Instruction notes

Exporting results from the semi-analytical Matalb model into Gkyl

Andréas Sundström

2018-08-15

Abstract

This document is comprised of a series of notes related to the export of results from the semi-analytical shock model into Gkyl.

Contents

1 Introduction (in progress)

The shocks we use in this package are from the semi-analytical model of electrostatic shocks described in my thesis [1]. The Matlab shock package, `Shock_pgk_new`, is used here to generate the electrostatic potentials and electric fields, which are exported into Gkyl.

2 Normalization for multi-species plasmas

The use of numerical tools and computer programs requires normalized (dimensionless) quantities[1]. In this section we start from the (non-normalized) ion distribution function of the shocks,

$$\hat{f}_{\mathrm{j}}(\hat{v},\hat{\phi}) = \frac{\hat{n}_{\mathrm{j},0}}{\sqrt{2\pi\hat{T}_{\mathrm{j}}/\hat{m}_{\mathrm{j}}}}\exp\left[-\frac{\hat{m}_{\mathrm{j}}}{2\hat{T}_{\mathrm{j}}}\left(\sqrt{\hat{v}^2+\frac{2eZ_{\mathrm{j}}\hat{\phi}}{\hat{m}_{\mathrm{j}}}}-\hat{V}\right)^2\right], \tag{1}$$

and all the physical quantities needed to describe it. We will be basing out normalization on the idea that the central quantity is the speed of sound, $\hat{c}_{\mathrm{s}}$, and then normalize the other quantities above in the most natural way.

We will use the sound speed in an arbitrary ion composition plasma[2]

$$\hat{c}_{\mathrm{s}}^2 = \frac{\hat{T}_{\mathrm{e}}}{\hat{n}_0}\sum_{\mathrm{j}}\frac{Z_{\mathrm{j}}^2\hat{n}_{\mathrm{j},0}}{\hat{m}_{\mathrm{j}}} = \hat{T}_{\mathrm{e}}\frac{\sum_{\mathrm{j}}\frac{Z_{\mathrm{j}}}{\hat{m}_{\mathrm{j}}}\hat{n}_{\mathrm{j},0}Z_{\mathrm{j}}}{\sum_{\mathrm{j}'}\hat{n}_{\mathrm{j}',0}Z_{\mathrm{j}'}}. \tag{2}$$

From the last step we see that $\hat{c}_{\mathrm{s}}^2/\hat{T}_{\mathrm{e}}$ represents an average charge-to-mass ratio, $Z/\hat{m}$, weighted with $\hat{n}_{\mathrm{j},0}Z_{\mathrm{j}}$. We will therefore use this in our normalization of the charge-to-mass ratio.

Using the normalized velocity and charge-to-mass ratio

$$v \coloneqq \frac{\hat{v}}{\hat{c}_{\mathrm{s}}} \quad \text{and} \quad \zeta_{\mathrm{j}} \coloneqq \frac{Z_{\mathrm{j}}/m_{\mathrm{j}}}{\hat{c}_{\mathrm{s}}^2/\hat{T}_{\mathrm{e}}}, \tag{3}$$

it becomes natural to normalize the electrostatic potential and temperatures to

$$\psi \coloneqq \frac{e\hat{\phi}}{\hat{T}_{\mathrm{e}}} \quad \text{and} \quad \tau_{\mathrm{j}} \coloneqq \frac{Z_{\mathrm{j}}\hat{T}_{\mathrm{e}}}{\hat{T}_{\mathrm{j}}}. \tag{4}$$

In this normalization, the ion distribution function becomes

$$f_{\mathrm{j}}(v,\psi) = n_{\mathrm{j},0}\sqrt{\frac{\tau_{\mathrm{j}}}{2\pi\zeta_{\mathrm{j}}}}\exp\left[-\frac{\tau_{\mathrm{j}}}{2\zeta_{\mathrm{j}}}\left(\sqrt{v^2+2\zeta_{\mathrm{j}}\psi}-\mathscr{M}\right)^2\right], \tag{5}$$

[1] In these notes we use "hat" to denote a dimensional variables, e.g. $\hat{T}_{\mathrm{e}} = 1\,\mathrm{MeV}$, and no "hat" signifies that the variable is either dimensionless from the start or normalized.

