Notes about the internship at IRAP

Improving a Martian subsurface model for a GCM

Arnau Prat Gasull

March to July 2020

Contents

1 Introduction and goals

The LMDZ Global Circulation Model[1] is widely used to model the atmophere of Mars. However, current versions do not take into account the effect of time-dependant/active regolith, which may have a notable influence on the results of the simulations. Researcher P.-Y. Meslin from IRAP (CNRS) undertook the task of developing a module for the LMDZ GCM that feeds in data of subsurface temperatures and water/ice presence to simulate active regolith.

Because the final goals of the project are to understand (a) how the ice came to be distributed the way that it is around Mars, (b) how ice changes short-term—say throughout the day, short-term—and long-term (e.g. thousand years), and (c) how these affect the climate; detailed knowledge of the impact of regolith (Martian soil) on the adsorption process[2] is necessary. Note that on Mars, the adsorption is strong due to high specific surface area SSA[3] and low temperature, and therefore it may be important to couple this phenomenon with the LMDZ GCM. The main object of this internship is—concretely—to try to improve the subsurface model by bettering the integration of the adsorption.

$$n_{\text{ads}} \propto n_{\text{gas}} \cdot SSA \cdot \exp\left(\frac{\Delta H}{RT}\right)$$

where n_{ads} and n_{gas} are the adsorbed and gas phase amounts, respectively, ΔH the enthalpy of adsorption, and SSA the specific surface area in m^2/kg.

2 Overview of the work to be done

Current simulations with active regolith follow the scheme below:

[1] Alternatives to the LMDZ GCM are: the NASA Ames Mars GCM, GEM-Mars.

[2] Adsorption is a fully reversible process where molecules *stick* to the surface without the presence of a chemical reaction that can be quantified as:

[3] The value of the specific surface area for the Martian soil is not currently known, but a measure of 17 m^2/g is often used. P.-Y. Meslin found an upper limit of 30 m^2/g which is close to the measured value, but simulations with the subsurface model have even used 100 m^2/g for the SSA for research purposes (Weinmann 2019).

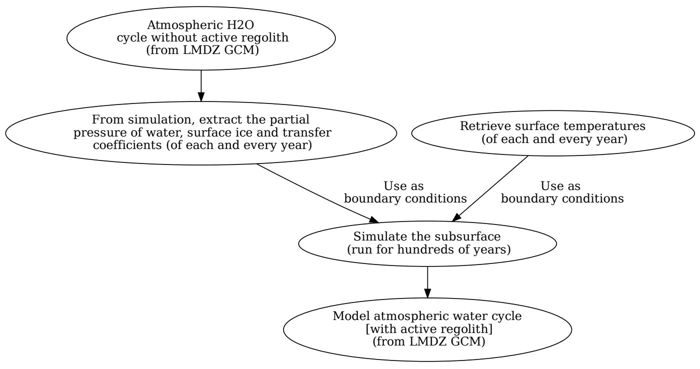

Atmospheric H2O cycle without active regolith (from LMDZ GCM)
From simulation, extract the partial pressure of water, surface ice and transfer coefficients (of each and every year)
Retrieve surface temperatures (of each and every year)
Use as boundary conditions
Use as boundary conditions
Simulate the subsurface (run for hundreds of years)
Model atmospheric water cycle [with active regolith] (from LMDZ GCM)

Figure 1: Basic scheme (Weinmann and Meslin 2019)

In Weinmann and Meslin (2019) further information about the subsurface model used is presented. This subsurface model implements the following equation:

$$\frac{\partial\left(\varepsilon\left(S_w\right)n_{\text{vap}}+n_{\text{ads}}+n_{\text{ice}}\right)}{\partial t}=\frac{\partial}{\partial z}\left(D_b\left(p,T,S_w\right)\frac{\partial n_{\text{vap}}}{\partial z}\right) \tag{1}$$

where n are the vapor, adsorbed and ice phases in molecules/kg, T is the temperature, p is the pressure, D_b the bulk diffusion coefficient, S the specific surface area, $Q_{\text{ads}} \in \{21, 34, 60\}$ kJ mol^{-1}, S_w the water saturation level in %, $\varepsilon = \varepsilon_0\left(1-S_w\right)^2$ the porosity, and θ the surface coverage of adsorbed molecules in %.

The current numerical implementation of this equation assumes that the relationship between the amount of water adsorbed n_{air} and the partial pressure of water $p_{\mathrm{H_2O}}$ is of Langmuir-type (see below) and approximates the linear part of this Arrhenius-type equation by means of a line that ends at the value of n_{ads} where the original equation tends to (towards saturation, see fig. 2).

The Langmuir isotherm (Langmuir 1918, 1932)[4] describes the how the molecules of water ice n_{ads} may lay themselves on the surface of the grain that constitutes the soil, but it is described only as a function of n_{vap} in the form $\frac{n_{\text{ads}}}{n_{\text{vap}}}=k_{\text{ads}}\left(T\right)=k^0_{\text{ads}}\left(SSA,T\right)e^{\frac{Q_{\text{ads}}}{RT}}$. In other words, the Langmuir isotherm only allows for a continuous monolayer of adsorbate molecules to be laid on top of the homogeneous solid surface.

[4]The Langmuir isotherm is also known as the type I isotherm, of the five distinct van der Waals adsorption isotherms. See the introduction in Brunauer et al. (1940). See the derivation of the equation in sec. 10.1.

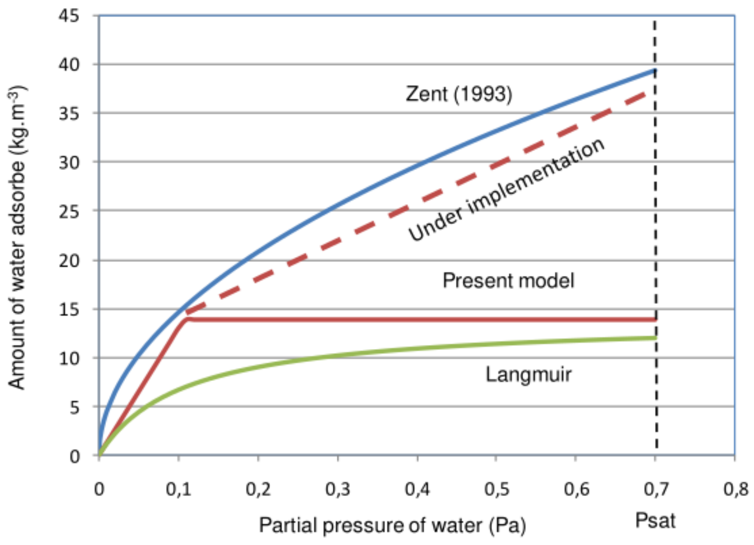


Figure 2: Langmuir equation and its approximation implemented in the current subsurface model (Weinmann and Meslin 2019)

New isotherms have been characterized (BET-type II, Dubinin…) since 1915 (see fig. 3 and fig. 4) and do take into account multimolecular adsorption of gases. In Brunauer, Emmett, and Teller (1938) a new expression is provided, which is an enhancement over/generalisation of type I isotherms. Therefore, type II is preferred over type I, as type I isotherms can be represented with the expression of type II isotherms. Type III isotherms can also be represented using the same equation as the one that describes the type II isotherm, and this equation is the one that will be implemented in the model[5] during the course of the internship.

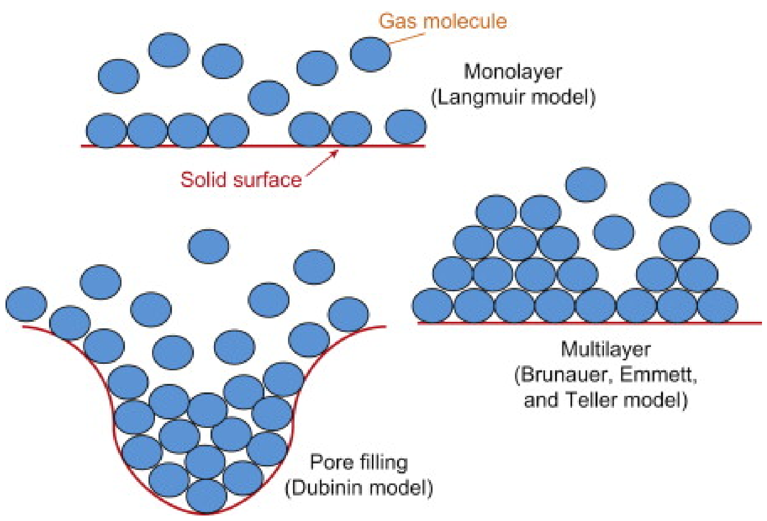


Figure 3: Different adsorption models (Flores 2014)

[5]Remember that the Langmuir/type I isotherm is the one that is currently implemented in the model.

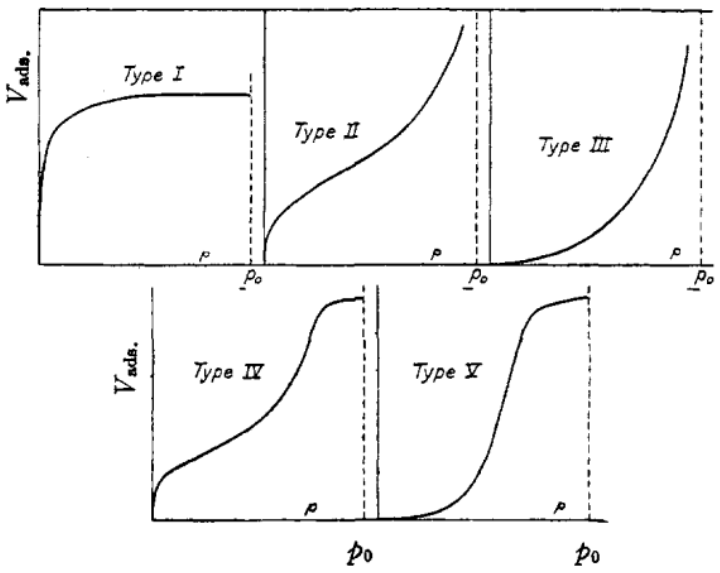


Fig. 1.—The five types of van der Waals adsorption isotherms.

Figure 4: The five types of isotherms (Brunauer et al. 1940)

Because these are isotherms, the shape may depend on the temperature as shown in fig. 5 for the type I isotherm, where the lower the temperature the more pronounced the slope of the *linear part* is. The isotherms are dependent on the vapor pressure of the gas at a given temperature (p_0 in the diagrams in fig. 4). Because the shape of the isotherms is determined by a given temperature and the vapor pressure also dependent on the temperature, one can expect different shapes of the isotherm for different vapor pressures. However, note that if the rate of change of the isotherm is exactly the same as that of the vapor pressure, the shape of the isotherms does not change with respect to p_0 (more in sec. 3.2.1.3).

