Dust grain potential calculator

User manual

Dogan Akpinar and George E. B. Doran

1 Setup

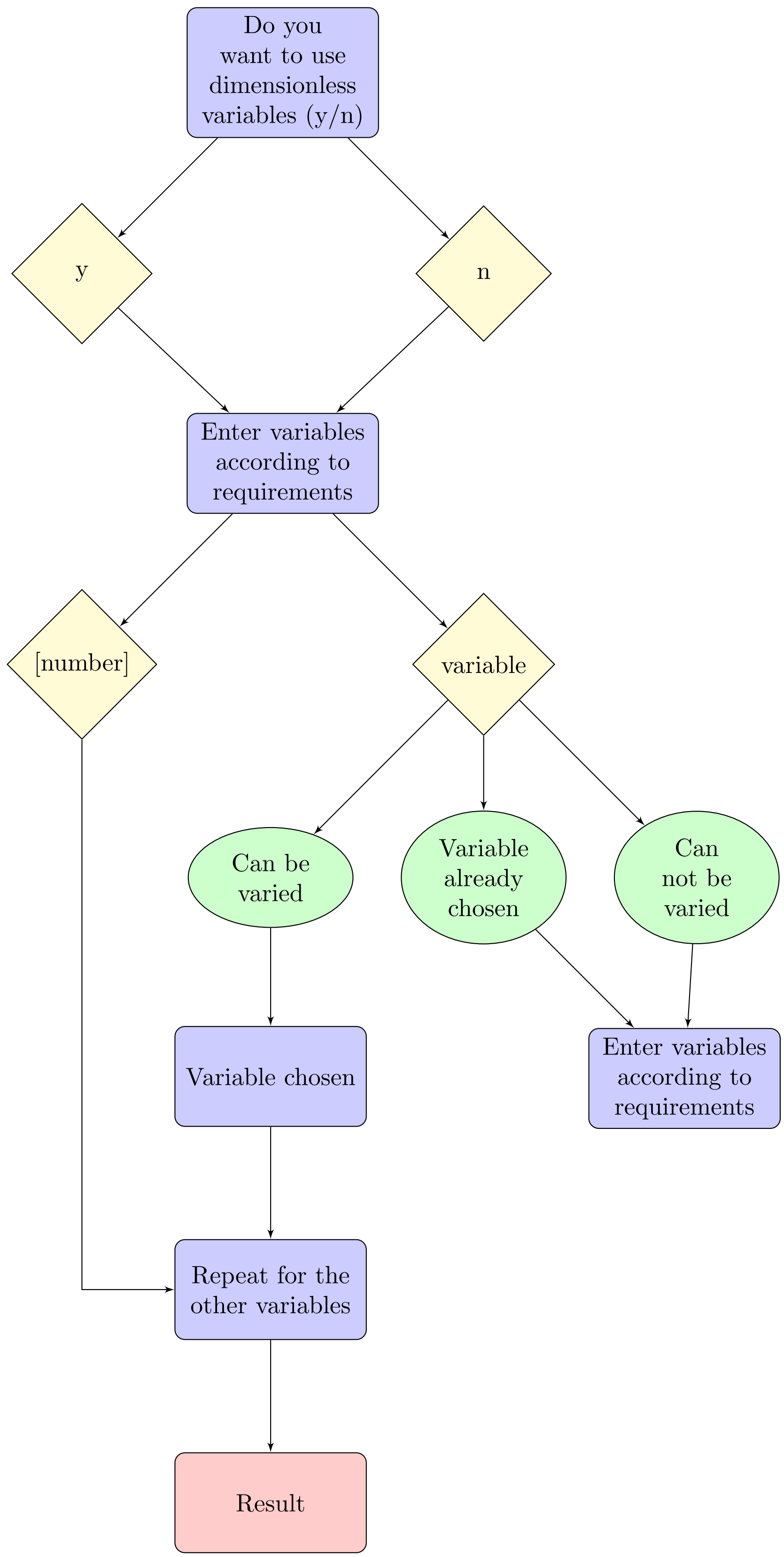
- Install the modules specified in *requirements.txt*
- Open *Dust_grain_potential_calculator.py* and edit the base path on line 8

