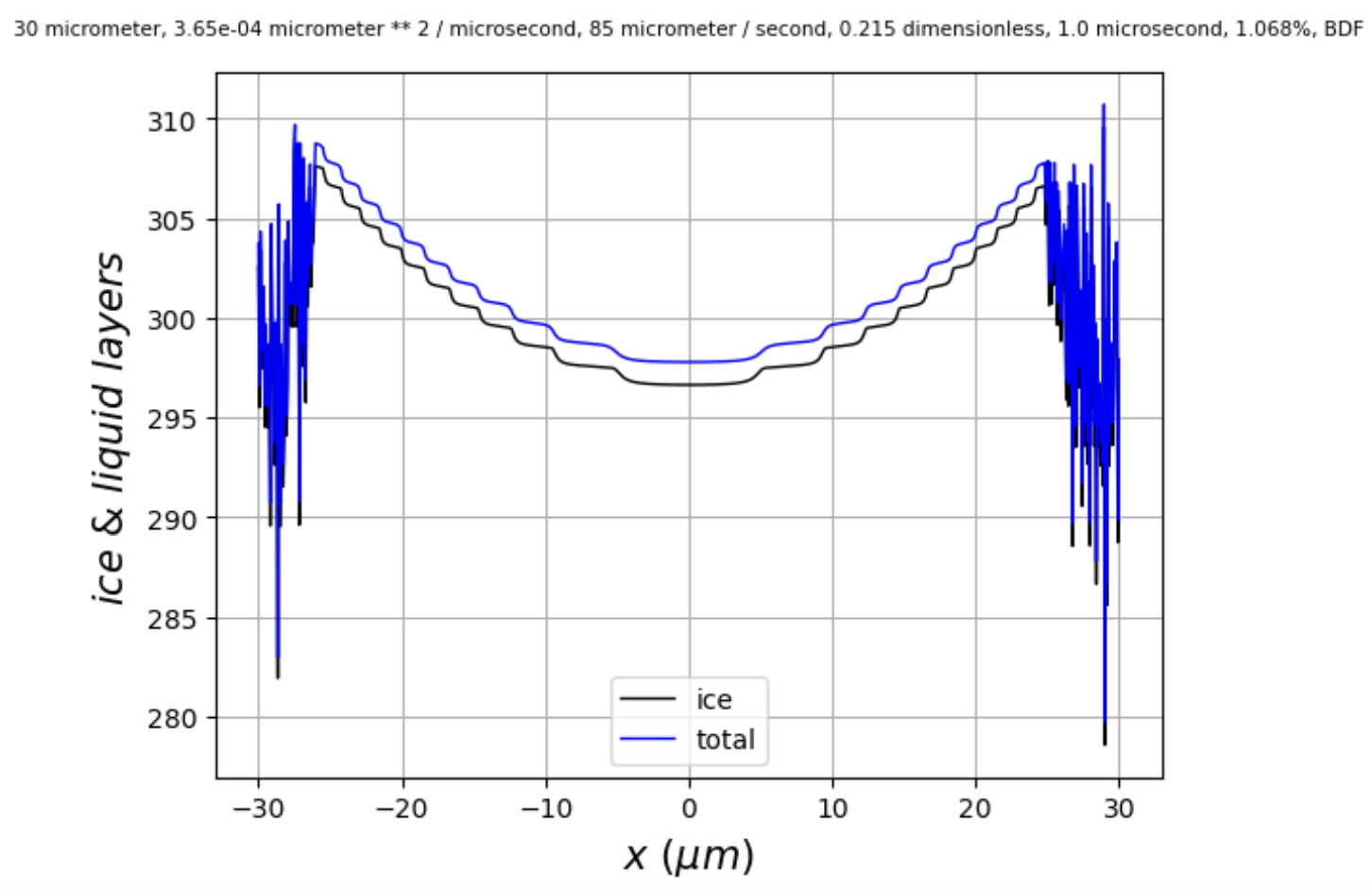
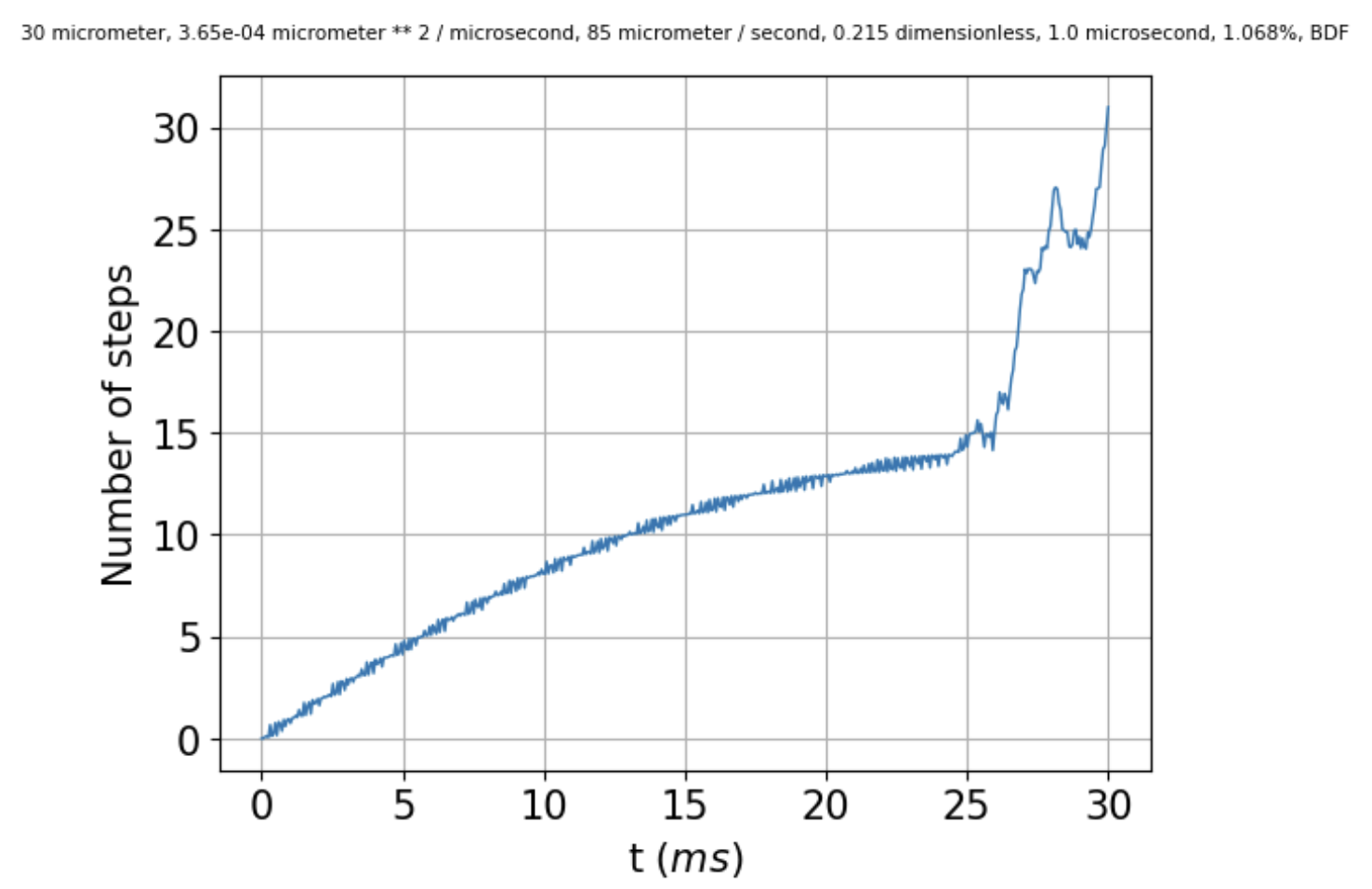
 

If we compare this to when the corner supersaturation is just a little bigger, at **0.215**,

we see that instability sets in at a later **time** (25 ms), but at the **same number of steps** (16), which corresponds to a **critical** **step width** of (at the corners).