2.1 Finding an expression for type I, II, and III isotherms

The equation that describe type II and III isotherms for an unlimited number of adsorbed layers, can be written as (Brunauer et al. 1940) (see sec. 10.2 for a derivation of eq. 2):

$$\frac{p}{v\left(p_0 - p\right)} = \frac{1}{v_m c} + \frac{c-1}{v_m c}\frac{p}{p_0} \tag{2}$$

where v is the volume adsorbed at pressure p and absolute temperature T, p_0 is the vapor pressure of the gas (also called saturation vapor pressure in the case of adsorption) at temperature T, v_m is the volume of gas adsorbed when the entire adsorbent surface is covered with a unimolecular layer, and $c \approxeq e^{(E_1 - E_L)/RT}$ with E_1 being the heat of adsorption of the gas in the first adsorbed layer and E_L the heat of liquefaction of the gas. eq. 2 and eq. 3 assume $c \ll 1$ and when $E_1 > E_L$ equations give Type II isotherms whereas Type III isotherms are obtained with $E_1 < E_L$.

For a limited number of layers n (i.e. limited space) and with $x = \frac{p}{p_0}$ (also called relative humidity RH), eq. 2 becomes:

$$v = \frac{v_m c x}{1-x}\frac{1-(n+1)\,x^n + n x^{n+1}}{1+(c-1)\,x - c x^{n+1}} \tag{3}$$

Obviously, eq. 3 can be reduced to eq. 2 when $n \to \infty$. eq. 3 leads to the Langmuir type equation eq. 4 when $n = 1$ is taken into consideration:

$$\frac{p}{v} = \frac{p_0}{v_m c} + \frac{p}{v_m} \tag{4}$$

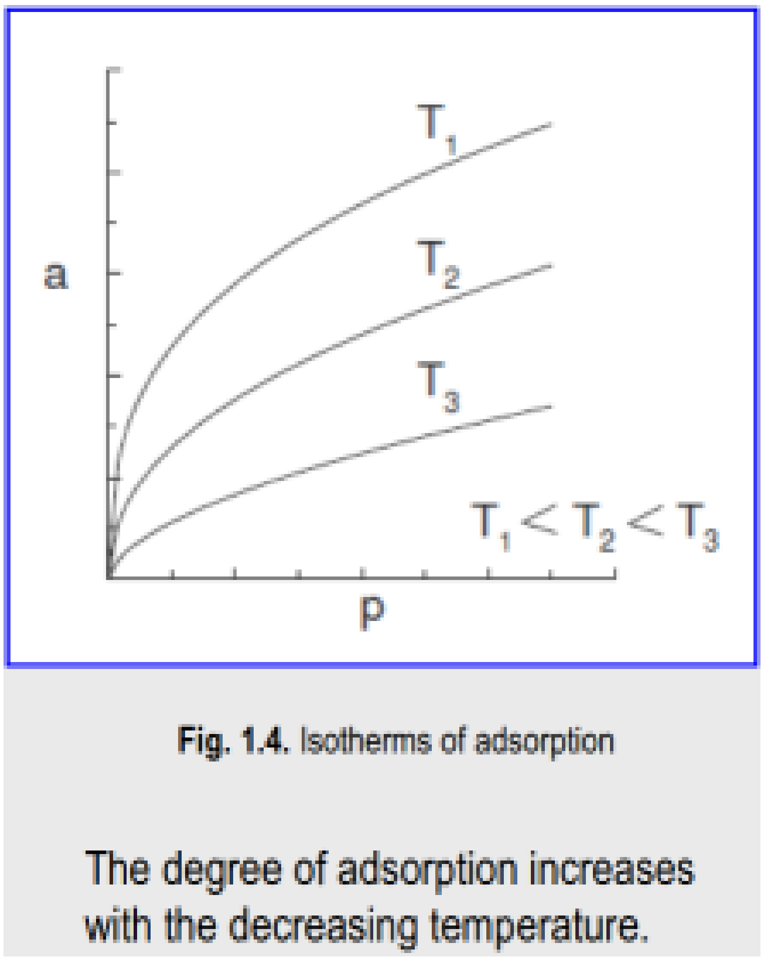


Fig. 1.4. Isotherms of adsorption

The degree of adsorption increases with the decreasing temperature.

Figure 5: Dependance of adsorption isotherms on temperature (Université Marie Curie-Skłodowska 2013)

2.2 Creating a fit for the data from Pommerol et al. (2009)

It has been found that the adsorption of the Martian soil can be characterized by isotherms of type II and III (see fig. 6). Note that isotherms may show hysteresis and/or other phenomena, but these are not the current object of study of this stage and will not be implemented in the model. In Pommerol et al. (2009), the authors provide a linear regression of the data that could potentially be integrated to the subsurface model (sec. 2.3).

As part of the internship however, a two-segment function (instead of the BET function) will be fitted to the data in fig. 6 and eventually integrated in the subsurface model (section sec. 3). Once this simple fit is achieved, including the BET function or better fits may be discussed (see sec. 2.5.1).

Internship work in sec. 3.

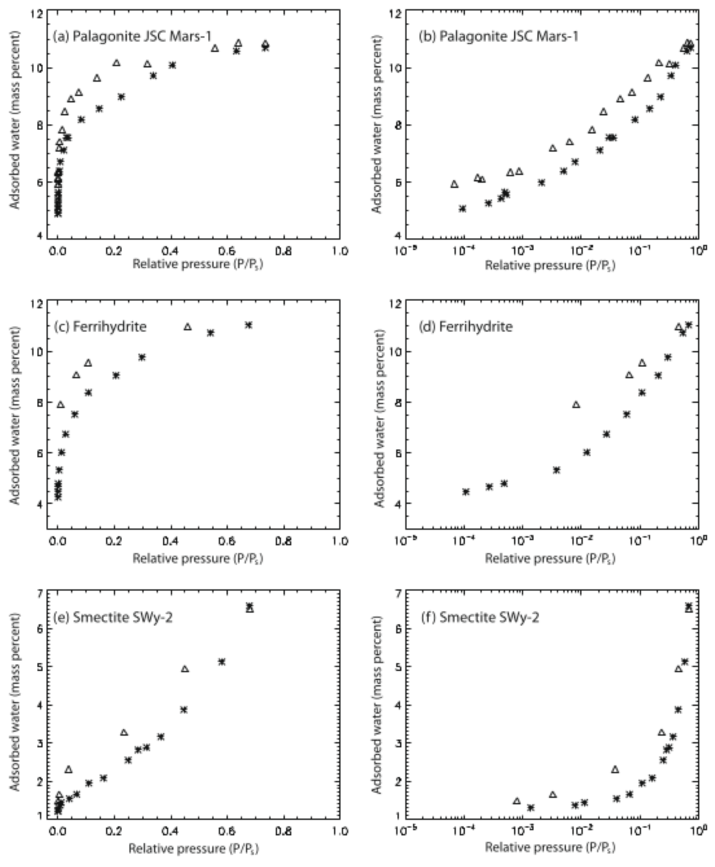


Fig. 7. Adsorption and desorption isotherms (stars and triangles, respectively) at −30 °C for three samples (a to f): JSC Mars-1, ferrihydrite and smectite SWy-2. For each sample, the relative mass of adsorbed water is plotted versus relative water vapor pressure in linear and logarithmic scales (respectively, left and right columns). Relative uncertainty on both water content and relative pressure is estimated to be 10%.

Figure 6: Data from Mars follow the isotherms of type II and III (Pommerol et al. 2009). **Adsorbed water is given as** $\frac{m_{\text{water}}}{m_{\text{dry}}}$

2.3 Including the expression in the subsurface model

Once the expression has been found, the aim is to include this expression in the subsurface model.

Internship work in sec. 5.

2.4 Coupling the subsurface model and the GCM

The aim is to feed data to the LMDZ GCM from the subsurface model.

TODO Internship work in ....

2.5 Future work

Once the goals of the internship have been achieved, the following tasks may be carried out if there is still some time left.

2.5.1 Use a better fit for the data

In order to improve the accuracy, the linear regressions in Pommerol et al. (2009) (see sec. 2.2) could replace the two segment fit (see sec. 2.3). A less ambitious approach is to fit the data with more than two segments.

2.5.2 Parallelize the subsurface model

The current subsurface model is sequential, and may act as a bottleneck in some simulations. After becoming familiar with MPI in Shallow Worlds (appearances in EPSC-DPS and Nature Astronomy), an parallel implementation of the subsurface model can be discussed.

3 A fitting for type I, II, and III (and possibly 4 and 5) isotherms

The constitution of the Martian soil is not known but it is safe-ish to assume that it has the properties of an “average” simulant, which is a mixture of all of the materials studied in Pommerol et al. (2009).

```python
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.colors as mcolors
from matplotlib import cm
from cycler import cycler
from scipy import optimize, stats
from scipy.constants import Avogadro, Boltzmann, R
import numpy as np
import pandas as pd
import random
```

Set matplotlib style

```python
plt.style.use("ggplot")
```

A `rotate()` function that shifts lists:

```python
def rotate(li, x):
    return li[-x % len(li):] + li[:-x % len(li)]
```