[2] See ch. 2.1.4 in my thesis [1] for more details on how this sound speed is derived. That this normalization differs slightly from the one in the thesis, due to the fact that we are using the *arbitrary* ion composition sound speed.

where the normalized density, $n_{\mathrm{j},0} \coloneqq \hat{n}_{\mathrm{j},0}/\hat{n}_0$, and distribution, $f_{\mathrm{j}} \coloneqq \hat{f}_{\mathrm{j}} \times \hat{c}_{\mathrm{s}}/\hat{n}_0$, with

$$\hat{n}_0 \coloneqq \sum_{\mathrm{j}} \hat{n}_{\mathrm{j},0} Z_{\mathrm{j}}. \tag{6}$$

2.1 Other normalized quantities

From Poisson's equation we get the link between $\hat{\phi}$ and $\hat{x}$, and thereby also the natural position normalization, $x = \hat{x}/\bar{x}$:

$$\frac{\hat{\epsilon}_0}{\hat{e}} \frac{\mathrm{d}^2\hat{\phi}}{\mathrm{d}\hat{x}^2} = \hat{n}_0 n_{\mathrm{e}} - \sum_{\mathrm{j}} Z_{\mathrm{j}} \hat{n}_0 n_{\mathrm{j}} = \hat{n}_0 \rho \quad \Longrightarrow \quad \frac{\hat{\epsilon}_0 \hat{T}_{\mathrm{e}}}{\hat{e}^2 \hat{n}_0} \frac{1}{\bar{x}^2} \frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} = \rho, \tag{7}$$

which we from here see is

$$x = \frac{\hat{x}}{\hat{\lambda}_{\mathrm{D}}}, \quad \text{where} \quad \hat{\lambda}_{\mathrm{D}} = \sqrt{\frac{\hat{\epsilon}_0 \hat{T}_{\mathrm{e}}}{\hat{e}^2 \hat{n}_0}}. \tag{8}$$

Do note that this normalization of x defines one length scale, $\hat{\lambda}_{\mathrm{D}}$, while the density normalization also implicitly defines a length scale, $1/\hat{n}_0$. However, as long as these two quantities are not added, this should not result in any major problems.

With the normalization of x, we can easily conclude that the natural normalization of the electric field must be

$$E = -\frac{\mathrm{d}\psi}{\mathrm{d}x} = -\frac{\hat{\lambda}_{\mathrm{D}} \hat{e}}{\hat{T}_{\mathrm{e}}} \frac{\mathrm{d}\hat{\phi}}{\mathrm{d}\hat{x}} = \frac{\hat{\lambda}_{\mathrm{D}} \hat{e}\, \hat{E}}{\hat{T}_{\mathrm{e}}}. \tag{9}$$

We can also use the position normalization to find the natural time normalization

$$t = \frac{\hat{t}}{\hat{\lambda}_{\mathrm{D}}/\hat{c}_{\mathrm{s}}}. \tag{10}$$

This also results in an elegantly normalized (non-static) Vlasov equation

$$\frac{\partial f}{\partial t} + v \frac{\partial f}{\partial x} - \zeta_{\mathrm{j}} \frac{\mathrm{d}\psi}{\mathrm{d}x} \frac{\partial f}{\partial v} = 0. \tag{11}$$

These are most of the quantities needed in any shock calculation.

2.1.1 Electrostatic force and charge-to-mass ratios

With the time normalized, I would just like to point out one special feature of this normalization scheme. That is the use of the charge-to-mass ratio, ζ_{j}, and not the individual charges (except in ρ) or masses.

This is clearly exemplified in Newton's second law and electric force,

$$\hat{m} \frac{\mathrm{d}^2\hat{x}}{\mathrm{d}\hat{t}^2} = \hat{e} Z\, \hat{E}, \tag{12}$$

which normalizes to

$$\frac{\hat{\lambda}_\mathrm{D}}{\hat{\lambda}_\mathrm{D}^2/\hat{c}_\mathrm{s}^2}\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{\hat{e}Z}{\hat{m}}\frac{\hat{T}_\mathrm{e}}{\hat{\lambda}_\mathrm{D}\hat{e}}E \quad\Longrightarrow\quad \frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{Z/\hat{m}}{\hat{c}_\mathrm{s}^2/\hat{T}_\mathrm{e}}E \equiv \zeta E. \tag{13}$$

In the same way only ζ will show up in the Vlasov equation, wich is what Gkyl solves. The fact that only ζ shows up here results in some minor complications in Gkyl. This is because Gkyl wants us to specify both the mass and charge of each species, which will be used to evolve the distribution function.

This problem is however easily mended by specifying either $\{\texttt{mass} = 1, \texttt{charge} = \zeta_\mathrm{i}\}$ for ions, and $\{\texttt{mass} = 1/|\zeta_\mathrm{e}|, \texttt{charge} = -1\}$ for electrons. In theory, the first alternative should work just as well for electrons as well. However, since the normalization of ζ is with respect to the ions, $\zeta_\mathrm{e} \sim -2000$ for electrons, and Gkyl doesn't seem to like the charge to be that large. In the same way it's also safes to set the mass of the electron to be positive, and the charge to be negative.