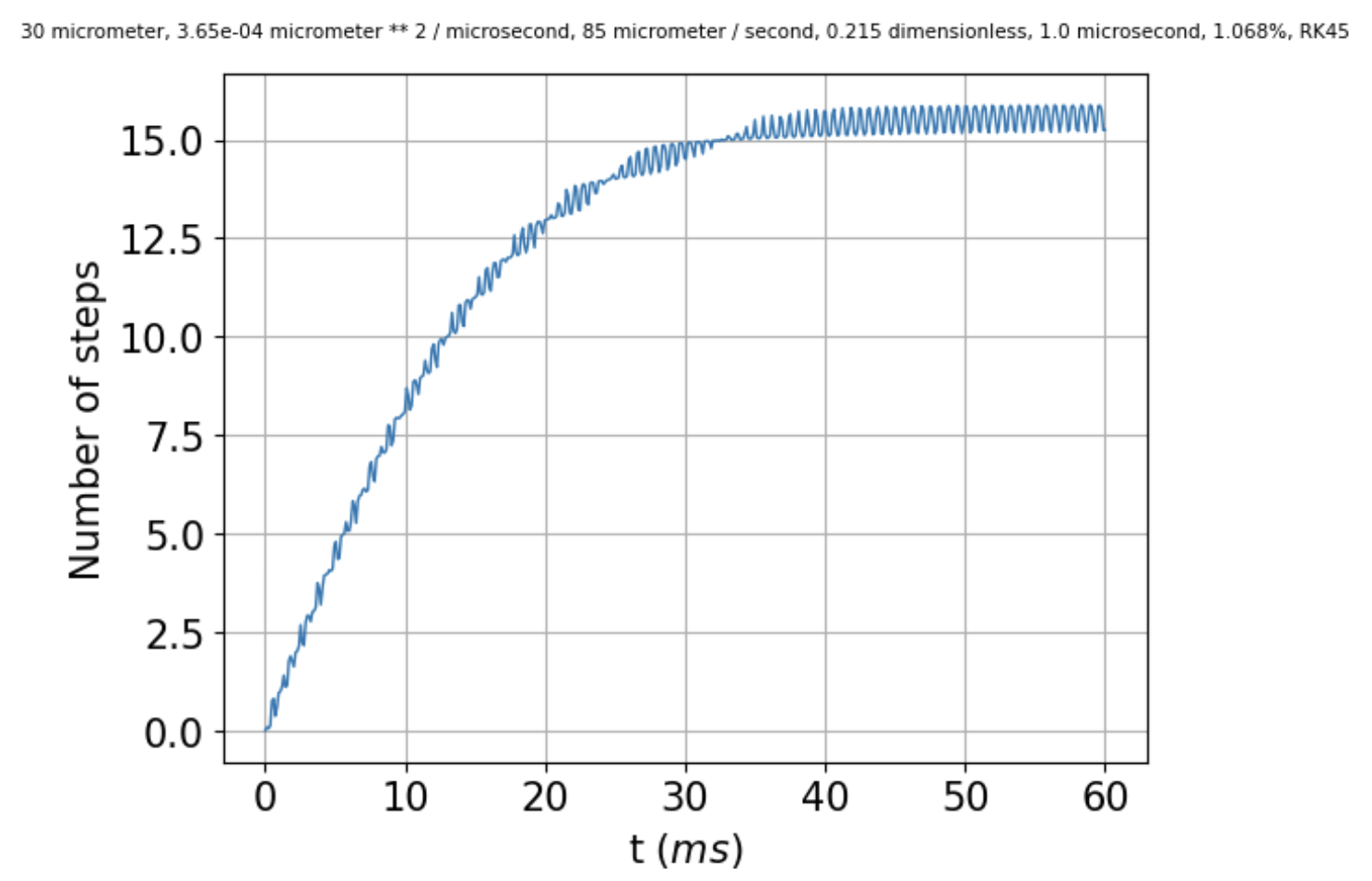
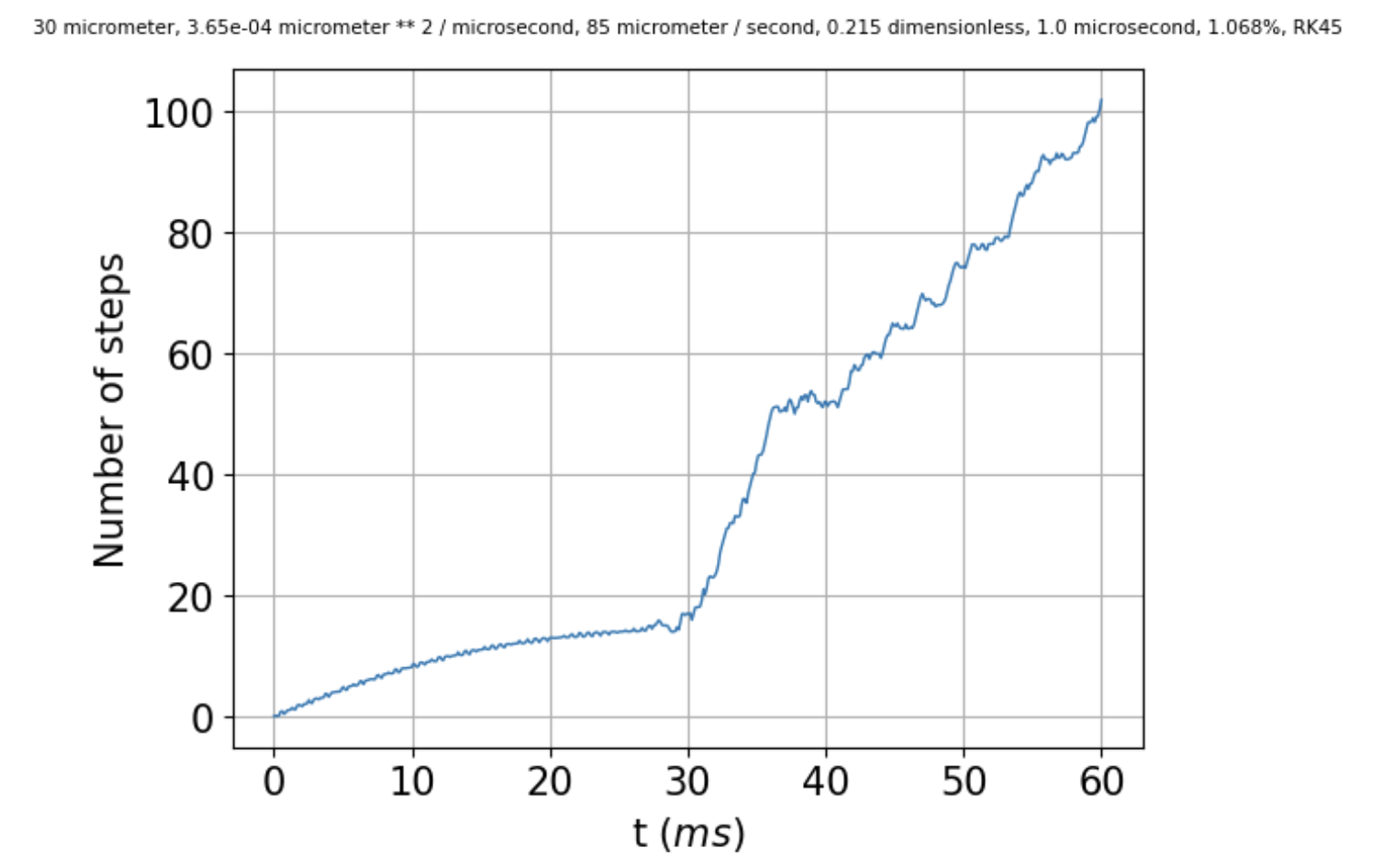
Below is a comparison of runs with vs . Instability seems to set in at a smaller number of steps (closer to 14 steps, vs 16 steps in the run shown above). That means a **bigger critical step width**, .