A `Material` class for each of the materials will be created:

```python
# %%writefile material.py

# Define new ground material
class Material:

    def simple_material(self, name, short_name, SSA, T = 243, Psmbar = 0.37, dens =
        1300, **kwargs):
        self.name = name  # Long name of the material
        self.short_name = short_name  # Name of the material as it appears in the csv

        self.msample = None  # [kg] Mass sample
        self.SSA = SSA  # Specific Surface Area normalized by sample mass
        material.yp = None # Monolayer saturation value

        self.ads_iso = None  # Adsorption isotherms data frame
        self.des_iso = None  # Desorption isotherms data frame

        self.T = T # [K]
        self.Psmbar = Psmbar # [mbar]
        self.Ps = Psmbar*100  # [Pa]
        self.dens = dens # [kg/m^3]

    def from_Pommerol2009(self, name, short_name, K1, K0, Vm, C, SSA, initial_water,
        final_water, T = 243, Psmbar = 0.37, dens = 1300, **kwargs):
        self.name = name  # Long name of the material
        self.short_name = short_name  # Name of the material as it appears in the csv
        self.K1 = K1  # BET linear regression constant
        self.K0 = K0  # BET linear regression constant
        self.C = C  # BET adsorption constant C = 1 + K1/K0
        self.Vm = Vm  # [m^3/kg] Volume of the water monolayer normalized by sample mass
        self.vm = None  # [m^3] Volume of the water monolayer

        self.msample = None  # [kg] Mass sample
        self.SSA = SSA  # Specific Surface Area normalized by sample mass

        self.ads_iso = None  # Adsorption isotherms data frame
        self.des_iso = None  # Desorption isotherms data frame

        # Water already present in the sample (see Table 2 in Pommerol2009)
        # Water content at the start (see Table 2 in Pommerol2009), i.e. the water that
            could not be removed
        self.initial_water = initial_water
        self.final_water = final_water  # Water content at the end (see Table 2 in
            Pommerol2009)

        self.xp = None # Monolayer saturation abscisse at x
        self.yp = None # Monolayer saturation value

        self.Qads = None # BET isotherm adsorption energy
        self.Q1ads =  None # Adsorption energy (segment 1)
        self.Q2ads =  None # Adsorption energy (segment 2)
        self.kads_ref = None # Reference adsorption constant (segment 1)
        self.kpads_ref = None # Reference adsorption constant (segment 2)

        self.T = T # [K] Temperature at which the data is found
        self.Psmbar = Psmbar # [mbar] Saturation pressure of water at 243 K
        self.Ps = Psmbar*100  # [Pa] Saturation pressure of water at 243 K
        self.dens = dens # [kg/m^3] 3000 kg/m^3 of rock * 0.45 porosity. 870 kg/m^3 for
            jsc1 https://www.lpi.usra.edu/meetings/LPSC98/pdf/1690.pdf

        # R_specific_H2O = 461.523 # [J/(kg K)] Specific gas constant calculated as R/M(
            H2O)
        # dens_H2O = Ps/(R_specific_H2O * T) # [kg/m^3] Density
        # self.dens_H2O = 1e3  # [kg/m^3] The density of the first monolayer must be
            around 1e3 kg/m^3
```

The data is imported from Pommerol et al. (2009) and preprocessed:

```
Tref = 243. # [K] Isotherm data from Pommerol2009 found at 293 K
Psmbar = 0.37 # [mbar] Saturation pressure at 293K
Ps = Psmbar*100 # [Pa] Saturation pressure at 293
dens_rego = 1300. # [kg/m^3]

# Initialize ground materials
jsc1_dict = {
    "name": "JSC Mars-1",
    "short_name": "jsc1",
    "K1": 4.17e7,
    "K0": 4.10e5,
    "Vm": 3.19e-5,
    "C": 103.4,
    "SSA": 1.06e5,
    "initial_water": 5.0,
    "final_water": 10.7,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
jsc1 = Material()
jsc1.from_Pommerol2009(**jsc1_dict)

ferri_dict = {
    "name": "Ferrihydrite",
    "short_name": "ferri",
    "K1": 2.82e7,
    "K0": 5.20e5,
    "Vm": 4.02e-5,
    "C": 56.5,
    "SSA": 1.34e5,
    "initial_water": 4.3,
    "final_water": 11.0,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
ferri = Material()
ferri.from_Pommerol2009(**ferri_dict)

swy2_dict = {
    "name": "Smectite SWy-2",
    "short_name": "swy2",
    "K1": 1.09e8,
    "K0": 2.36e7,
    "Vm": 1.58e-5,
    "C": 5.7,
    "SSA": 5.27e4,
    "initial_water": 1.2,
    "final_water": 6.6,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
swy2 = Material()
swy2.from_Pommerol2009(**swy2_dict)

dun_dict = {
    "name": "Dunite",
    "short_name": "dun",
    "K1": 1.58e9,
    "K0": 1.32e7,
    "Vm": 8.48e-7,
    "C": 120.8,
    "SSA": 2.83e3,
    "initial_water": 0.2,
    "final_water": 0.4,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
dun = Material()
dun.from_Pommerol2009(**dun_dict)

t1_dict = {
    "name": "Volcanic tuff",
    "short_name": "t1",
    "K1": 3.52e8,
    "K0": 2.64e6,
    "Vm": 4.10e-6,
    "C": 149.4,
    "SSA": 1.37e4,
    "initial_water": 0.3,
    "final_water": 1.0,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
t1 = Material()
t1.from_Pommerol2009(**t1_dict)
```

```
sulf0_dict = {
    "name": "Tuff + Mg-sulf.",
    "short_name": "sulf0",
    "K1": 3.87e8,
    "K0": 2.41e6,
    "Vm": 3.38e-6,
    "C": 196.0,
    "SSA": 1.13e4,
    "initial_water": 0.4,
    "final_water": 1.0,
    "T": Tref,
    "Psmbar": Psmbar,
    "dens": dens_rego
}
sulf0 = Material()
sulf0.from_Pommerol2009(**sulf0_dict)
```

```
# Import data
df = pd.read_csv("./data/Pommerol2009/pr_jsc1_ads_2.csv")
```

```
# Set up array
materials = [jsc1, ferri, swy2, dun, t1, sulf0]

for material in materials:

    # Store data
    material.ads_iso = df[["pr_"+material.short_name+"_ads", "wt_"+material.short_name+"
        _ads"]]
    material.ads_iso = material.ads_iso.rename(
        columns={material.ads_iso.columns[0]: "pr", material.ads_iso.columns[1]: "
            wt_percent"})
    material.msample = (1./(material.K0 + material.K1))/material.Vm

    # pr_ps, relative pressure P/Ps, pr over psaturation also (relative humidity)/100
    material.ads_iso["x"] = material.ads_iso["pr"]/material.Psmbar

    # Adsorbed water divided by msample and multiplied by 100
    material.ads_iso["wt_ads_percent"] = material.ads_iso["wt_percent"] + \
        material.initial_water  # In percentage

    # Normalized adsorbed water
    material.ads_iso["wt_ads"] = material.ads_iso["wt_ads_percent"] / 100  # Normalized

    # Adsorbed water divided by msample and multiplied by 100 with no offset (zero(0)-
        offset)
    material.ads_iso["wt_ads_percent_0o"] = material.ads_iso["wt_percent"] # In
        percentage

    # Normalized adsorbed water with no offset (zero(0)-offset)
    material.ads_iso["wt_ads_0o"] = material.ads_iso["wt_ads_percent_0o"] / 100  #
        Normalized

    # Compute other magnitudes
    material.vm = material.Vm * material.msample
    material.c = 1 / (material.vm * material.K0) # BET constant
```

Normalize the $\frac{m}{m_{\text{sample}}}$ by SSA[6] so as to obtain $\frac{1}{SSA} \cdot \frac{m}{m_{\text{sample}}}$. P.-Y.Meslin suggested using the adsorbed mass witout the offset `wt_ads_0o` as the curves appear closer together and are easier to fit with a curve. Therefore, **`wt_ads_0o` will be used instead of `wt_ads`**.

Import the already-initialized materials:

[6]The SSA in Pommerol et al. (2009) is given in m^2/kg.

```
for material in materials:

    # Normalize by SSA
    material.ads_iso["wt_ads_SSA"] = material.ads_iso["wt_ads_0o"] / material.SSA
```

The plot of this new magnitude is:

```
ax = plt.gca(ylim=[0, 1.5e-6], title="Isotherms normalized by SSA")

for material in materials:

    # Plot material isotherm
    ax.plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], "o-", label =
        material.name)

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
# plt.savefig('./figures/wt_ads_SSA_all_isotherms.png', dpi=1000)
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

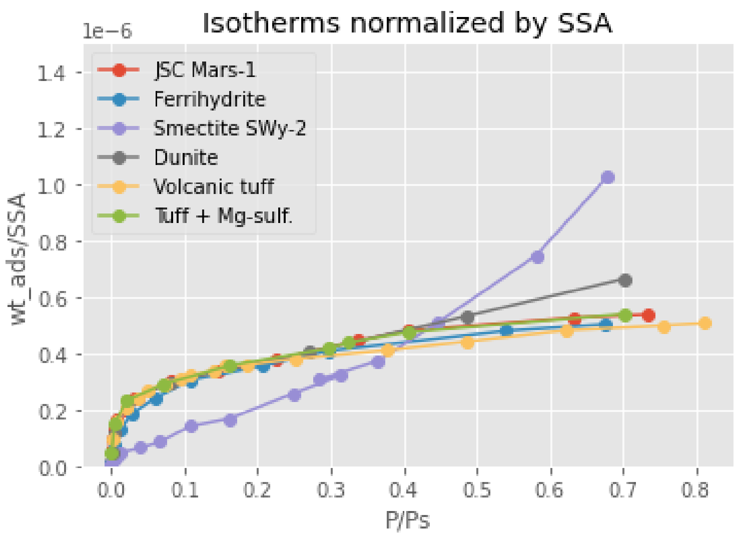


Figure 7: png

Note that the Smectite SWy-2 follows a different pattern. This material will not be included in the `materials` list that the fitting will use. Therefore, it is convenient to redefine `materials`.

```
materials = [jsc1, ferri, dun, t1, sulf0]
```

```
ax = plt.gca(ylim=[0, 1.5e-6], title="Isotherms normalized by SSA")

for material in materials:

    # Plot material isotherm
    ax.plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], "o-", label =
        material.name)

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
# plt.savefig('./figures/wt_ads_SSA_good_isotherms.png', dpi=1000)
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

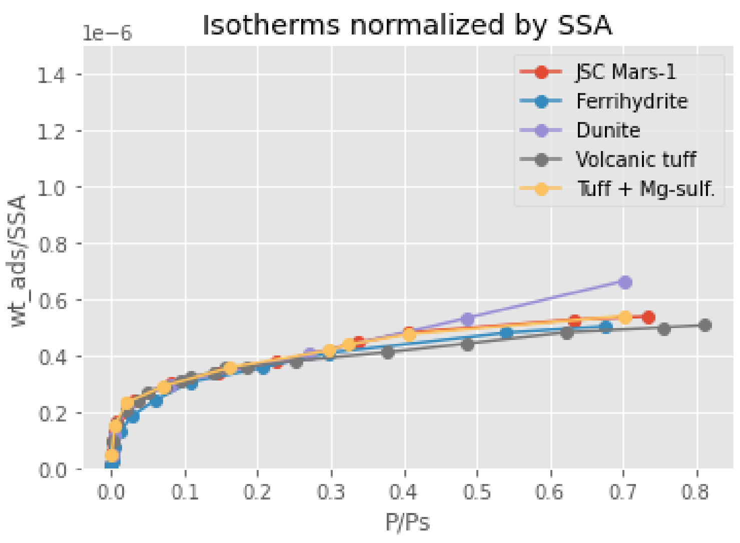


Figure 8: png

3.1 Fitting a BET isotherm to the data

The goal of this task is to parametrize the data shown in fig. 6. It is clear that the data follows the shape of isotherms of type I, II, and III, so the best approach is to fit an equation that ressembles eq. 2. However, the final form of the equation should include the adsorbed water in percentage of mass and the relative pressure.