2.2 The electron distribution function

For now the electron distribution function is set as a Maxwell-Boltzmann distribution, and is normalized using the same scheme as the ions:

$$f_\mathrm{e} = n_{\mathrm{e},1}\sqrt{\frac{1}{2\pi|\zeta_\mathrm{e}|}}\exp\left[-\frac{(v+\mathscr{M})^2}{2|\zeta_\mathrm{e}|} + \psi\right]. \tag{14}$$

To keep quasi-neutrality, we must set the electron density such that it balances the far upstream ion density (including the reflected ions),

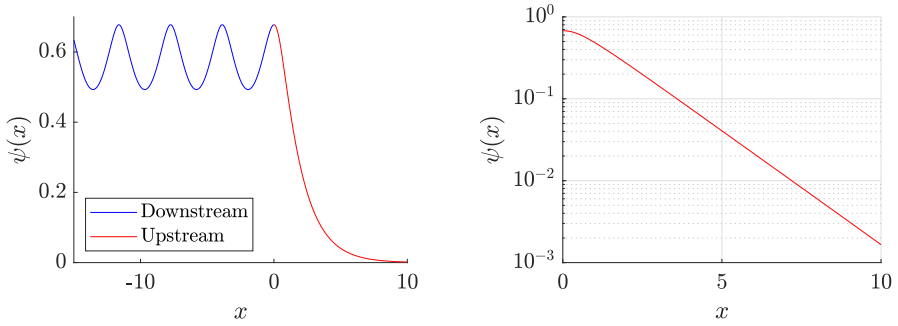
$$\begin{aligned} n_{\mathrm{j},1} &= \int_{-\infty}^{v_0} \mathrm{d}v\, f_\mathrm{j}(v, \psi = 0) \\ &= \frac{n_{\mathrm{j},0}}{2}\left\{1 + 2\,\mathrm{erf}\left[\sqrt{\frac{\tau_\mathrm{j}}{2\zeta_\mathrm{j}}}\mathscr{M}\right] + \mathrm{erf}\left[\sqrt{\frac{\tau_\mathrm{j}}{2\zeta_\mathrm{j}}}\left(\sqrt{2\zeta_\mathrm{j}\psi_\mathrm{max}} - \mathscr{M}\right)\right]\right\}, \end{aligned} \tag{15}$$

where $v_0 = \sqrt{2\zeta_\mathrm{j}\psi_\mathrm{max}}$. We therefore get quasi-neutrality by setting

$$n_{\mathrm{e},1} = \sum_\mathrm{j} Z_\mathrm{j} n_{\mathrm{j},1}. \tag{16}$$

3 The analytical approximation scheme

To export the shock to Gkyl, we need some expression for $\psi(x)$ and $E(x)$ which can be evaluated at an arbitrary x point. However, sine the semi-analytical model only produces ψ and E values at discrete points, we have to create some kind of analytical approximation.



(a) Down- and upstream

(b) Upstream with log scale

Figure 1: Example of the electrostatic potential of a shock (a). In (b) the upstream portion of the shock is shown in a log scale, to show that $\psi(x)$ decays exponentially for high x.

3.1 Downstream approximation

The downstream part of the shock is oscillating with a fixed wavenumber, as we can see in Figure 1a. This means that we can express $\psi(x < 0)$ with a Fourier series. Since we further know that $x = 0$ is a maximum of ψ, $\mathrm{d}\psi/\mathrm{d}x$ must be 0 at $x = 0$, meaning that we only need a cosine expansion.

Since the raw data is discrete, it can not be used directly to find the Fourier coefficients. Instead a cubic spline interpolation is calculated on all the downstream data points. The coefficients are calculated using fft on one wavelength, which is densely sampled (1024 points) with the spline.

To be able to find the precise wavelength, λ, of the oscillations, the local maxima and minima of $\psi(x)$ are calculated analytically from the spline interpolation. This is done using the function `spline_extrema.m` [sec. 5.1.4]. Given the wavelength of the oscillation, we can also define a base frequency $K = 2\pi = \lambda$ of the Fourier series.