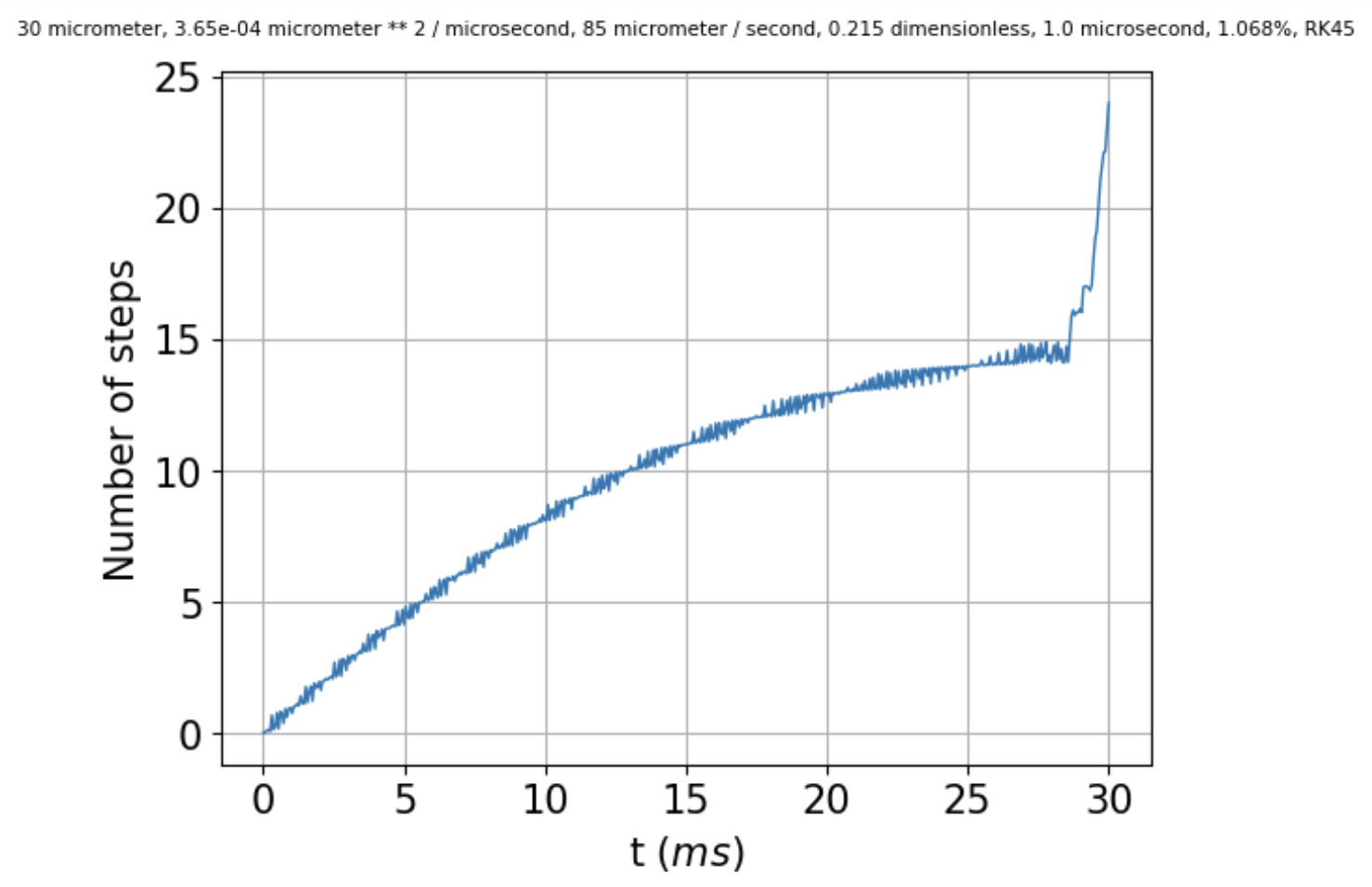
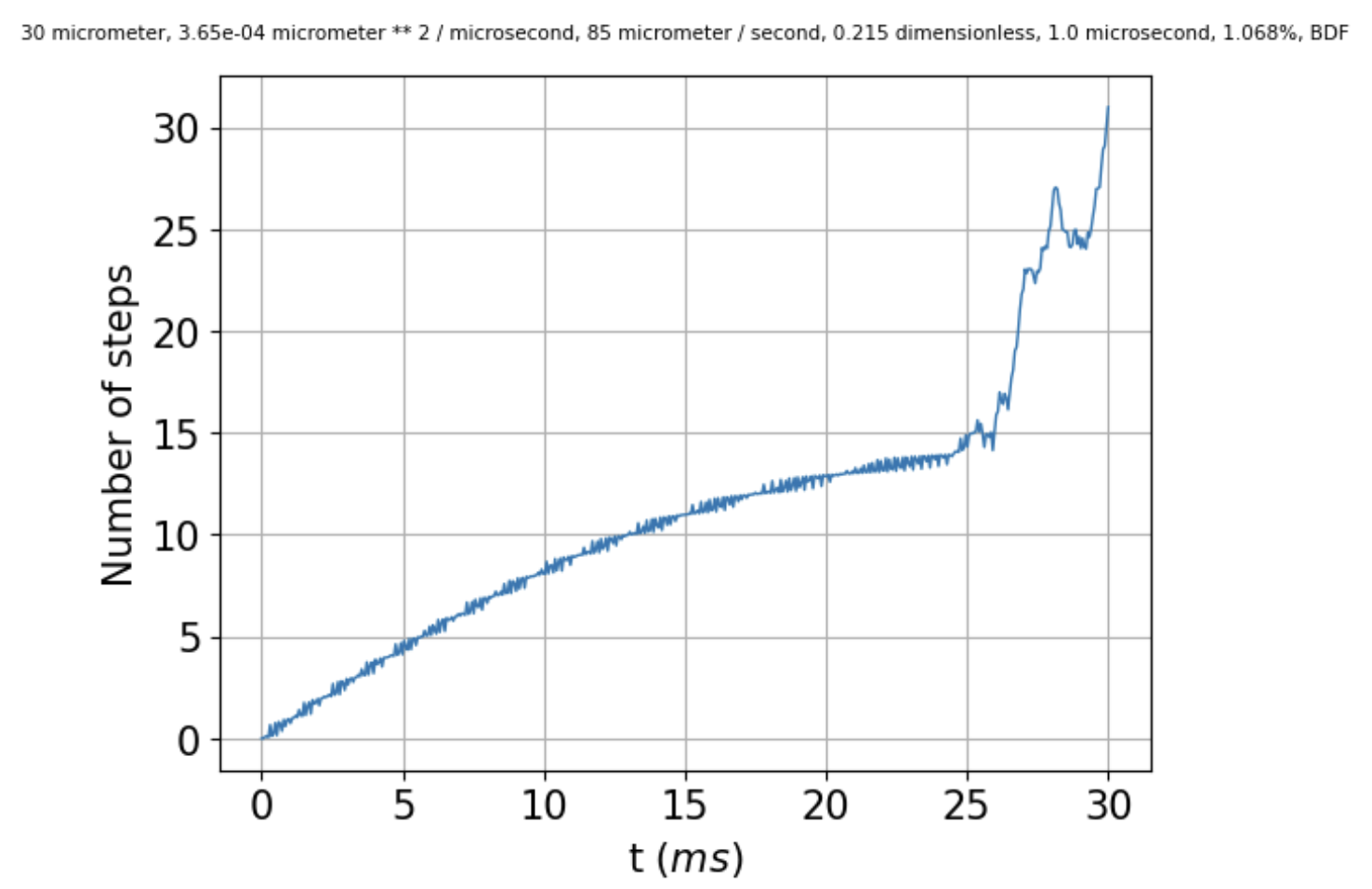
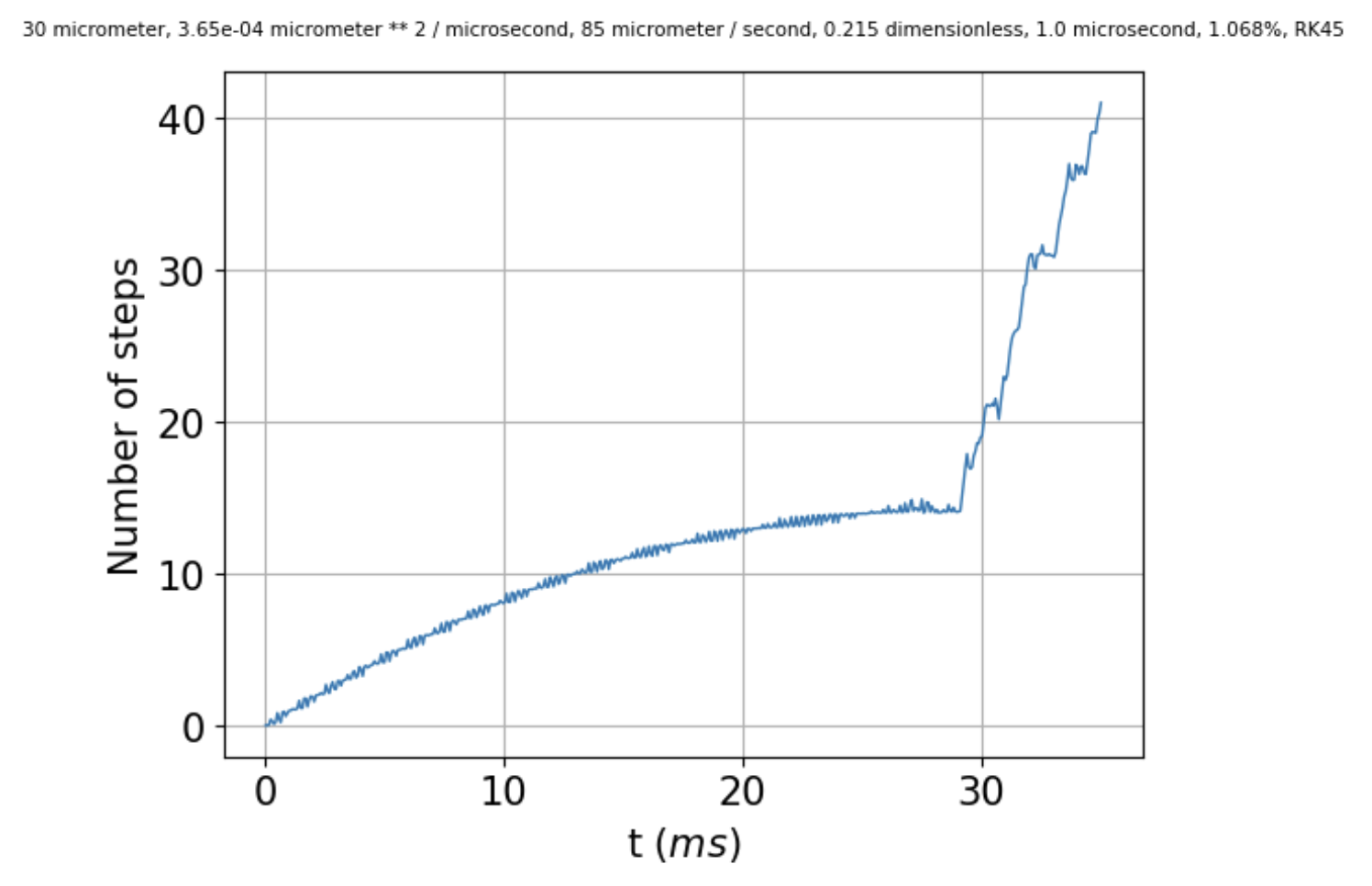
So from this, it would seem to be that the issue is **not** insufficient spatial resolution to capture the shape of the step.