In Pommerol et al. (2009), the authors present the BET parameters estimated by fitting the adsorption isotherms. They present the coefficients of the linear regression in the BET space $y = K0 + K1 \cdot x$, where:

$$\underbrace{\frac{1}{v}\frac{\frac{p}{p_0}}{\left(1-\frac{p}{p_0}\right)}}_{y} = \underbrace{\frac{1}{v_m c}}_{K0} + \underbrace{\frac{c-1}{v_m c}}_{K1}\underbrace{\frac{p}{p_0}}_{x}$$

With $m_{\text{sample}} = \underbrace{\frac{1}{K0+K1}}_{v_m} \cdot \frac{1}{V_m}$, wth V_m being the volume of the water monolayer normalized by sample mass (as given in Pommerol et al. (2009)), and $m = \rho_{\mathrm{H_2O}} v$ being the adsorbed mass with $\rho_{\mathrm{H_2O}} = \frac{p_0}{\frac{R}{M}T(p_0)}$, the following development can be made:

$$\begin{aligned} v &= \frac{v_m c x}{(1-x)\left[1+(c-1)x\right]} \\ v &= \frac{V_m m_{\text{sample}} c x}{(1-x)\left[1+(c-1)x\right]} \\ v\frac{\rho_{\mathrm{H_2O}}}{\rho_{\mathrm{H_2O}}} &= \frac{V_m m_{\text{sample}} c x}{(1-x)\left[1+(c-1)x\right]} \\ \frac{m}{m_{\text{sample}}} &= \frac{V_m \rho_{\mathrm{H_2O}} c x}{(1-x)\left[1+(c-1)x\right]} \end{aligned}$$

```
dens_H2O_ice = 1e3 # 920. # [kg/m^3] Density not changing with temperature
dens_H2O = dens_H2O_ice # 997 # [kg/m3] # IMPORTANT: these are equal because they have
    indistinctly been used
```

Note that $c = \frac{1}{v_m K0} = \frac{1}{V_m m_{\text{sample}} K0}$.

3.1.1 Finding a BET fit

BET is only valid in the following range:

```
BETlim = [0.05, 0.35]
```

The BET isotherm is described by:

```
def BET_isotherm(x, a, c):
    """Expression of a BET isotherm. See BET1938 or BET1940."""
    return a * c * x / ((1-x) * (1 + (c-1) * x))
```

Select the range for which the BET theory is valid:

```
for material in materials:

    # Select data to be used for BET fitting. BET is only valid when P/Ps is in BETlim =
        [0.05, 0.35]
    material.BET = material.ads_iso[(material.ads_iso["x"] >= BETlim[0])
                                    & (material.ads_iso["x"] <= BETlim[1])]
```

The density should also be found, along with other parameters:

```
for material in materials:

    material.ads_iso["v"] = material.BET["x"] / ((material.K0 + material.K1*material.BET
        ["x"]) * (1-material.BET["x"]))
    material.ads_iso["m"] = material.BET["wt_ads"] * material.msample
    material.ads_iso["dens_H2O"] = material.ads_iso["m"]/material.ads_iso["v"]
```

The following procedure compares a linear regression with the ones presented in Pommerol et al. (2009):

```
axbetfit = plt.gca(title="BET fitting") # ylim=[0, 1.5e-6]

colors = ['r','b','y','m','c','k']

for material in materials:

    color = colors[0]
    colors = rotate(colors, -1) # shift colors

    xbet = material.BET["x"]

    v = xbet / ((material.K0 + material.K1*xbet) * (1-xbet)) # From above

    # Plot material isotherm
    ybet = material.K0 + material.K1*xbet
    axbetfit.plot(xbet, ybet, ':x', color=color, label=material.name + " (regression in
        Pommerol)")

    # Linearize data
    ybet = 1 / (1 / xbet - 1) * 1 / ( material.BET["wt_ads_SSA"] * material.SSA *
        material.msample / dens_H2O )
    axbetfit.plot(xbet, ybet, 'o', color=color, label=material.name + " (linearized data
        )")
    K1reg, K0reg, *paramsreg  = stats.linregress(xbet, ybet)
    axbetfit.plot(xbet, K0reg + K1reg*xbet, '--', color=color, label=material.name + " (
        regression)")

    print(material.name + ":")
    print("K0 Pommerol: " + str(material.K0) + ", K0 regression: " + str(K0reg))
    print("K1 Pommerol: " + str(material.K1) + ", K1 regression: " + str(K1reg))
    print("C Pommerol (data): " + str(material.C) + ", C Pommerol (from regression): " +
         str(1 + material.K1/material.K0) + ", C regression: " + str(1 + K1reg/K0reg))
    print()

plt.legend(loc="best", bbox_to_anchor=(1, 1))
plt.xlabel("P/Ps")
plt.ylabel("1/v*1/(Ps/P-1) = 1/v*1/(1/x-1)")
```

```
JSC Mars-1:
K0 Pommerol: 410000.0, K0 regression: 316621.883805451
K1 Pommerol: 41700000.0, K1 regression: 42097041.31996083
C Pommerol (data): 103.4, C Pommerol (from regression): 102.70731707317073, C regression
    : 133.9568279172625

Ferrihydrite:
K0 Pommerol: 520000.0, K0 regression: 487895.88462594524
K1 Pommerol: 28200000.0, K1 regression: 28310624.91625444
C Pommerol (data): 56.5, C Pommerol (from regression): 55.23076923076923, C regression:
    59.02595555393816

Dunite:
K0 Pommerol: 13200000.0, K0 regression: 13613579.975638628
K1 Pommerol: 1580000000.0, K1 regression: 1574235026.784142
C Pommerol (data): 120.8, C Pommerol (from regression): 120.6969696969697, C regression:
     116.63710865189177

Volcanic tuff:
K0 Pommerol: 2640000.0, K0 regression: 812774.5910276473
K1 Pommerol: 352000000.0, K1 regression: 363192942.35825235
C Pommerol (data): 149.4, C Pommerol (from regression): 134.33333333333334, C regression
    : 447.85567975131005

Tuff + Mg-sulf.:
K0 Pommerol: 2410000.0, K0 regression: 2161939.096821904
K1 Pommerol: 387000000.0, K1 regression: 387782813.16282624
C Pommerol (data): 196.0, C Pommerol (from regression): 161.58091286307055, C regression
    : 180.3680560811704






Text(0, 0.5, '1/v*1/(Ps/P-1) = 1/v*1/(1/x-1)')
```

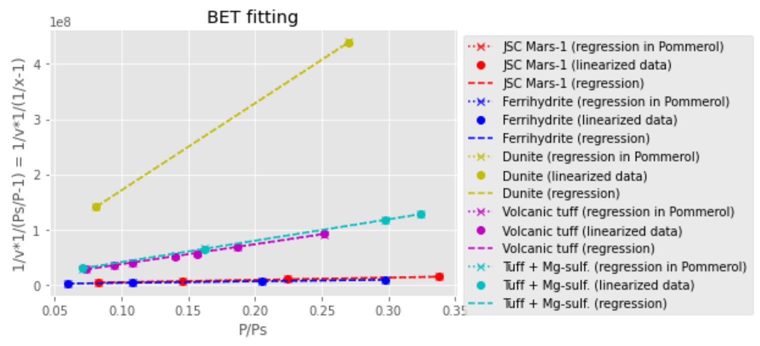


Figure 9: png

The new columns in `material` were added in the pressure ranges where the BET theory is valid. For `jsc1`:

```python
print(jsc1.ads_iso[["x", "v", "m", "dens_H2O"]])
```

```
           x             v         m     dens_H2O
0   0.000000           NaN       NaN          NaN
1   0.000000           NaN       NaN          NaN
2   0.000000           NaN       NaN          NaN
3   0.000095           NaN       NaN          NaN
4   0.000262           NaN       NaN          NaN
5   0.000430           NaN       NaN          NaN
6   0.000530           NaN       NaN          NaN
7   0.000497           NaN       NaN          NaN
8   0.002108           NaN       NaN          NaN
9   0.005000           NaN       NaN          NaN
10  0.007838           NaN       NaN          NaN
11  0.020811           NaN       NaN          NaN
12  0.035135           NaN       NaN          NaN
13  0.029730           NaN       NaN          NaN
14  0.082703  2.336513e-08  0.000061  2605.019813
15  0.145946  2.630657e-08  0.000064  2422.900221
16  0.224324  2.961788e-08  0.000067  2257.808610
17  0.337838  3.519174e-08  0.000072  2054.363531
18  0.405405           NaN       NaN          NaN
19  0.632432           NaN       NaN          NaN
20  0.735135           NaN       NaN          NaN
```