In the end, the approximate downstream potential becomes

$$\tilde{\psi}(x < 0) = \sum_{i=0}^{n_\mathrm{F}} C_i \cos(iKx), \tag{17}$$

where n_F is the number of Fourier modes to use in the approximation. This can also easily be differentiated to give the approximate electric field and charge density,

$$\begin{aligned} \tilde{E}(x < 0) &= -\frac{\mathrm{d}\tilde{\psi}}{\mathrm{d}x} = \sum_{i=1}^{n_\mathrm{F}} iKC_i \sin(iKx), \\ \tilde{\rho}(x < 0) &= -\frac{\mathrm{d}^2\tilde{\psi}}{\mathrm{d}x^2} = \sum_{i=1}^{n_\mathrm{F}} (iK)^2 C_i \cos(iKx), \end{aligned} \tag{18}$$

3.2 Upstream approximation

In the upstream we can use the fact that the potential is exponentially decaying for large x, Figure 1b, but we also need to take into account the behavior near the peak. To do

that we fit a rational function $r(x) = p(x)/q(x)$ to $\log(\psi(x))$. To take the asymptotic behavior into account we can choose

$$\begin{aligned} p(x) &= B_2 x^n + B_1 x^{n-1} + \log(\psi_{\max}) + \sum_{k=2}^{m} A_k x^k \\ q(x) &= x^{n-1} + 1, \end{aligned} \tag{19}$$

where B_2 and B_1 are given by the linear asymptote of $\log(\psi(x))$. The constant term, $\log(\psi_{\max})$, is there to get the right value at $x = 0$. The fit is then done on the remaining polynomial coefficients A_k, by fitting $p(x)$ to $q(x)\log(\psi(x))$. Notice that there is no A_1, since we want $\mathrm{d}r/\mathrm{d}x = 0$ at $x = 0$. To determine the values of B_1 and B_2, a cutoff point, $x = x_{\mathrm{co}}$, has to be chosen above which a linear fit is made to $\log(\psi(x))$.

Spelled out fully, we get the upstream approximation as

$$\tilde{\psi}(x > 0) = \mathrm{e}^{r(x)} = \exp\left(\frac{p(x)}{q(x)}\right) = \exp\left(\frac{B_2 x^n + B_1 x^{n-1} + \log(\psi_{\max}) + \sum_{k=2}^{m} A_k x^k}{x^{n-1} + 1}\right), \tag{20}$$

and the approximate electric field and charge density are

$$\tilde{E}(x > 0) = -\frac{\mathrm{d}\tilde{\psi}}{\mathrm{d}x} = r'(x)\mathrm{e}^{r(x)} = -r'(x)\tilde{\psi}(x), \tag{21}$$

and

$$\tilde{\rho}(x > 0) = \frac{\mathrm{d}\tilde{E}}{\mathrm{d}x} = -r''(x)\tilde{\psi}(x) - r'(x)\tilde{E}(x), \tag{22}$$

where "prime" denotes derivatives and

$$r'(x) = \frac{\mathrm{d}r}{\mathrm{d}x} = \frac{p'}{q} - \frac{pq'}{q^2} = r(x)\left[\frac{p'(x)}{p(x)} - \frac{q'(x)}{q(x)}\right] \tag{23}$$

and

$$r''(x) = \frac{\mathrm{d}^2 r}{\mathrm{d}x^2} = r(x)\left[\frac{p''}{p} - \frac{q''}{q} + 2\left(\frac{q'}{q}\right)^2 - 2\frac{p'q'}{pq}\right]. \tag{24}$$

3.2.1 Problem with too high degree polynomials

There are a couple of free parameters to choose when making an approximation like this. One of them is the degree, n, of the polynomial to use. The most intuitive idea is that the higher n and m are the better the fit will be. But there is a very real risk of over-fitting the polynomials, i.e. introducing artificial variations between data points. This problem of over-fitting is especially noticeable in the derivatives. So, since we need to input the electric field into the initial conditions in Gkyl, we have to be careful to check that the electric field does not show too many signs of over-fitting.

The best way to prevent errors due to over-fitting, is to check both the fit of $\psi(x)$ and $E(x)$. If the fit is bad, then try with a different n. (Usually the best fit is with just keeping $m = n - 2$.) It also need not be that n is too high, sometimes there are some sweet-spots in the range $n = 15{-}25$. Another trick is to try with ever so slightly different values of $B_{1,2}$, by changing the asymptotic cutoff point, x_{co}.

4 How to use this package (in progress)

IMPORTANT NOTE: You must have an environment variable `SHOCKLIB` leading to the ShockLib directory (e.g. `/home/andsunds/SVN/erc/andsunds/Shock-project/ShockLib`), to run the examples in this package. To test if you have this environment variable run the command:

```
$: echo $SHOCKLIB
```

If you do not have it (above command returns empty), run:

```
$: export SHOCKLIB=<path to Shock library directory>
```

This line can also be added to your `.bashrc` file or equivalent.

In this section we give a short walk-through of the process of running a Gkyl simulation of the semi-analytical shocks – all the way from choosing the shock parameters to the Gkyl initialization script.