According to the Princeton lecture notes in *cos323\_f11\_lecture19\_pde2.pdf* (found [here](https://www.cs.princeton.edu/courses/archive/fall11/cos323/notes/cos323_f11_lecture19_pde2.pdf)) for the heat equation, stability requires that . This would mean that the integrator must make smaller than a certain value. If the integrator has some minimum value of , then there would be some minimum spatial resolution.

I think I’ve seen this while using RK45. The run on the left below has , while the run on the right has a higher resolution, . That means the critical is , which in turn means (using ) that has to be smaller than .

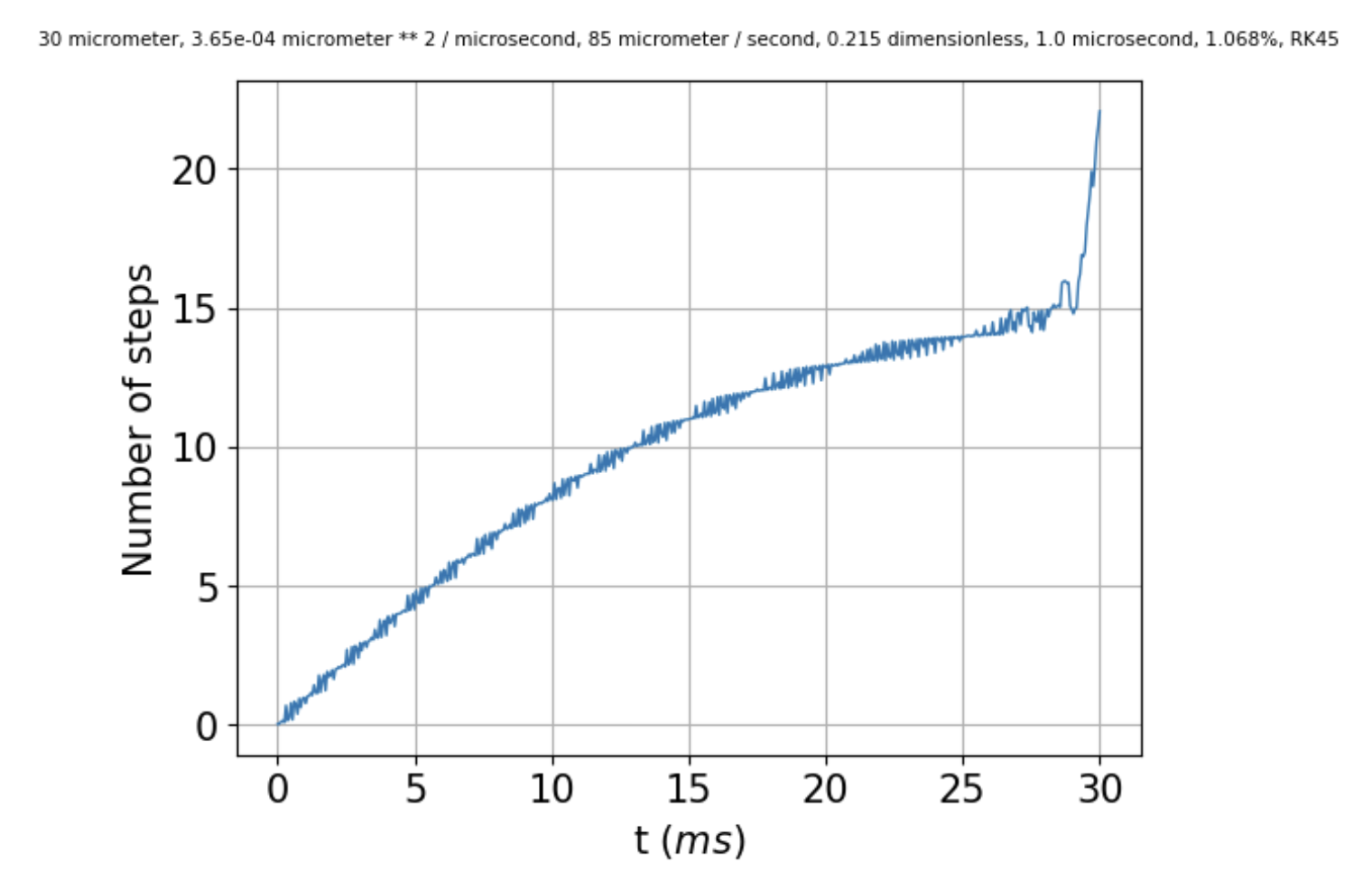
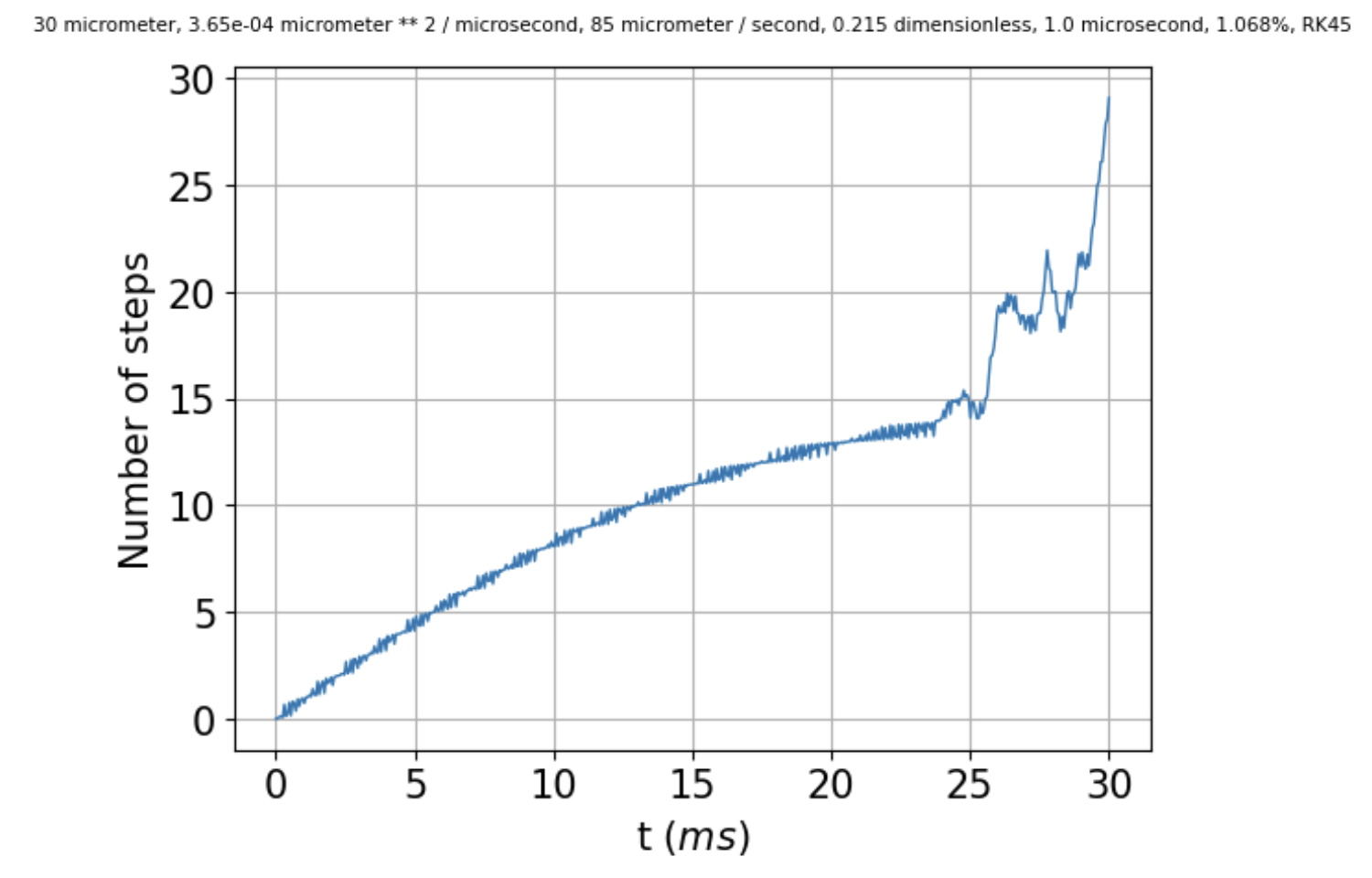
 

I don’t know how to query RK45 directly as to the time step it uses, but we can specify max\_step in the call to scipy.integrate.solve\_ivp. When that’s set to 1, there’s hardly any difference in run time. When it’s set to 0.1 (middle image below), the run time is noticeably longer, which tells me that RK45 is choosing a time step somewhere between 1 and 0.1 microseconds. The results are a little better at 0.1 microseconds, in the sense that the solution remains stable for a few more milliseconds (from 25 to about 28 ms). The kernel died when I tried max\_step = 0.01, but when I set it to 0.02, the solution remained stable even a little longer (to about 29 ms).