2 Variable table

Variable name	Unit	Requirements	Normalised variable name	Normalisation factor	Default value	Variable
Electron temperature (T_e)	K	$T_e > 0$	-	-	-	Yes
Ion temperature (T_i)	K	$T_i \geq 0$	Θ	T_e	-	Yes
Relative ion charge (z)	-	$0 < z \leq z_{max}$ z $\in \mathbb{Z}$	-	-	-	No
Ion mass (m_i)	kg	$m_i > 0$	μ^2	m_e	-	No
Electron number density at infinity (n_0)	m^{-3}	$n_0 > 0$	-	-	-	No
Dust grain radius (a)	m	$a \geq 0$	α	$\lambda_D = \sqrt{\frac{\varepsilon_0 k_B T_e}{n_0 e^2}}$	-	Yes
Flow speed (v)	ms^{-1}	v ≥ 0	υ	$\upsilon_B = \sqrt{\frac{z k_B T_e}{m_i}}$	0	Yes

- The potential ϕ is normally negative and is normalised to $\Phi = -\frac{e\phi}{k_B T_e}$ which is a positive quantity.

3 Using the code



To run the code, see the flow diagram above where the yellow boxes show exactly what to input into the terminal except [number] which means type a number. It is possible to use the SI prefixes as shown in the following table.

Prefixes	Value
Y	1×10^{24}
Z	1×10^{21}
E	1×10^{18}
P	1×10^{15}
T	1×10^{12}
G	1×10^{9}
M	1×10^{6}
k	1×10^{3}
m	1×10^{-3}
u	1×10^{-6}
n	1×10^{-9}
p	1×10^{-12}
f	1×10^{-15}
a	1×10^{-18}
z	1×10^{-21}
y	1×10^{-24}

It is also possible to type $5 * 10 \wedge 6$ for example which will be interpreted as 5×10^6. Entering temperatures with "ev" after the number will allow the temperature to be inputted in electron volts, for example, 5ev will be interpreted as $57970K$. Typing "variable" will prompt the user to enter a maximum and minimum value which will produce a graph rather than a number as the result. Once "variable" has been inputted once it will be impossible to set another parameter as the variable.

4 Adding a new variable

To add a new variable, the procedure is as follows:

Copy the dictionary for flow speed on line 255 and paste it underneath the existing one. Change the values to reflect the new variable and remove all that do not apply. The requirement classes found above the dictionaries can be used to make requirements, to find how to do this, the other dictionaries in the code should provide sufficient examples.

```python
nfv = Norm_v()
v_dict = {
    "var_name": "Flow speed",
    "Requirements": [GreaterThanEqualTo(0)],
    "Norm_factor": nfv,
    "Norm_var_name": "upsilon",
    "Unit": "meters per second",
    "Unit_symbol": "ms^-1",
    "Graph_unit_label": "[$ms^{-1}$]",
    "isvariable": True,
    "default value": 0,
}
```

All new variables should have a default value corresponding to the case where they are not considered, for example, not considering flow is to have a flow speed of zero. If a normalisation factor is required that depends on the other variables, then an object must be created above the new dictionary. This object will require a new class, to make this copy and paste the class for the flow velocity.

```python
189 class Norm_v(Norm):
190     '''The normalisation factor for the flow speed'''
191     def getnormfactor(self):
192         _T_i = self.getvarvalue("Ion temperature")
193         _m_i = self.getvarvalue("Ion mass")
194         if _T_i == 0:
195             return 1
196         return np.sqrt(2 * k_B * _T_i / _m_i)
197
```

Change the variables it collects for the ones required to make the normalisation factor. Note that if a value within the normalisation factor can be zero then it will cause a dividing by zero error, so something such as the code on lines 194-195 should be added to prevent this.