4.1 The Matlab part

The semi-analytical shocks are implemented in Matlab through a shock class, so all calculations regarding the shock is done through a *shock object*. This shock object is initialized with the input parameters:

`Z` – ion charge number (may be a vector)

`m` – ion mass (may be a vector)

`n` – ion initial density (may be a vector)

`tau` – electron-to-ion temperature ratio $(T_e/T_i)^3$

`Mach` – the shock Mach number

`tol` – the numerical tolerance wanted

The shock initialization also requires an initial guess of ϕ_{max} and ϕ_{min}, `psimaxmin_in`. The closer this guess is to the true value, the higher the chances are of numerical convergence and the faster the shock calculation will run.
To initialize a shock object use e.g.

```
Sh0=Shock_MB(Z,m,n, tau, Mach, psimaxmin_in, tol);
```

`Sh0` is now a shock object of the class `Shock_MB`, and displaying it should produce an output like:

```
Shock_MB with properties:

       tol: 1.0000e-09
         Z: 1
```

```
        m: 1
        n: 1
     taui: 200
        M: 1.3000
   psimax: 0.6775
   psimin: 0.4929
```

Note that `taui` (τ_j) is listed here instead of the input `tau`. If any of the values listed here are `NaN`, then the code did not converge, and you need to try some other `psimaxmin_in` (or combination of `Mach` and `tau`).

Shock classes There are currently two shock classes supported in the `Shock_pkg_new` package:

- `Shock_MB(Z,m,n, tau, Mach, psimaxmin_in, tol)`
  the simplest type of shock, with Maxwell-Boltzmann distributed electrons.
- `Shock_col(Z,m,n, tau, Mach, t0, nustar, psimaxmin_in, tol)`
  which is our collisional shock model.

Note that the different classes require different input parameters.

There are also classes for electron trapping under development in this new package. (There already exists electron trapping classes in the original shock package, `Shock_pkg`, but not in this one with the new normalization.)

4.1.1 Calculating the electrostatic potential

Using this shock object, the electrostatic potential is then easily calculated with the built-in function

`[X,psi]=Sh0.find_psi(xmin,xmax)`,

which operates on the shock object `Sh0`, and calculates its electrostatic potential in the range `xmin` to `xmax`.

When calculating the electrostatic potential, we cannot use a too large range in x. This is because at large positive x, where ψ is very close to zero, numerical errors can affect the result and become unstable. An example of this is shown in Figure 2. To avoid these types of errors, we have to manually check that the potential looks okay. There is also a tendency to get complex numerical values when this happens.

If we want to, we can also get the electric field and charge density at the same time by running:

`[X,psi,E,rho]=Sh0.find_psi(xmin,xmax)`

The electric field and charge density are useful, but not required, if we also want to find the approximation later on.

[3] This is similar to τ_j, but without the factor Z_j. Also, as of now, different ion temperatures has not been implemented.

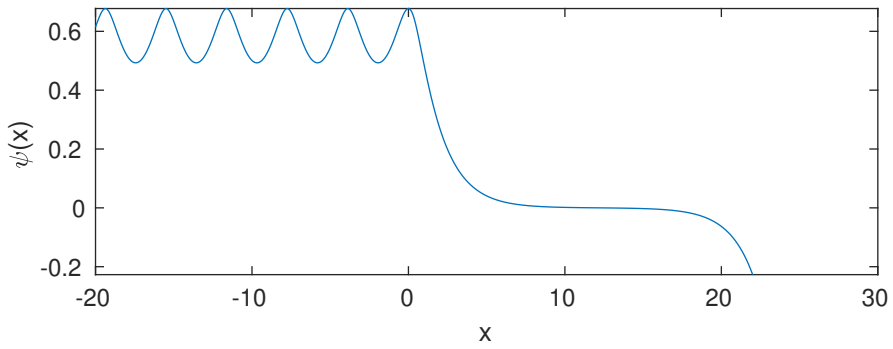


Figure 2: Erroneous potential, ψ, due to numerical error which manifest at large positive x values.

4.1.2 Finding the exportable approximation

Now that we have found ψ, E, and ρ, we can use them to find the approximations described in Section 3. To do that we have the function

```
[Cm, K, Cp] = find_approx(X,psi,E,rho,N_sample,exp_threshold,poly_deg)
```

which takes in the discrete `X`, `psi`, `E`, and `rho` and calculates the approximation coefficients in the negative x range, `Cm`, as well as for positive x, `Cp`. In the negative x range, `K` is the base frequency of the cosine series. The positive approximation coefficients, `Cp`, are just all polynomial coefficients of $p(x)$ arranged in increasing powers.

For a list of the other input arguments to this function, see Section 5.1.1.