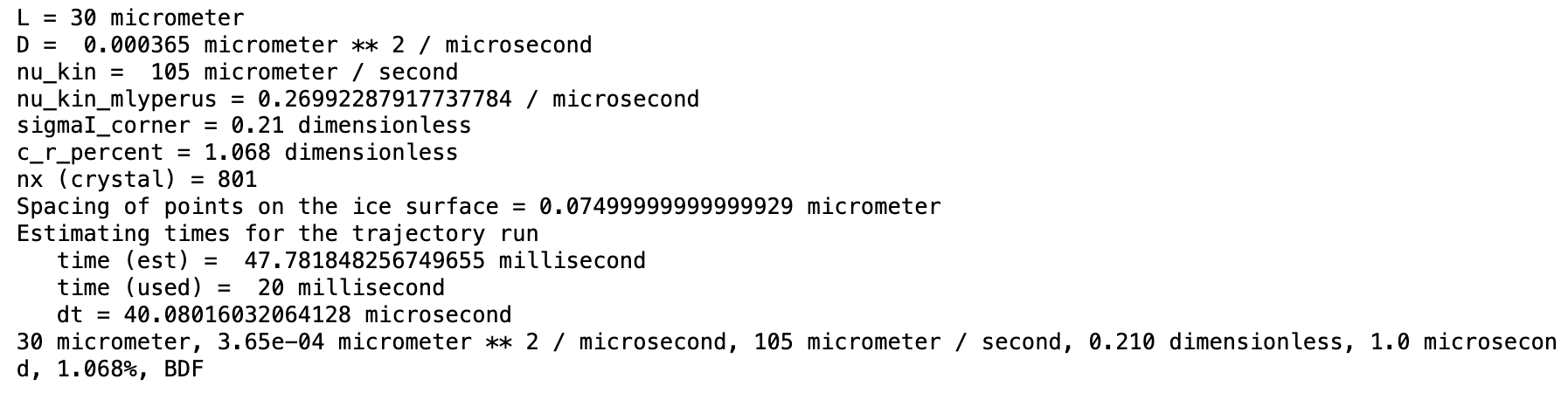
 

For comparison, [Chloe0x0x](https://github.com/chloe0x0/RK45/blob/main/src/rk45.py) uses a fixed time step of 0.01, which easily satisfies .

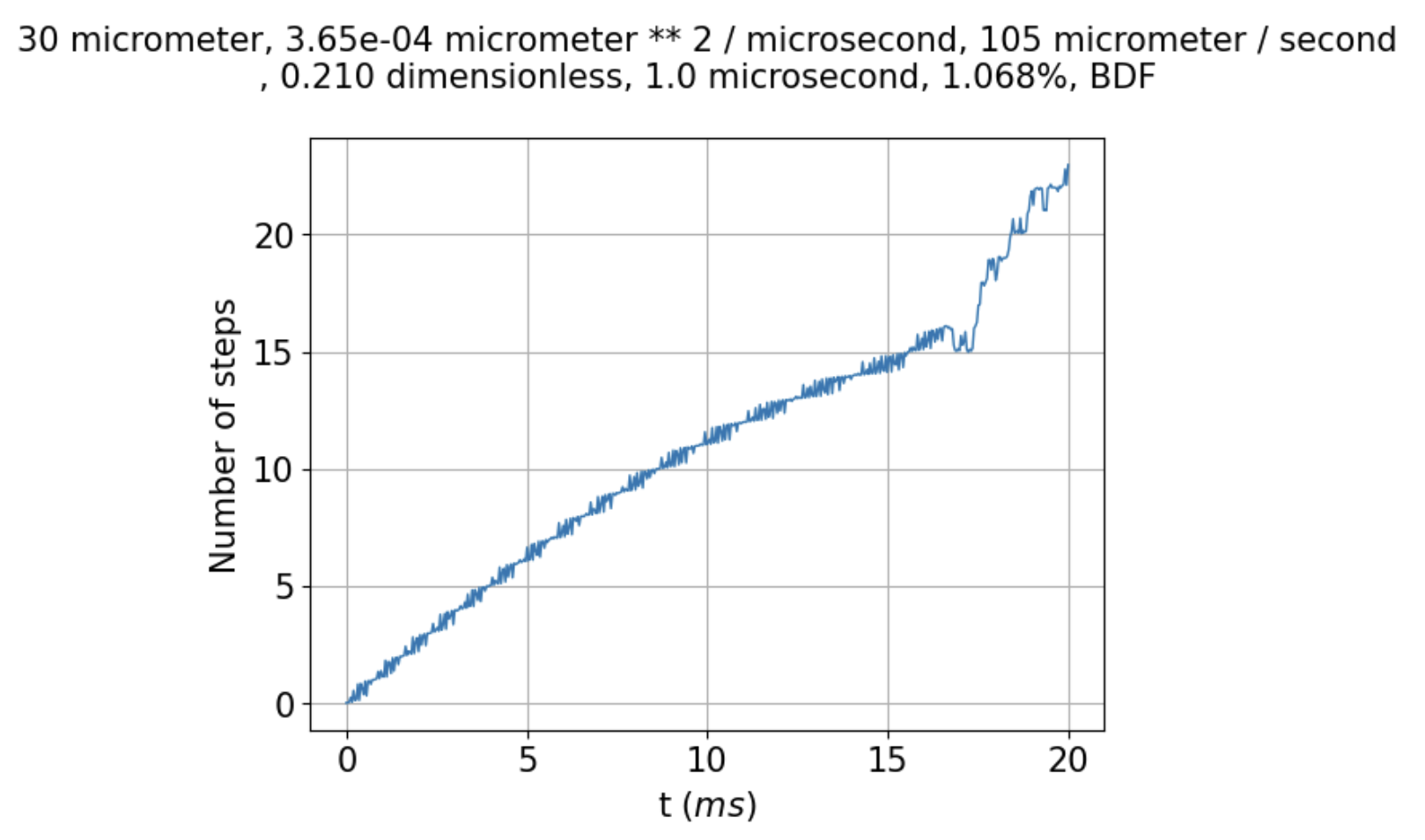
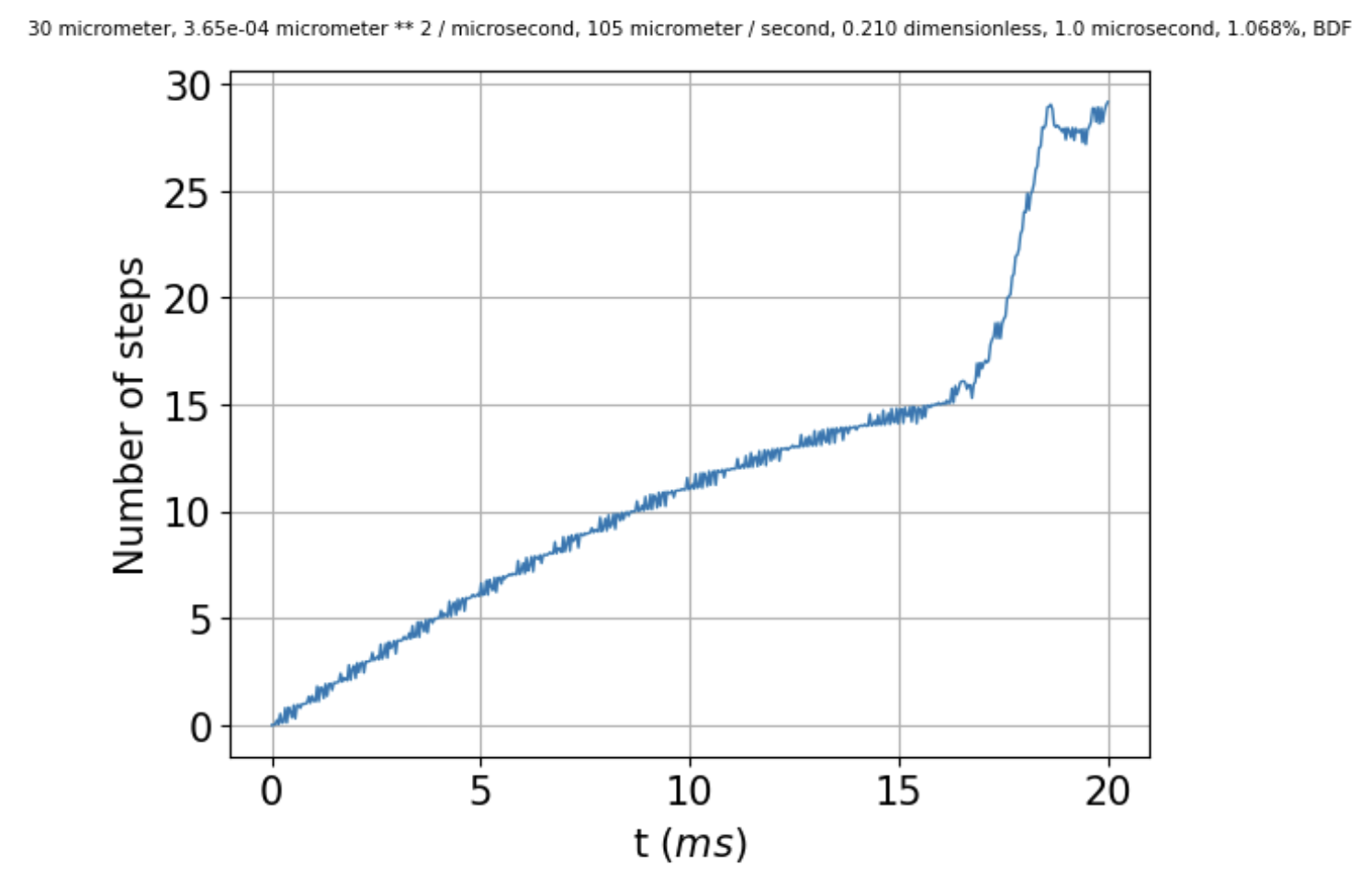
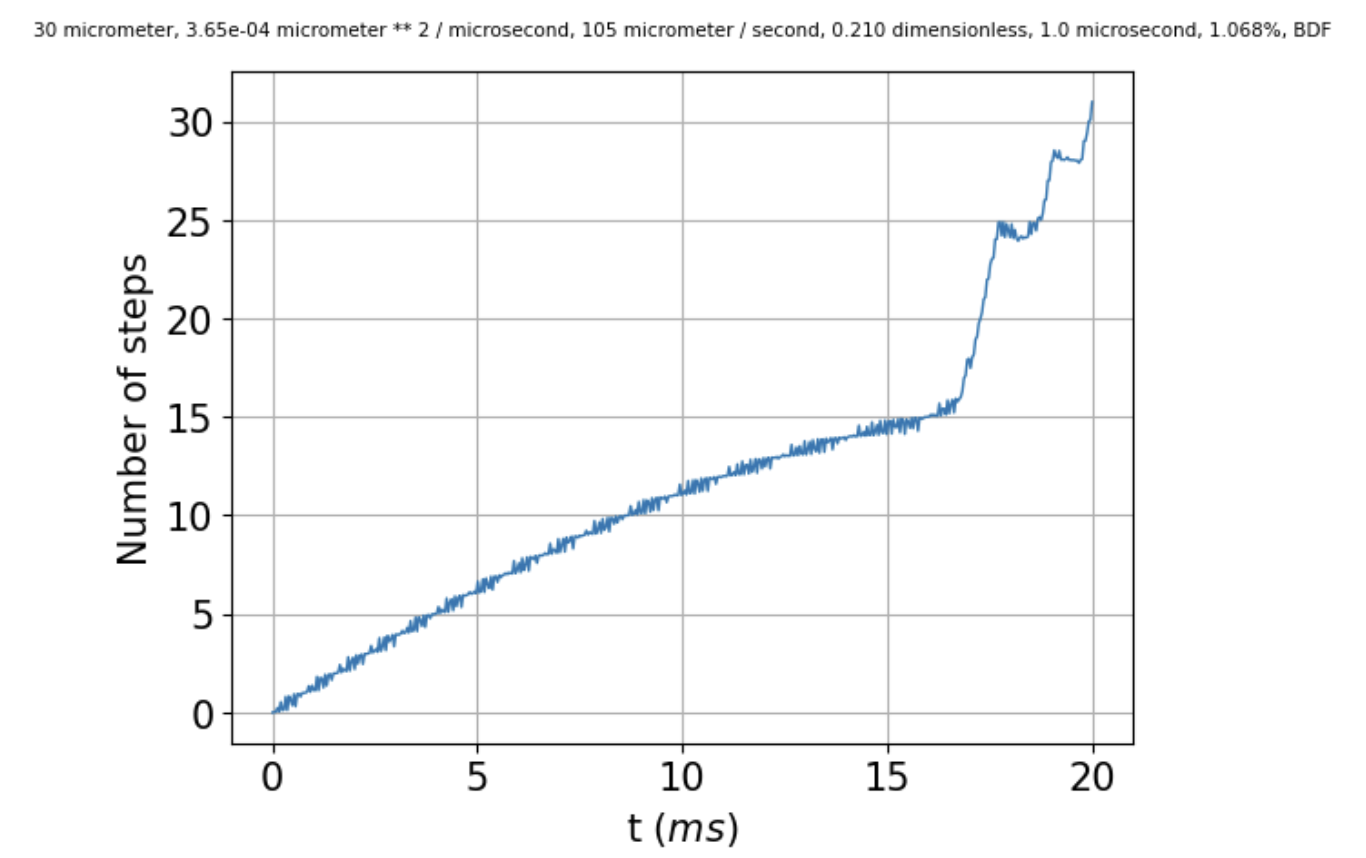
*If we increase rtol, starting from , to , to there’s hardly any difference until we get to the last one. Similar with atol, actually. So no help there.*