Herebelow is an attempt to plot the linear regression on the pair of axes `wt_ads_SSA`–`x` using the `K0` and `K1` constants in Pommerol et al. (2009), and alongside the data from Pommerol et al. (2009):

```python
axbetp = plt.gca(title="Isotherms normalized by SSA with the linear regression")

colors = ['r','b','y','m','c','k'] # color list

for material in materials:

    color = colors[0]
    colors = rotate(colors,-1) # shift colors

    xbetp = material.BET["x"]
    ybetp = material.BET["wt_ads_SSA"]
    v = material.ads_iso["v"].dropna().values # Remove nans
    m = material.ads_iso["m"].dropna().values # Remove nans
    ybetpreg =  v * (material.K0 + material.K1 * xbetp) * (1/xbetp - 1) * m/material.
        msample * 1 / material.SSA

    # 1 / (1 / material.BET["x"] - 1) * 1 / ( material.BET["wt_ads_SSA"] * material.SSA
        * material.msample / dens_H2O )
    ybetpreg2 = 1 / (material.K0 + material.K1 * xbetp) * 1 / (1/xbetp - 1) * dens_H2O /
        (material.msample * material.SSA)

    # Plot material isotherm
    axbetp.set_yscale('log')
    axbetp.plot(xbetp, ybetp, "x", color=color, label = material.name)
    axbetp.plot(xbetp, ybetpreg, ":", color=color, label = material.name + " (old
        regression)")
    axbetp.plot(xbetp, ybetpreg2, "--", color=color, label = material.name + " (
        regression)")

plt.legend(loc="best", bbox_to_anchor=(1, 1))
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
# plt.savefig('./figures/wt_ads_SSA_good_isotherms.png', dpi=1000)
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

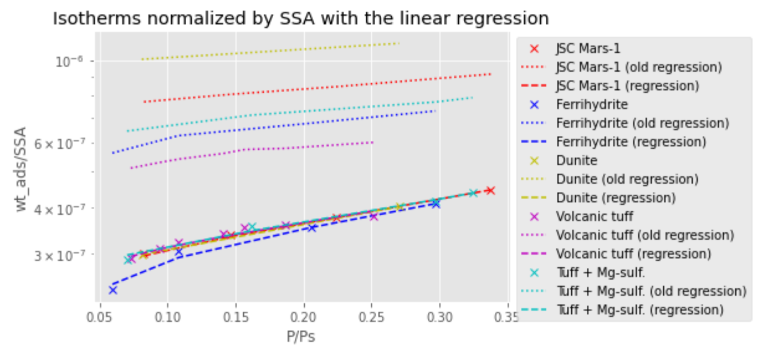


Figure 10: png

Although not that far off, the old regression lines do not provide an accurate description of the isotherms.

3.1.2 Problems determining the density of the adsorbed water

~~In Pommerol et al. (2009), it is shown that the data follows a BET isotherm when P/Ps is in BETlim = [0.05, 0.35], and provides the K0 and K1 coefficients of the linear regression. The initial goal was to reshape the regression line (which is a function of v and P/Ps) and use the density of the adsorbed water so as to add the regression line in the plots of Fig. 7 and Fig. 8, where they show the mads/msample (%) as a function of P/Ps.~~

~~Using the ideal gas law, I found an incorrect value of density. As told by, P.-Y. Meslin, I tried to use the value 1 g/cm^3, but the regression line did—in any way—follow the data in the plots of Fig. 7 and Fig. 8.~~

~~Therefore, the final idea, was to fit a BET isotherm to the data, and by determining the coefficients, I might have been able to find the density, as in the expressions of these coefficients, the density is present.~~

~~However, I could not find any reasonable value (as the nonlinear fitting depends a lot on the initial values chosen). I moved on to more productive things, as it turned out that the GCM does not need, in any way the volume (or the expression of the BET isotherm). Therefore, the data provided (mads/msample (%) as a function of P/Ps) is enough for the GCM.~~

Finally, the problem arised from the available significant figures. The operation `material.ads_iso["wt_ads_SSA"]*material.SSA*material.msample` should be equivalent to `material.ads_iso["wt_ads"]*` `material.msample`, but the small differences were enougth to obtain a significantly different regression line.

```
print(material.ads_iso["wt_ads_SSA"]*material.SSA*material.msample)
print(material.ads_iso["wt_ads"]*material.msample)
```

```
0     4.059510e-07
1     1.286410e-06
2     1.984796e-06
3     2.481762e-06
4     3.059339e-06
5     3.575459e-06
6     3.756229e-06
7     4.070397e-06
8     4.627521e-06
9              NaN
10             NaN
11             NaN
12             NaN
13             NaN
14             NaN
15             NaN
16             NaN
17             NaN
18             NaN
19             NaN
20             NaN
Name: wt_ads_SSA, dtype: float64
0     0.000003
1     0.000004
2     0.000005
3     0.000006
4     0.000006
5     0.000007
6     0.000007
7     0.000007
8     0.000008
9          NaN
10         NaN
11         NaN
12         NaN
13         NaN
14         NaN
15         NaN
16         NaN
17         NaN
18         NaN
19         NaN
20         NaN
Name: wt_ads, dtype: float64
```

3.2 A simple fit to be used as GCM input

The next goal of the project is to parametrize the *under implementation* fitting in fig. 2. The current description of the data presented in fig. 11 and the input given to the GCM is basically a sloped line followed by a horizontal line (see *present model* in fig. 2). While for Langmuir-type isotherms, this may work sufficiently good, it is not good enough for all kinds of simulats (Earth materials with similar properties to the Martian soil).

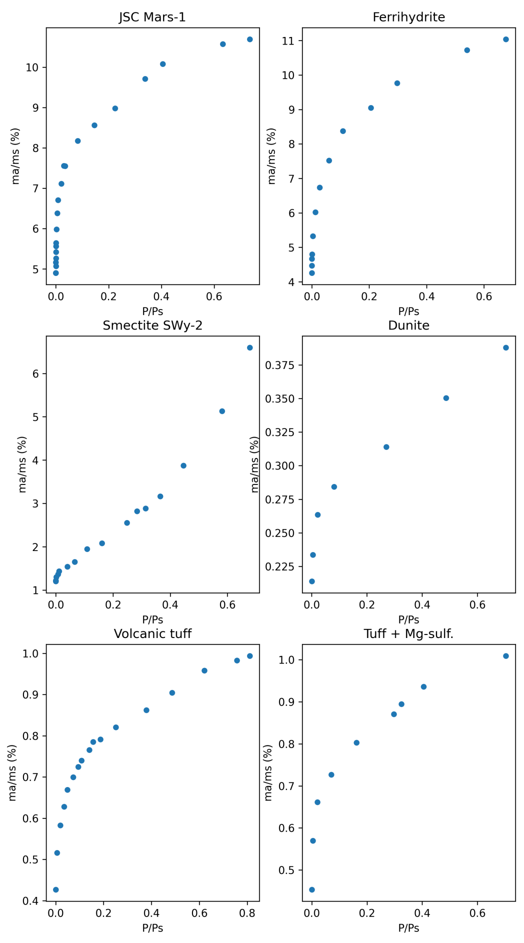


Figure 11: Plots of the isothermss that appear in Fig 7. and Fig. 8 in Pommerol et al. (2009). Some of these plots also appear in fig. 6

As seen in fig. 13, two straight lines will be used to fit the data. Given that the first line (coloured in red in fig. 13) starts at $x, y = 0$, the main unknowns are the coordinates x_p, y_p of the point where the straight lines join, and the slope of the second line (as seen in green in fig. 13). Note that $y = \frac{1}{SSA} \frac{m}{m_{\text{sample}}}$.

P.-Y. Meslin suggested setting the point where y_p is representative of the monolayer being filled/saturated. In other words, the lines should join when $n_{\text{ads}} = N \iff \theta = \frac{n_{\text{ads}}}{N} = 1$. This is in fact what he uses in fig. 2 and he has written this as $n_{\text{ads}} = \min\left(k_{\text{ads}}\left(T\right) n_{\text{vap}}, n_{\text{ads}}\left(\theta = 1\right)\right)$, with $\frac{n_{\text{ads}}}{n_{\text{vap}}} = \frac{K_a(T)}{K_d(T)} = k_{\text{ads}}\left(T\right) = k^0_{\text{ads}}\left(S, T\right) e^{\frac{Q_{\text{ads}}}{RT}}$.

As a remainder, time/kinetic constants $K_a\left(T\right)$ and $K_d\left(T\right)$ appear in the transport equation (see eq. 1):

$$\begin{aligned}
\frac{\partial\left(\varepsilon\left(S_w\right) n_{\text{vap}}\right)}{\partial t} &= \frac{\partial}{\partial z}\left(D_b\left(p, T, S_w\right) \frac{\partial n_{\text{vap}}}{\partial z}\right) - K_a\left(T\right)\left(1-\theta\right) n_{\text{vap}} + K_d\left(T\right) n_{\text{ads}} \\
\frac{\partial\left(n_{\text{ads}}\right)}{\partial z} &= K_a\left(T\right)\left(1-\theta\right) n_{\text{vap}} - K_d\left(T\right) n_{\text{ads}} \\
\frac{\partial\left(\varepsilon\left(S_w\right) n_{\text{vap}}\right)}{\partial t} &= \frac{\partial}{\partial z}\left(D_b\left(p, T, S_w\right) \frac{\partial n_{\text{vap}}}{\partial z}\right) - K_a\left(T\right)\left(1-\theta\right) n_{\text{vap}} + K_d\left(T\right) n_{\text{ads}} \\
\frac{\partial\left(n_{\text{ads}}\right)}{\partial z} &= K_a\left(T\right)\left(1-\theta\right) n_{\text{vap}} - K_d\left(T\right) n_{\text{ads}}
\end{aligned}$$

The work will be carried out as follows (**this is a starting point, and by no means it is guaranteed that the order nor the method will be followed exactly**):

1. Divide $\frac{m}{m_{\text{sample}}}$ by the SSA, so $\frac{1}{SSA}\frac{m}{m_{\text{sample}}}$ is obtained.
2. Superpose the plots of $\frac{1}{SSA}\frac{m}{m_{\text{sample}}}$ vs. $x = \frac{p}{p_0}$. In theory, the plots for different materials should follow the same curve more or less (see blue lines in fig. 13). Discard those curves that are radically different (i.e. Smectite SWy-2).
3. Find two lines that minimize the *error* by means of the least squares fitting method, similarly to fig. 12, but taking into account the whole ensemble of curves (see fig. 13). In other words, the goal is to minimize the sum of errors of the fitting for each material. The constitution of the Martian soil is not known but it is safe-ish to assume that it has the properties of an "average" simulant, which is a mixture of all of the materials studied in Pommerol et al. (2009). The final fitting has to take into account the effects of all simulants, and by minimizing the sum of the errors from the least square fitting method, this is achieved.

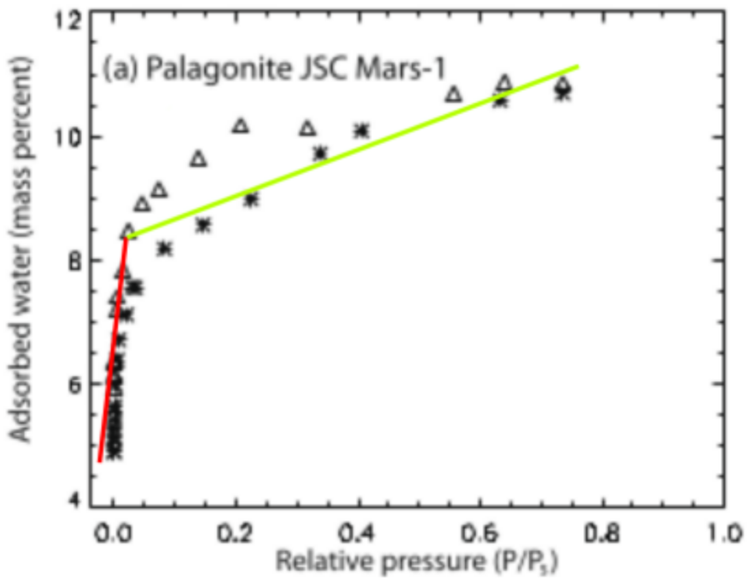


Figure 12: Plot of the isotherm for Palagonite JSC Mars-1 with the simple fit superimposed

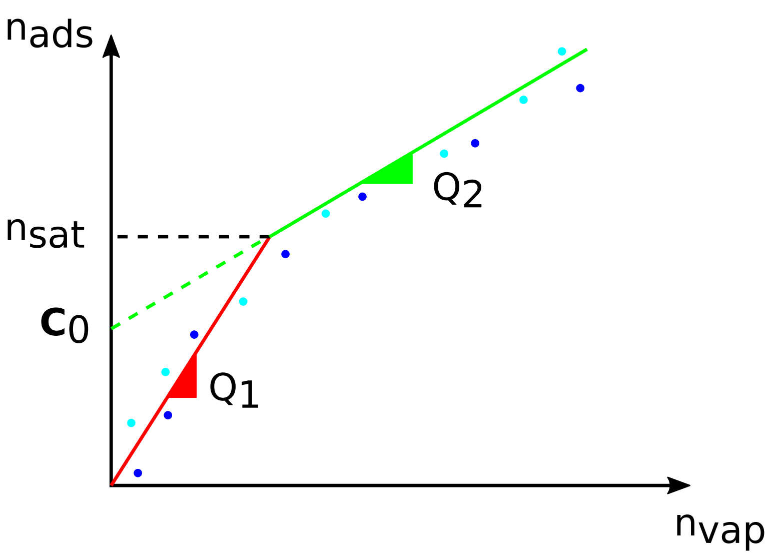


Figure 13: Plots of isotherms with the simple fit superimposed

3.2.1 Finding a simple fit to isotherms

3.2.1.1 A fit for a single isotherm (or material)

The following selects the material to be used:

```
material = ferri
```

3.2.1.1.1 Understanding the variables that come into play

In order to understand the factors that come into play, the two lines will be fit to the isotherm of the simulant using the least squares fitting method.