Once the dictionary is complete add the name of the new dictionary to the list of dictionaries on line 269

```
269    dict_list = [T_i_dict, T_e_dict, z_dict, m_i_dict, n_0_dict, a_dict, v_dict]
```

5 Adding a new model

To add a new model, open the models folder and create a new model as a *.py* file. The name of the file must be the same as the name of the model. The model requires at least five functions; get_name(), colour(), get_info(), potential_finder() and priority(). For simplicity, copy and paste *OML.py* into the new model and change the details.

```
def get_name():
    return "OML"

    colour()
def colour():
    return "orange"
```

Change the name in get_name() to the new name and change the colour to whatever colour you want the model to be displayed as on a graph.

```
def get_info():
    assumptions_list = [
        "Model assumptions:\n",
        "Spherical symmetry\n",
        "No collisions\n",
        "No magnetic field\n",
        "No external electric field\n",
        "No electron emission of any kind\n",
        "Quasi-neutrality in bulk plasma\n",
        "Conservation of particle energy\n",
        "Conservation of particle angular momentum\n",
        "Limiting trajectory is the grazing incidence\n",
    ]
    validity_list = [
        "Validity:\n",
        "Static plasma\n",
        "Small dust\n",
        "Any ion temperature\n",
    ]
    reference_list = [
        "References:\n",
        "C. T. N. Willis, “Dust in stationary and flowing plasmas,” Physics PhD Thesis, Imperial College London, Ma
        "D. M. Thomas, “Theory and simulation of the charging of dust in plasmas,” Physics PhD Thesis, Imperial Col
        "K. R. V. and A. J. E., “The floating potential of spherical probes and dust grains. ii: Orbital motion the
    ]
    string = (
        " ".join(assumptions_list) + " ".join(validity_list) + " ".join(reference_list)
    )
    return string
```

The function get_info() should include all the assumptions made by the model, the range of parameters it is valid over, a reference to where to find the model and any other information that the user may wish to know about the model.

```python
def potential_finder(dictionarylist):
    for _vardict in dictionarylist:
        if _vardict.get("Norm_var_name") is not None:
            if _vardict.get("Norm_var_name") == "alpha":
                alpha = _vardict.get("Norm_value")
            elif _vardict.get("Norm_var_name") == "z":
                z = _vardict.get("Norm_value")
            elif _vardict.get("Norm_var_name") == "mu":
                mu = _vardict.get("Norm_value")
            elif _vardict.get("Norm_var_name") == "upsilon":
                upsilon = _vardict.get("Norm_value")
            elif _vardict.get("Norm_var_name") == "Theta":
                Theta = _vardict.get("Norm_value")

    Phi = bisect(OML_function, -10, 10, args=(Theta, mu, z, alpha, upsilon))
    return np.absolute(Phi)

```

The potential_finder() function collects all the variables it needs to calculate the potential like so. All required variables should be collected in this way. Note the output should have the potential normalised in the same way as the existing models, $\Phi = -\frac{e\phi}{k_B T_e}$.

```python
def priority(dictionarylist):
    P = 0
    for _vardict in dictionarylist:
        if _vardict.get("Norm_var_name") is not None:
            if _vardict.get("Norm_var_name") == "alpha":
                alpha = _vardict.get("Norm_value")
            elif _vardict.get("Norm_var_name") == "z":
                P += 1
            elif _vardict.get("Norm_var_name") == "mu":
                P += 1
            elif _vardict.get("Norm_var_name") == "Theta":
                Theta = _vardict.get("Norm_value")
                if Theta >= 1e-4:
                    P += 1
                else:
                    P += 0.5
            else:
                if _vardict.get("Norm_value") != _vardict.get("default value"):
                    return 0
    if alpha > 1.25 * Theta ** (0.4):
        return 0
    P += 1
    return P

```

The priority function deals with which model will be chosen for a given set of inputs. For the new priority function the variables should be collected as shown, if they are within the accepted range the priority value should increase by one and return a zero otherwise. Any input without a restriction should still be added and increase the priority function by one. To account for variables added after the creation of the model lines 85-87 should be included such that if there is an input variable not in the list of expected input variables that is not equal to its default value then the priority will be set to zero and the model will not be used. For example if the flow speed is not zero (its default value) then OML will have a priority of zero and will not be used.