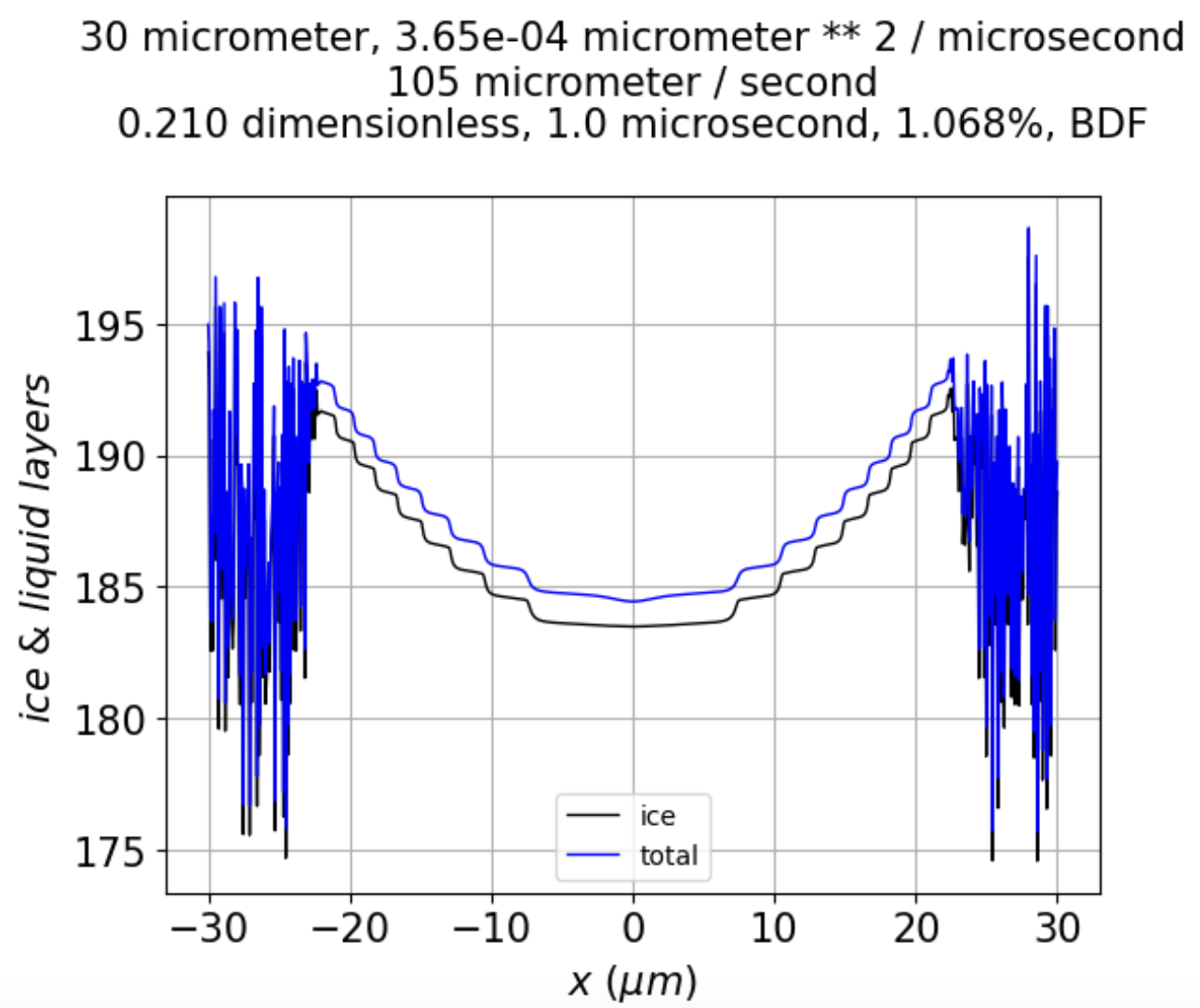
So, if specifying max\_step improves matters (at least a little) for explicit methods, might it help for implicit methods? Duplicating