```
figc, axc = plt.subplots()
material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", style="-x", ax=axc)
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

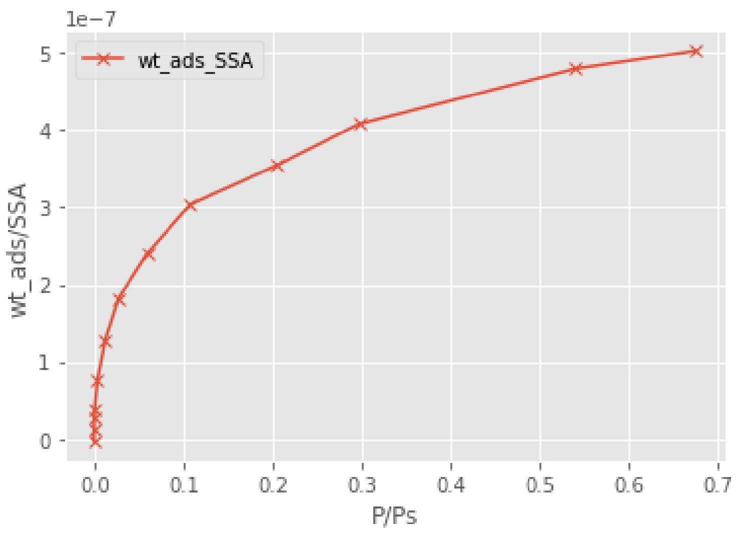


Figure 14: png

The two lines that will fit the data need to be plotted: - the first line starts at $x, y = 0, 0$ and ends at $x, y = x_p, y_p$. The slope of this line is—therefore—$m_1 = \frac{y_p}{x_p}$ - the second line starts at $x, y = x_p, y_p$ and has a slope m_2

The goal is to minimize the error, so the method of the least squares will be used, i.e. S will be minimized (r are the residuals):

$$S = \sum r_i^2$$

The ordinates of the lines at each x is wanted, and these will be subracted from the ordinates of the isotherm:

```
x = material.ads_iso["x"].values # x values
x = x[~np.isnan(x)] # x values without nans
y = material.ads_iso["wt_ads_SSA"].values # y values
y = y[~np.isnan(y)] # y values without nans

xp, yp = 1e-1, 3e-7 # Joining point

x1 = x[(x<=xp)] # Select x values lower than or equal to xp
# x1 = np.append(x1, xp) # Append xp to x1 [or prepend to x2 (see below)]
m1 = yp/xp # Slope of the first line
y1 = m1*x1 # The first line goes through x,y = 0,0

x2end, y2end = x[-1], y[-1]

x2 = x[(x>xp)] # Select x values larger than xp
# x2 = np.insert(x2, 0, xp) # Prepend xp to x2
m2 = (y2end-yp)/(x2end-xp) # Slope of the second line
y2 = m2*(x2-xp) + yp # Expression of the second line

axc.plot(x1, y1, "-x") # Plot first line
axc.plot(x2, y2, "-x") # Plot second line

figc # Display plot
```

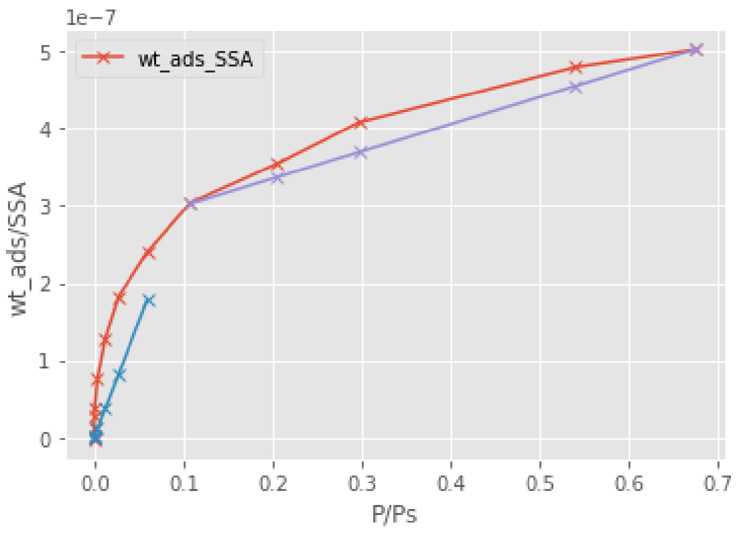


Figure 15: png

Note that **lines do not join because they are evaluated at discrete points**. If both `x1 = np.append(x1, xp)` and `x2 = np.insert(x2, 0, xp)` had been used, they would join, but the array [x1, x2] would have repeated numbers. Also note that **the point** $x, y = x_p, y_p$ **is not generally part of the isotherm**.

```
# xl = np.append(x1, x2)
# print(xl == x) # True always
```

```
yl = np.append(y1, y2)
```

The residuals and the sum of their squares are computed as follows:

```
r = (yl - y)
S = np.sum(r**2)
print(S)
```

```
3.1016774314815187e-14
```

3.2.1.1.2 Fitting the data with SciPy

The function `scipy.optimize.curve_fit` will be used to perform the fitting as it uses the least squares method. It requires that the lines are expressed in terms of a Python function, with the first argument being the independent variable. It was seen that the variables that will change the shape of the fit are x_p, y_p, and m_2.

Note that the function is also found in the MINPACK library for Fortran.

Testing `scipy.optimize.curve_fit`

The piecewise function (the two curves) may be created as shown below:

```
p0_piecewise = [xp, yp, m2]

def piecewise_linear(x, xp, yp, m2):
    return np.piecewise(x, [x < xp], [lambda x: yp/xp * x, lambda x: m2 * (x-xp) + yp])
        # Define piecewise function
```

Get the ordinates of this function `y_test` at abscissae `x_test`. Plot these values and fit them with `piecewise_linear`. Theoretically, the fitting should pass exactly through the coordinates $x, y = x_{\text{test}}, y_{\text{test}}$.

```
fig, ax = plt.subplots()

x_test = x
y_test = piecewise_linear(x_test, xp, yp, m2);

plt.plot(x_test, y_test, "o", label="Data") # Plot piecewise function

p, e = optimize.curve_fit(piecewise_linear, x_test, y_test, p0 = p0_piecewise)
plt.plot(x_test, piecewise_linear(x_test, *p), "-x", label="Fit") # Plot fit

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

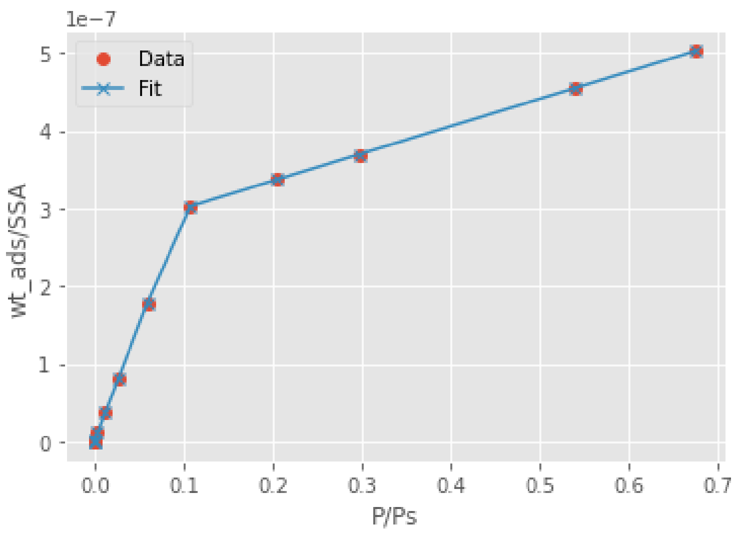


Figure 16: png

Finding a fit to real data

The goal now is to repeat the same procedure but using real data.