It is possible to run `find_approx` without giving `E` and `rho`. This will work in most cases, but it will be harder to find a good fit.

As was mentioned in Section 3.2.1, there are some manual work required to get a good fit. It is therefore recommended to also use

```
[psi,E,rho] = get_approx(X,Cm,K,nf, Cp)
```

to check that the approximation functions are satisfactory (also in E), and if needed experiment with `poly_deg` and `exp_threshold`. This function takes the approximation coefficients, `Cm` and `Cp`, and returns the approximate values of ψ, E, and ρ at the points specified in `X`. The argument `nf` is the number of frequencies (cosine terms) to use for the DS approximation.

Export protocol When we have found a satisfying approximation, we have to save the coefficients to file. This is done in a function

```
[save_data] = save_coefs(save_path,Cm,K,Cp)
```

which saves the coefficients in a tsv-file with the correct format [sec. 5.1.3]. The `save_path` is a string with the path and fime name, to which the coefficients will be saved. Also, `Cm` should only contain the `nf` number of coefficients as you want to save.

When saving, you will be prompted if you want to save the coefficients to the path provided or not.

4.1.3 Other things to do with the shock classes

We do, of course, not all the time want to calculate $\psi(x)$. This is a bit outside the scope of these instruction notes but some other functions in the `Shock_pkg_new` package are

- `[Sh_cell] = t_sweep(pre_calc_Shock,i0,N_steps,t0,output_file)`
  A function to be used with `Shock_col` objects, for sweeping through time to see how the collisions evolve the shock. The output is a cell list filled with shock objects for each time step.

- `[Sh_cell] = tau_sweep(pre_calc_Shock,i0,N_steps,tau0,dtau)`
  A function for sweeping through different `tau` values, and recording how the shock behavior changes.

- `[Sh_cell] = Mach_sweep(Sh_handle,pre_calc_Shock,i0,N_steps,M0,dM)`
  A function for sweeping through different `Mach` values.

- `[Sh_cell] = M_tau_sweep(pre_calc_Shock,iT0,TT,iM0,M0,dM,NM)`
  This function does a 2D scan for different `Mach` and `tau` values. For each `tau` value, the function sweeps through the `Mach` range.

In each of these scanning/sweeping functions, each new step is initialized with the `psimax` and `psimin` of the previous step. In all but the time scan, the function tries a step and if it doesn't work, takes a half step back and tries again. By doing this the ends of the each corresponding range can be studied with less manual supervision.

4.2 The Gkyl part

Once the approximation coefficients have been saved in the proper format, we can head over to Gkyl to import the shock. With the help of the **shApprox** package [sec. 5.2.1], this is done quite easily. First, we need to add it to the package path, which is done with the line

```
package.path = package.path..";"..<path to shApprox.lua>
```

Then we also need to import the package:

```
local Approx = require "shApprox"
```

In our example the path to the package can be written as

```
os.getenv("SHOCKLIB").."/luaLib/?.lua"
```

This adds any file in the path `$SHOCKLIB/luaLib/`, ending with `.lua`, to the list of packages. (The operator `..` is string concatenation in Lua.) Another central part of the **shApprox** package is the coefficient file, whose filename is passed around a lot. It is therefore a good idea to also have a variable

```
local filename = "<path to coef file>"
```

with the path and file name of the coefficient file.

The next step is to set all the parameters for the simulation. Please see the example input file for a more details of the way of doing this. In some more general terms, one way of setting up the input parameters is to use some units when defining, for instance,

the masses of the species, e.g. `me_hat=1` and `mi_hat=1836.153`, and then use them in the normalized quantities as described in Section 2. We must also keep the comments in Section 2.1.1, about ζ and how Gkyl wants mass and charge separately, in mind.

When initializing the species, we have to provide the distribution functions for each species and fields. This is also easily done through the `shApprox` package which has the functions

`fi(x,v, ni0,taui, zetai, M, filename, dv)`,
`fe_MB(x,v,ne1,zetae,M,filename)`, and
`Ex(x, filename)`,

which calculates precisely the distribution functions and fields that we want. The input argument `dv` is the simulation grid size in velocity, and is used to create a more smooth transition to the empty regions of phase-space. Also note that the electron distribution function takes `ne1`, i.e. (16), as its input argument.

For other inquiries regarding the Gkyl part, the reader is referred to the example script and the Gkyl manual, `http://gkyl.readthedocs.io/en/latest/`.

5 Code documentation (in progress)

This package consists of libraries with scripts for Matlab and Lua. The Matlab library concerns the semi-analytical shocks, and fining their approximations. The Lua library is just a single package that takes the approximation coefficients given by the Matlab scripts, and gives the wanted function values in Lua.