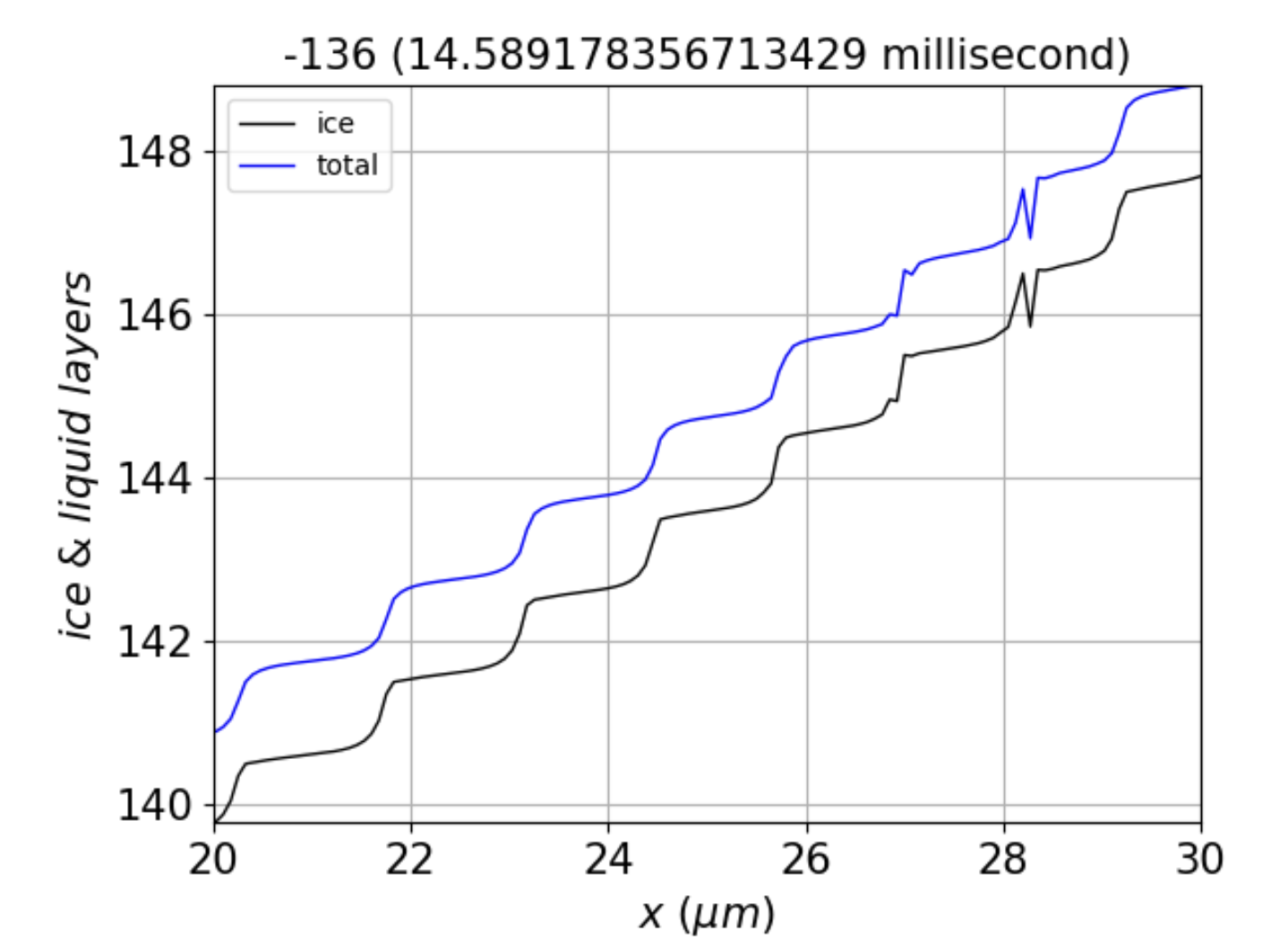
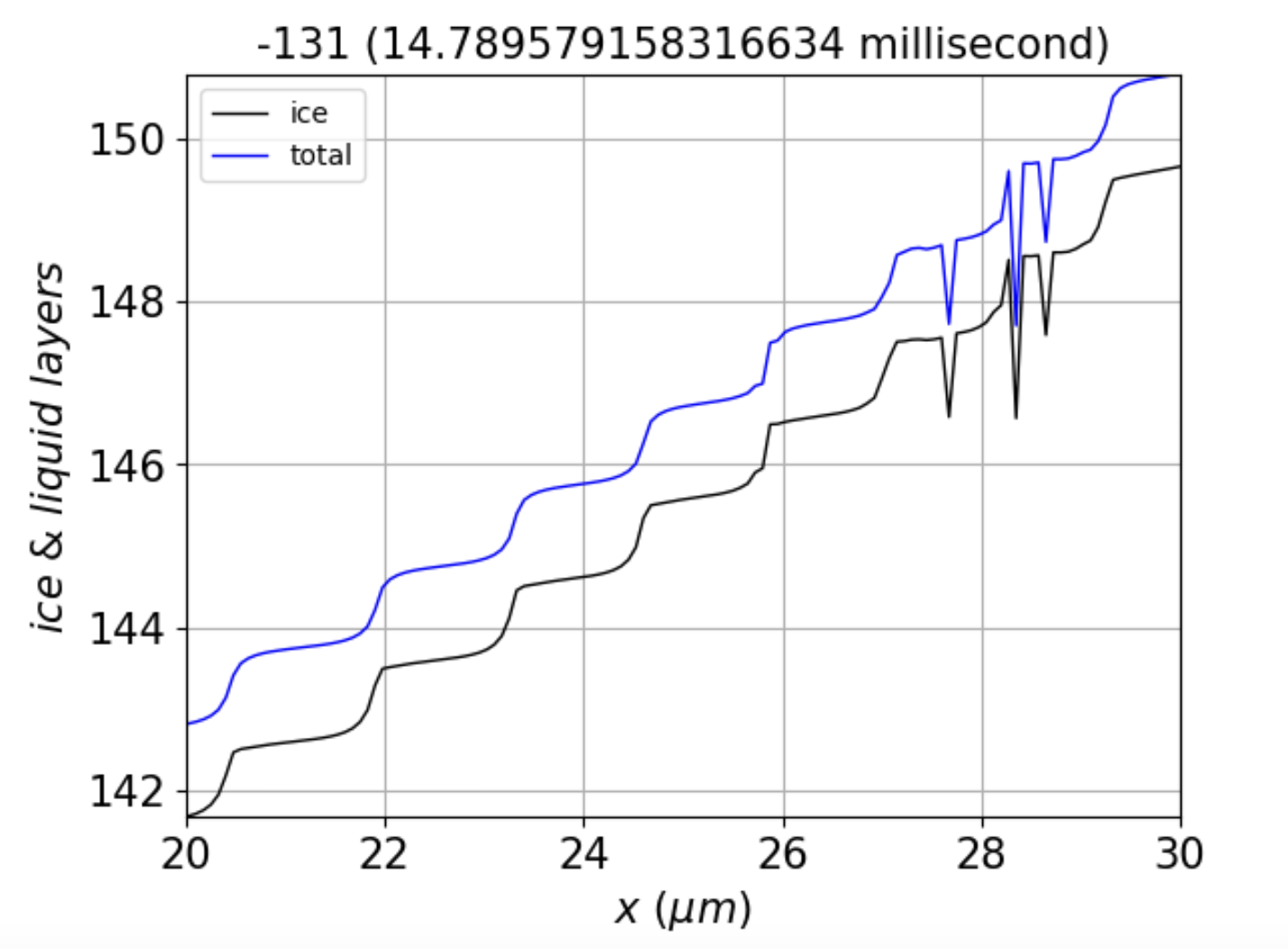
I get the following: no max\_step specified on the left, max\_step=1 microsecond in the middle (no help), and max\_step=0.1 microsecond on the right.



But there seems to be a difference now, in that the number of steps spanned when max\_step=0.1 is a little tamer, going up to less than 25 by the end of the run.



Instability at **risers** seems to persist longer, although they can still disappear over time, as shown by the red arrow below.

The fact that instabilities persist at risers, separated by , seems to point to roughness at a length scale related to (although not necessarily equal to it).

So, what’s the answer? If we take seriously the Princeton formula, , then it would seem that the much smaller value of required would have to be due to **some factor other than diffusion**. According to what I said at the beginning of this document, that would leave **deposition** as the likely culprit. But how does that help?

What I really want is a way to tell whether solutions to the PDE are unstable because:

1. The numerical algorithm is not up to the task, or
2. The instability is intrinsic to the PDE (under certain conditions).

**Thinking about a Fourier option.**

**“Scheme A”**

Say we express the total ice thickness as

(1)

and the QLL as

(2)

Then if you put those expressions into , multiply by and integrate over , it’s possible to derive expressions like

(3)

where (for )

(4)

(5)

while (for )

(6)

(7)

(see the Python code *k-space solution.ipynb*.)