```
figr, axr = plt.subplots()
material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", style="-x", ax=axr)

Npoints = 100 # Fit eval points

pr, er = optimize.curve_fit(piecewise_linear, x, y, p0 = p0_piecewise)
xd = np.linspace(x[0],x[-1],Npoints) # Increase resolution
axr.plot(xd, piecewise_linear(xd, *pr), label="Fit") # Plot fit

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")

print("Fit: ", pr, er)
```

```
Fit:  [3.55624247e-02 2.69977060e-07 3.99456925e-07] [[ 3.33735831e-05  6.90227242e-11
   -1.21335926e-10]
 [ 6.90227242e-11  4.50366925e-16 -9.21152038e-16]
 [-1.21335926e-10 -9.21152038e-16  3.12955206e-15]]
```

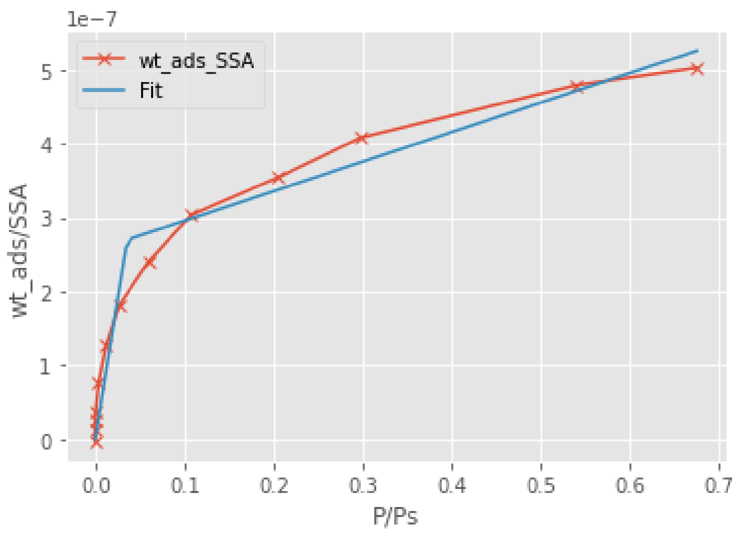


Figure 17: png

Values can also be fixed at desired values:

```
p0_mod_piecewise = [xp, m2] # Variable to be changed p0 for mod_piecewise_linear
yp = 3e-7

def mod_piecewise_linear(x, xp, m2): # Adjust the number of variables
    return piecewise_linear(x, xp, yp, m2)
```

```
figm, axm = plt.subplots()
material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", style="-x", ax=axm)

pm, em = optimize.curve_fit(mod_piecewise_linear, x, y, p0 = p0_mod_piecewise)
axm.plot(xd, mod_piecewise_linear(xd, *pm), label="Fit") # Plot fit

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")

print("Constrained fit: ", pm, em)
```

```
Constrained fit:  [4.02444988e-02 3.37904890e-07] [[3.31461375e-05 2.44879680e-11]
 [2.44879680e-11 1.38991853e-15]]
```

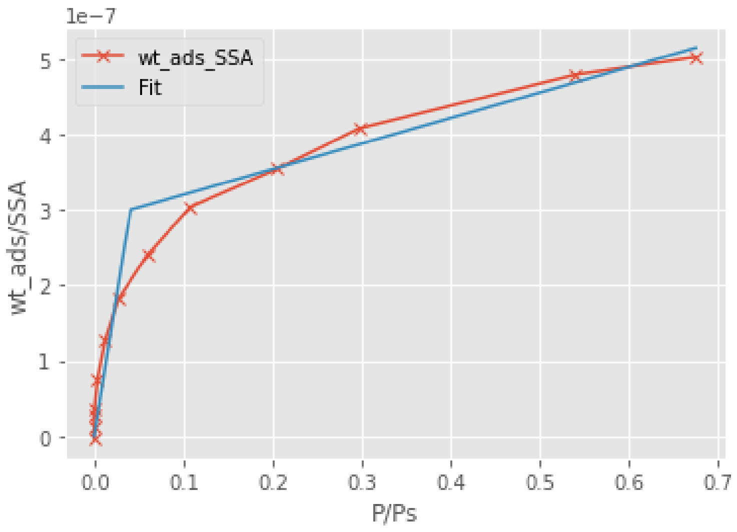


Figure 18: png

Note that **fit constraints can also be added** using `scipy.optimize.leastsq` as shown in https://stackoverflow.com/questi
do-i-put-a-constraint-on-scipy-curve-fit.

3.2.1.2 A fit for multiple/all isotherms (or materials)

The set of materials is defined:

```
materials = [jsc1, ferri, dun, t1, sulf0]
```

The endpoints/limits of `x` are found:

```
x = []

for material in materials:

    xa = material.ads_iso["x"].values # x values
    xa = xa[~np.isnan(xa)] # x values without nans
    x = np.append(x,xa)

x = np.sort(x) # Sort by value
# xd = np.linspace(x[0],x[-1],Npoints) # Function defined to overall max value of x
xd = np.linspace(x[0],min([max(material.ads_iso["x"].values) for material in materials])
    ,Npoints)# Alternative definition: the function only makes sense where data is
    availale, so get the minimum of each of the materials maximum x
```

A loop is created, and the coordinates of each isotherm are stored in `x` and `y`:

```
figa, axa = plt.subplots()

x = []
y = []

for material in materials:

    xa = material.ads_iso["x"].values # x values
    xa = xa[~np.isnan(xa)] # x values without nans
    x = np.append(x,xa)

    ya = material.ads_iso["wt_ads_SSA"].values # y values
    ya = ya[~np.isnan(ya)] # y values without nans
    y = np.append(y,ya)
    material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", label=material.name, style
        ="-.x", ax=axa)

pa, ea = optimize.curve_fit(piecewise_linear, x, y, p0 = p0_piecewise)
axa.plot(xd, piecewise_linear(xd, *pa), label = "Fit", lw=3, c="k", ls="solid") # Plot
    fit
print("Fit: ", pa, ea)

pam, eam = optimize.curve_fit(mod_piecewise_linear, x, y, p0 = p0_mod_piecewise)
axa.plot(xd, mod_piecewise_linear(xd, *pam), label = "Constrained fit", lw=3, c="lime",
    ls="dashed") # Plot constrained fit
print("Constrained fit: ", pam, eam)

axa.legend(loc="best")
axa.set(xlabel="P/Ps", ylabel="wt_ads/SSA")
```

```
Fit:  [2.31834754e-02 2.74766206e-07 4.05638856e-07] [[ 4.87169983e-06  1.20853181e-11
    -2.06622762e-11]
 [ 1.20853181e-11  1.24714247e-16 -2.44885459e-16]
 [-2.06622762e-11 -2.44885459e-16  8.39695968e-16]]
Constrained fit:  [3.04604163e-02 3.59847940e-07] [[5.44348673e-06 4.01970672e-12]
 [4.01970672e-12 3.91642905e-16]]





[Text(0, 0.5, 'wt_ads/SSA'), Text(0.5, 0, 'P/Ps')]
```

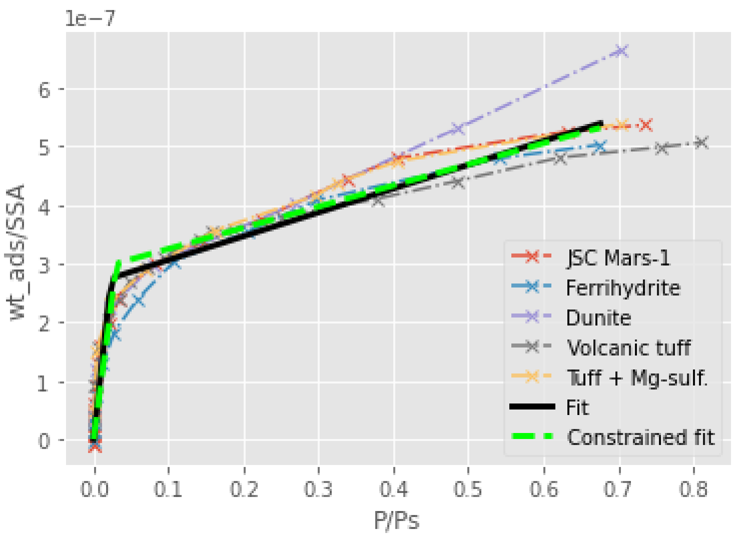


Figure 19: png

The semi-logarithmic plot of the above is shown below:

```
logplot = True # False

# Plot logplot

if logplot:
    figalog, axalog = plt.subplots()
    axalog.set_xscale('log')
    # axa.set_yscale('log') # The first piece of the piecewise function is of type y = m
        *x and appears as a straight line in a loglog plot

    for material in materials:
        material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", label=material.name,
            style="-.x", ax=axalog)

    axalog.plot(xd, piecewise_linear(xd, *pa), label = "Fit", lw=3, c="k", ls="solid") #
         Plot fit
    axalog.plot(xd, mod_piecewise_linear(xd, *pam), label = "Constrained fit", lw=3, c="
        lime", ls="dashed") # Plot constrained fit
    axalog.legend(loc="best")
    axalog.set(xlabel="P/Ps", ylabel="wt_ads/SSA")
```

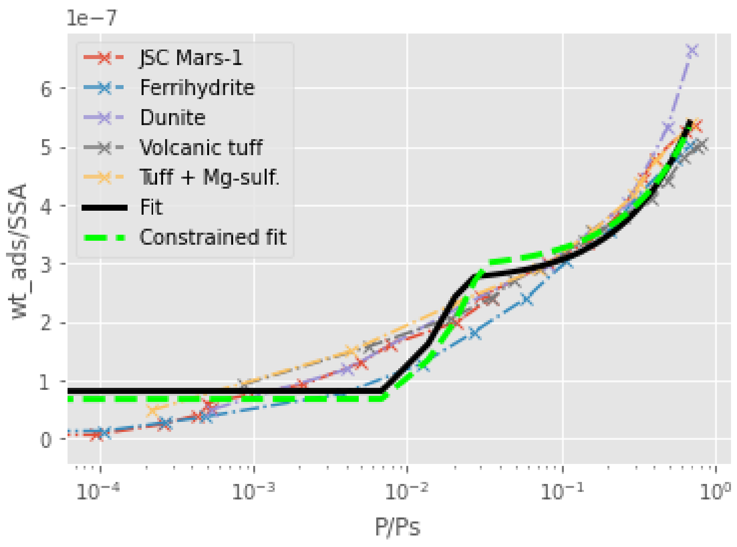


Figure 20: png

3.2.1.2.1 EXTRA: Another attempt at fitting the BET isotherm (see sec. 3.1)

It would *still* be nice to fit a BET isotherm to the data, despite the past infructuous attempts. The equation below is based on sec. 3.1.

$$y = y(x, a, c) = \frac{acx}{(1-x)(1+(c-1)x)}$$

The BET theory only applies for $0.05 < \frac{p}{p_0} < 0.35$. Therefore, the fit should be done with only a subset of the data.