5.1 mLib/

The Matlab library has four functions, `find_approx`, `get_approx`, `save_coefs`, and `spline_extrema`. All exept `spline_extrema` are directly used by the user.

5.1.1 `find_approx.m`

```
[Cm, K, Cp] = find_approx(X,psi,E,rho,N_sample,exp_threshold,poly_deg)
```

This function calculates the approximation coefficients, `Cm`, `K`, and `Cp`, according to the approximation schemes described in Section 3.

This function takes in the true values of `psi`, `E`, and `rho`, this is the recommended way of running this function. However, this function can be run with only `psi` as input, although the US fit will not be as good. The other input arguments are

- `N_sample` is the number of sample points to be used in the FFT of the DS oscillation.
- `exp_threshold` is the cut-off point, x_{co}, above which ψ is said to be decaying exponentially.

- `poly_deg` is the degree, n, of the numerator polynomial, $p(x)$; `poly_deg` may also be a two component vector, in that case, the second element is the degree, m, of the fitted part of $p(x)$ [see (19)].

Downstream approximation Since the DS oscillation is periodic, we should be able to fit a Fourier series to it. We also know that $\mathrm{d}\psi/\mathrm{d}x\,|_{x=0} = 0$, which leaves us with only the cosine terms (i.e. the real part of complex F-coefficients). However, as the data points are unevenly spaced, and not aligned with the exact period of the oscillation, we first need to create a cubic spline interpolation to get the exact period. Then we use the dedicated function, `spline_extrema`, that finds the analytical min/max points of the cubic spline of the DS oscillations. With the location of the extremum points, we can determine the wavelength, λ, of the first (closest to $x = 0$) oscillation. (We want to use the first oscillation, since that has the least numerical errors, from the ODE solver, in it.)

Then we create a densely sampled version of $\psi(x)$ in this first wavelength, with `N_sample` sampling points, using the spline interpolation. We can then find the F-coefficients with FFT. Finally we convert the complex Fourier transform into a cosine transformation by only choosing the real part of the complex coefficients, and also doubling the non-zero frequency terms. The FFT does give some imaginary parts to the coefficients, but they are a couple of orders of magnitude smaller that the real parts, so it is indeed safe to discard the sine terms of the Fourier series.

Upstream approximation We want to fit the rational function, $r(x) = p(x)/q(x)$, where p and q are given by (19), to $\log(\psi(x))$. First, B_1 and B_2 are chosen by finding a linear fit to $\log(\psi(x))$ for $x >$ `exp_threshold`. Then the lower order coefficients, A_k, are fitted using a least square fit on all the data points.

If `E` and `rho` are given at the input, the first two coefficients, A_2 and A_3, are set to be the corresponding Taylor coefficients of

$$q(x)\log(\psi(x)) = (x^{n-1} + 1)\,\log(\psi(x)), \tag{25}$$

which are

$$\begin{aligned} A_2 &= \frac{1}{2\psi_{\max}}\left.\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2}\right|_{x=0} = \frac{\rho(x=0)}{2\psi_{\max}} \\ A_3 &= \frac{1}{6\psi_{\max}}\left.\frac{\mathrm{d}^3\psi}{\mathrm{d}x^3}\right|_{x=0} = -\frac{1}{6\psi_{\max}}\left.\frac{\mathrm{d}\rho}{\mathrm{d}x}\right|_{x=0}. \end{aligned} \tag{26}$$

The rest of the coefficients are then fitted with zero as the initial guess for all of them. If, on the other hand, `E` and `rho` are not given at the input, then also A_2 and A_3 are subject to fitting.

5.1.2 `get_approx.m`

```
[psi,E,rho] = get_approx(X,Cm,K,nf,Cp)
```

This is a function that gets the value of the approximation [sec. 3] in the points `X`, given

the approximation coefficients `Cm` and `Cp`. The other arguments, `K`, is the frequency of the downstream oscillations, and `nf` is the number or Fourier modes to use in the approximation. (Usually Cm is 1024 elements long, and we only need the first $\mathcal{O}(10)$ modes to get an accurate representation.) The main use of this function is to display the result of `find_approx` to manually inspect how good the fit is.

Thi function can also calculate the approximate electric field, `E`, and the charge density, `rho`. This is done analytically, with the analytical derivatives of the approximation functions.