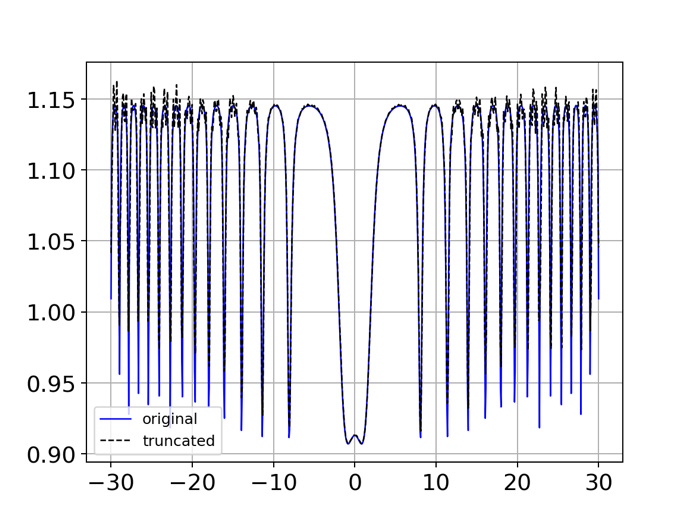
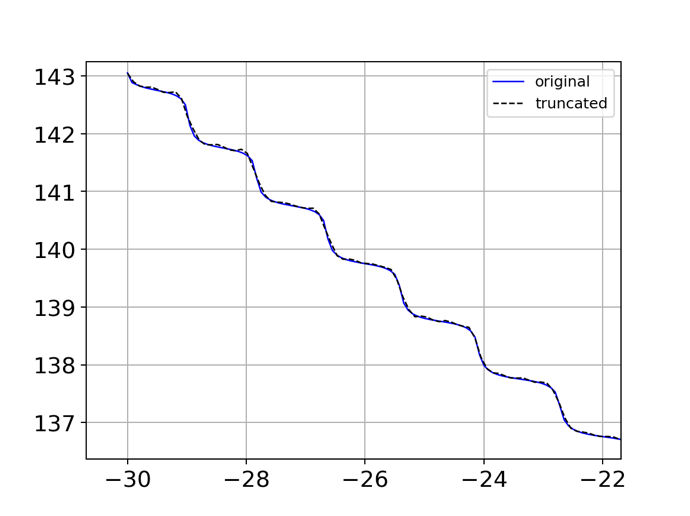
For b-coefficient derivatives, we start with , and again multiply by and integrate over , giving

(8)

(9)

So basically, **Scheme A** runs the simulation in k-space (the Fourier space), except that to describe freezing and melting, we **inverse FT** the a-values to get , use that to compute , then **FT** the result to get back to .

**How many -values do we need?** To get a sense of that, we can take cosine transforms of and obtained from the x-space simulations. Here’s the result for , truncated to have (rather than ) in the expansion.



**Scheme “B”**

An alternative to the foregoing is to get in a truncated (low-pass) Fourier space. How?

1. Transform to its Fourier space:

bj\_list = rfft()

1. Get rid of the high-frequency elements of (by *shortening* or *zeroing*) and create a truncated list (bj\_list\_tr).
2. Since , compute , as in

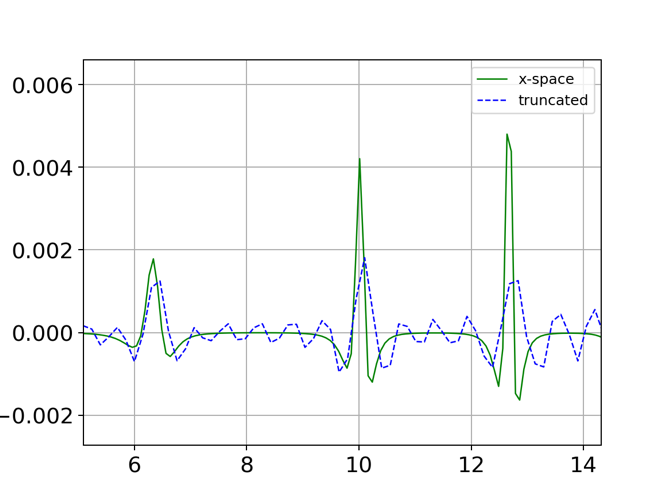
cj\_list\_tr = j\_list\_tr\*\*2 \* bj\_list\_tr

1. Transform back to the original space to get :

Dterm irfft(cj\_list\_tr)\* rescale

where rescale = nx\_Ntot\_tr\_regen/nx\_Ntot\_regen if the Fourier space was truncated by shortening (no need to do this if the truncation is done by zeroing).

The above formalism is correct. Some comparisons are shown for Dterm below, using jmax=100 (instead of 200). The FT/truncated result is much less “spikey” at the risers than the original x-space. That means QLL will diffuse into the riser areas more slowly, having the effect of a smaller diffusion coefficient – which promotes stability. But the FT/truncated result is also noisier – it should be close to zero on steps, but instead it’s bumpy there.



We can modify the integration code (f1d\_solve\_ivp) accordingly:

# Ntot diffusion

# dy = np.empty(np.shape(NQLL0))

# for i in range(1,len(NQLL0)-1):

# dy[i] = DoverdeltaX2\*(NQLL0[i-1]-2\*NQLL0[i]+NQLL0[i+1])

# dy[0] = DoverdeltaX2\*(NQLL0[-1] -2\*NQLL0[0] +NQLL0[1]) # Periodic BC

# dy[-1] = DoverdeltaX2\*(NQLL0[-2] -2\*NQLL0[-1]+NQLL0[0])

# Diffusion term based on a zeroed-out FT

Dcoefficient1 = 4\*DoverdeltaX2/l\*\*2\*np.pi\*\*2; #print('Dcoefficient1', Dcoefficient1)

bj\_list = rfft(NQLL0)

jmax = 130

j\_list = np.array([j for j in range(len(bj\_list))])

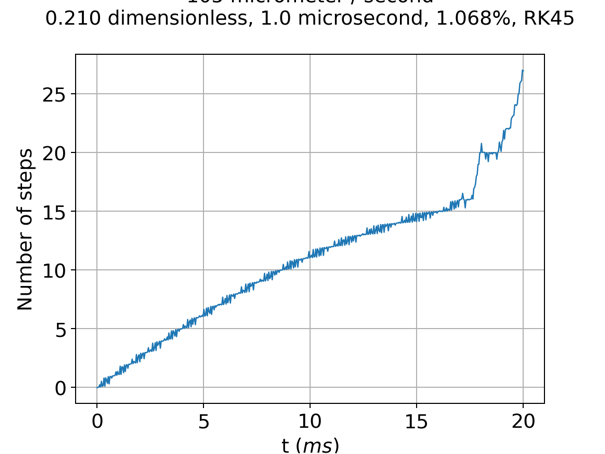
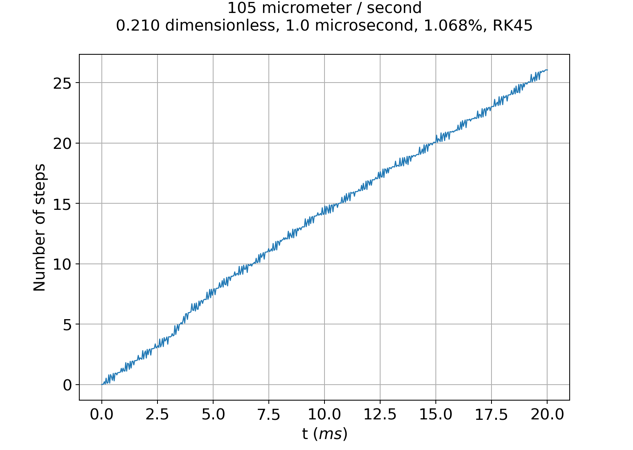
j2\_list = np.array(j\_list)\*\*2

j2\_list[(j\_list>jmax)] = 0

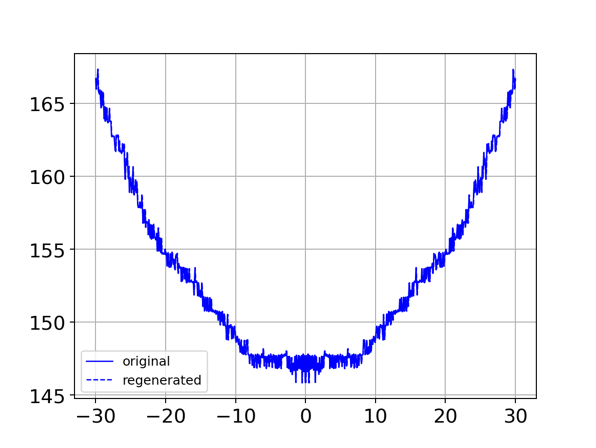
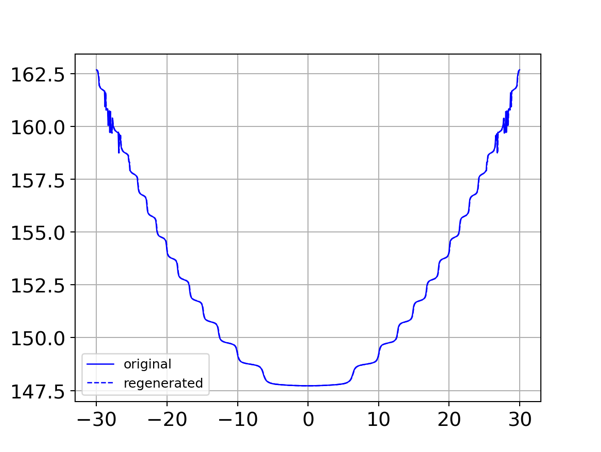
cj\_zeroed\_list = bj\_list\*j2\_list

dy = -Dcoefficient1 \* irfft(cj\_zeroed\_list)

But the result is terrible. On the left is the (original) x-space diffusion code, which is OK up to about 17 ms. On the right is the zeroed-out FT code, which is seen to be unstable long before that time.

The profile is very noisy. Here are the two systems at a time of 16 ms:



So the answer is, **Scheme B won’t work**.