5.1.3 save_coefs.m

```
[save_data] = save_coefs(save_path,Cm,K,Cp)
```

Arranges the approximation coefficients into a matrix, `save_data`, and propts you if you want to save it to the path provided in `save_path`. The format in which the coefficients have to saved are as follows:

```
 K      Cp_0
Cm_0    Cp_1
Cm_1    Cp_2
 |       |
 0      Cp_n
```

That is, two columns, with the first one starting with the base frequency of the DS oscillations, `K`, and then followed by the coefficients of the cosine series, `Cm`. The second column instead consists of all the polynomial coefficients of $p(x)$.

The downstream coefficient vector, `Cm`, should only contain the Fourier coefficients that you want to save. So the first column ends with zeros, since in general not as many cosine coefficients are needed as polynomial coefficients. The first column may be as long as, *but not longer than*, the second column in terms of values given. I.e. **the second column may not end with zeros.** This is because the Lua package has no way of knowing the degree, n, of the polynomial, $p(x)$, other than looking at the length of the second column.

5.1.4 spline_extrema.m

This is a function that finds all the extrema of a spline object by analytically calculating the roots of the derivative of each spline:

```
[x0,y0] = spline_extrema(pp_spline)
```

`x0` and `y0` are the x and y coordinates of the extremum points, and `pp_spline` is a Matlab spline pp-object.

To find a local extremum of a cubic spline interpolation, we use the fact that the interpolations consists of multiple cubic polynomials of the form

$$S_n(x) = a_n(x - x_n)^3 + b_n(x - x_n)^2 + c_n(x - x_n) + d_n \quad \text{for} \quad x_n \leq x < x_{n+1}. \tag{27}$$

The roots of the derivatives are simply given by the quadratic formula

$$x_n^{\pm} - x_n = \frac{-b_n \pm \sqrt{b_n^2 - 3a_n c_n}}{3a_n}. \tag{28}$$

We call these the pseudo roots, since not all of them represent actual extremum points of the spline interpolation. The proper roots of the spline derivatives are the ones that fall inside the interval of their respective polynomial, $x_n \leq x_n^{\pm} < x_{n+1}$.

5.2 luaLib/

The Lua library only contains one single file, **`shApprox.lua`**, which is a package of the different lua functions we would want to use in Gkyl.

5.2.1 `shApprox.lua`

This script file contains several different function which can be used by importing this file as a package.

All functions in the user interface takes in a `filename` argument, which is the path and name of the file containing the approximation coefficients. This file must be formatted correctly [sec. 5.1.3].

All the functions that take an input `x`, have the convention that the boundary between up- and downstream, is at `x = 0`.

`lambda(filename)`
This function returns the wavelength of the DS oscillations, by simply using the frequency that is given in `filename`.

`psi(x, filename)`
Calculates the approximate electrostatic potential [sec. 3] at the point `x`, given the approximation coefficients in the file `filename`.

`psimax(filename)`
Calculates the maximum value of the electrostatic potential, given the coefficients in `filename`. Gives the same result as `psi(0,filename)`.

`Ex(x, filename)`
Calculates the approximate electric field analytically at the point `x`, given the approximation coefficients in the file `filename`.

`fi(x,v,ni0,taui,zetai,M, filename, dv)`
Calculates the ion distribution function (5) at the phase-space point `(x,v)`.

This function does also take into account the ion reflection, meaning that the distribution function gets cut off above $v_0 = \sqrt{2\zeta_j(\psi - \psi_{max})}$ in the upstream ($-v_0$ in the downstream). However, to prevent large errors due to the velocity discretization in Gkyl, the cut-off is made soft by having a Gaussian decay, `math.exp(-10*(v-USDS*v0)^2/dv^2)`. The Gaussian starts at the cut-off point, `v-USDS*v0`, where `USDS = 1` (`-1`) in the upstream (downstream), and where `dv` is the discretization step size.

`ni1(ni0,taui,zetai,M, filename)`
Calcualtes the far-upstream ion density (15) by numerically integrating the ion-distribution, (5), with $\psi = 0$ from $-\infty$ to $v_0 = \sqrt{2\zeta_\mathrm{j}\psi_\mathrm{max}}$.

`fe_MB(x,v,ne1,zetae,M, filename)`
Calcualtes the Maxwell-Boltzmann electron distribution function (14). Note that `zetae` is negative since $Z_\mathrm{e} = -1$, and hence

$$\zeta_\mathrm{e} = \frac{Z_\mathrm{e}/\hat{m}_\mathrm{e}}{\hat{c}_\mathrm{s}^2/\hat{T}_\mathrm{e}} = -\frac{\hat{T}_\mathrm{e}}{\hat{m}_\mathrm{e}\hat{c}_\mathrm{s}^2} \sim -2000 < 0. \tag{29}$$

References

[1] A. Sundström, "Effects of electron trapping and ion collisions on electrostatic shocks," MSc thesis, Chalmers University of Technology, 2018.