```
xbet = x[(x>BETlim[0]) & (x<BETlim[1])]
ybet = y[(x>BETlim[0]) & (x<BETlim[1])]
```

```
# data_bet = dict(sorted(zip(xbet,ybet), key = lambda elem: elem[0])) # sort by x
# xbet = [float(x) for x in data_bet.keys()]
# ybet = [float(y) for y in data_bet.values()]
```

```
pabet, eabet = optimize.curve_fit(BET_isotherm, xbet, ybet, p0 = [0.064, -71.91])
axa.plot(xd, BET_isotherm(xd, *pabet), label = "BET fit", c="red", ls="dotted", lw=3) #
     Plot BET fit
print("BET fit: ", pabet, eabet)
axa.get_legend().remove() # Remove legend
axa.legend(loc="best") # Redo legend
figa
```

```
BET fit:  [ 2.84907699e-07 -1.69578700e+09] [[ 2.18257060e-17 -6.57924347e+06]
 [-6.57924347e+06  3.88764444e+30]]
```

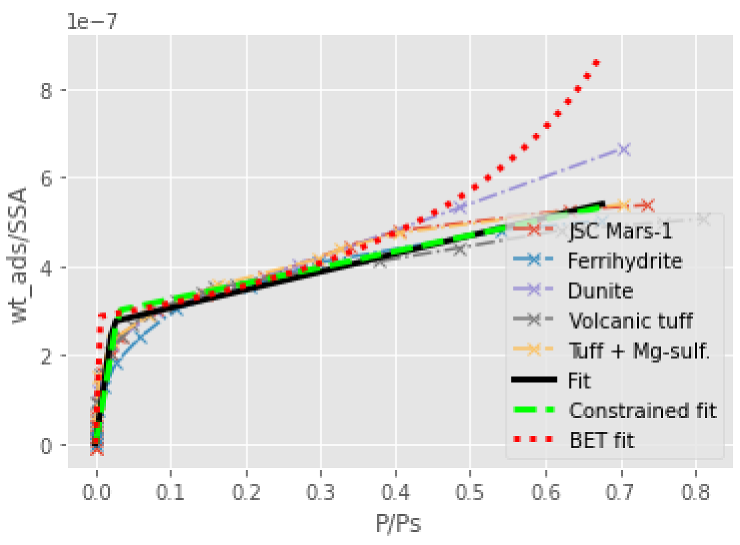


Figure 21: png

The plot in red is the BET fit. It can be seen that `scipy.optimize.curve_fit` does not output the desired/expected results.

```
print("BET fit: ", pabet, eabet)
```

```
BET fit:  [ 2.84907699e-07 -1.69578700e+09] [[ 2.18257060e-17 -6.57924347e+06]
 [-6.57924347e+06  3.88764444e+30]]
```

CONCLUSION: The BET function appears to be not suitable to fit the data or the method is incorrect.

```
plt.plot(xd, BET_isotherm(xd, *pabet), label = "BET fit") # Plot BET fit
plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

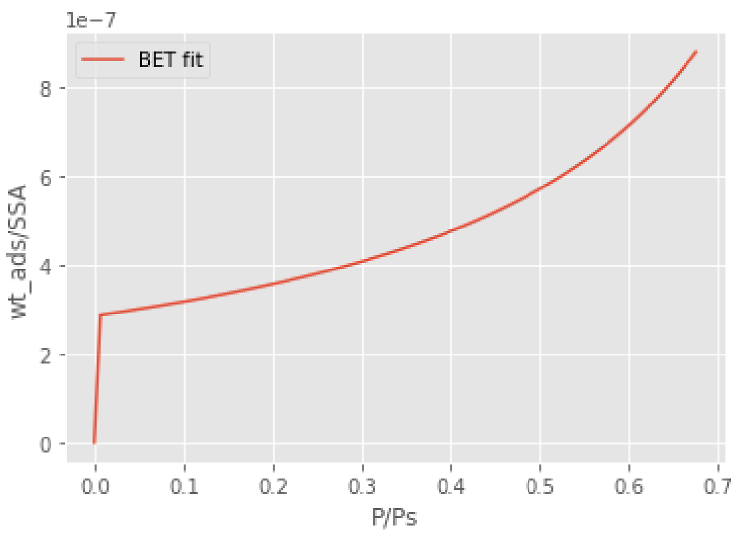


Figure 22: png

3.2.1.2.2 Reducing the number of parameters of the fitting curve In this section, a reduction on the number of parameters will be studied. While it seems that the unconstrained fit is good enough, it may be interesting to set some of the parameters to a computed value.

Fixing y_p to the monolayer saturation value

With the volume of the monolayer v_m, the mass of the molecules in the monolayer m_m can be found. If the expression $v_m = v_{\text{molecule H}_2\text{O}} N_m$ is considered, with $N_m = \frac{m_m}{m_{\text{molecule H}_2\text{O}}} = \frac{m_m}{\frac{M_{\mathrm{H_2O}}}{N_A}}$ (it assumes that the monolayer density is that of the molecule of water) and $M_{\mathrm{H_2O}} = 18.01528\,\text{g mol}^{-1} = 18.01528 \times 10^{-3}\,\text{kg mol}^{-1}$, one can find $m_m = \frac{v_m}{v_{\text{molecule H}_2\text{O}}} m_{\text{molecule H}_2\text{O}} = \frac{v_m}{v_{\text{molecule H}_2\text{O}}} \cdot \frac{M_{\mathrm{H_2O}}}{N_A}$. Normalized by mass sample, one can find the mass ratio of the molecules in the monolayer:

$$\frac{m_m}{m_{\text{sample}}} = \frac{V_m}{v_{\text{molecule H}_2\text{O}}} \cdot \frac{M_{\mathrm{H_2O}}}{N_A}$$

While the volume of the water molecule may be not known, the surface occupied by one molecule is around $S_{\mathrm{H_2O}} = 14\,\text{Å}^2$. With the Specific Surface Area **normalized by sample mass**[7] SSA, the number of molecules that are needed to cover SSA can be found, and therfore m_m. In fact, the number of molecules in the monolayer is given by $N_m = \left\lfloor \frac{SSA}{S_{\mathrm{H_2O}}} \right\rfloor$, so $m_m = \frac{M_{\mathrm{H_2O}}}{N_A} \left\lfloor \frac{SSA}{S_{\mathrm{H_2O}}} \right\rfloor$, and therefore the saturation of the first monolayer happens at:

$$y_p = \frac{m_m}{m_{\text{sample}}} = \frac{1}{m_{\text{sample}}} \frac{M_{\mathrm{H_2O}}}{N_A} \left\lfloor \frac{SSA}{S_{\mathrm{H_2O}}} \right\rfloor$$

Note that from the data in Pommerol et al. (2009), $m_{\text{sample}} = \frac{1}{K0+K1} \cdot \frac{1}{V_m}$.

For each material different m_{sample} and SSA have been found, so the formula above must use representative values of all materials. In a first approach, the averages $\bar{m}_{\text{sample}}$ and $S\bar{S}A$ will be used as the standard deviation is small-ish (not for SSA):

```
msamples = [material.msample for material in materials]
msample = np.mean(msamples)
print("avg msamples =",msample)
print("std msamples =",np.std(msamples))
SSAs = [material.SSA for material in materials]
SSA = np.mean(SSAs)
print("avg SSA =",SSA)
print("std SSA =",np.std(SSAs))
```

```
avg msamples = 0.0007596506772683123
std msamples = 5.8518012637059576e-05
avg SSA = 53566.0
std SSA = 55079.59171961971
```

The known values are:

```
MH2O = 18.01528e-3 # [kg/mol]
SH2O = 14e-10 # [m^2]
```

The value of saturation of the first monolayer normalized by SSA happens at:

```
yp = MH2O/Avogadro*np.floor(SSA/SH2O) / SSA # *1/msample # msample already inside SSA
print(yp)
```

```
2.1367911604339438e-17
```

[7]The SSA in Pommerol et al. (2009) is given in m^2/kg.

y_p can nonetheless be fixed differently in $\mathrm{kg_{ice}}/\mathrm{kg_{regolith}}$. The value of `yp` will be fixed to the saturation of the [first] monolayer as follows:

$$y_p = \frac{v_m}{m_{\text{sample}}} \rho_{\text{ice}} \frac{1}{SSA} = \rho_{\text{ice}} \frac{V_m}{SSA}$$

Assuming ice to have a density of `dens_H2O_ice`, the value is:

```
for material in materials:
    material.yp = dens_H2O_ice*material.Vm/material.SSA # Set monolayer saturation line
        for each material
yp = np.mean([material.yp for material in materials]) # Set monolayer saturation line
    average
[print(material.name,"yp:", material.yp) for material in materials]
print(yp)
```

```
JSC Mars-1 yp: 3.009433962264151e-07
Ferrihydrite yp: 3e-07
Dunite yp: 2.9964664310954064e-07
Volcanic tuff yp: 2.992700729927007e-07
Tuff + Mg-sulf. yp: 2.991150442477876e-07
2.997950313152888e-07
```

```
axbet = plt.gca(title="Monolayer saturation") # ylim=[0, 1.5e-6]

colors = ['r','b','y','m','c','k'] # color list

for material in materials:

    color = colors[0]
    colors = rotate(colors,-1) # shift colors

    # Plot material isotherm
    axbet.plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], ".-", color=color,
         label = material.name)
    axbet.axhline(y = material.yp, color=color, linestyle='--', label = "Monolayer sat.
        for " + material.name)

axbet.axhline(y = yp, color='k', ls='solid', lw=3, label = "Avg. monolayer sat.")

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

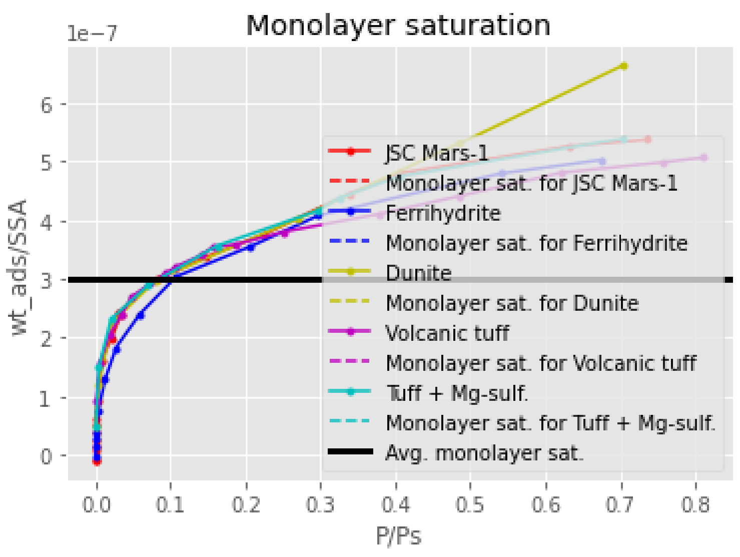


Figure 23: png

The plot of the fit with this constraint `yp` is shown below:

```
figa, axa = plt.subplots()

x = []
y = []

for material in materials:

    xa = material.ads_iso["x"].values # x values
    xa = xa[~np.isnan(xa)] # x values without nans
    x = np.append(x,xa)

    ya = material.ads_iso["wt_ads_SSA"].values # y values
    ya = ya[~np.isnan(ya)] # y values without nans
    y = np.append(y,ya)

    material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", label=material.name, style
        ="-.x", ax=axa)

pam, eam = optimize.curve_fit(mod_piecewise_linear, x, y, p0 = p0_mod_piecewise)
axa.plot(xd, mod_piecewise_linear(xd, *pam), label = "Constrained fit at monolayer
    saturation", lw=3, c="lime", ls="dashed") # Plot constrained fit
print("Constrained fit: ", pam, eam)

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Constrained fit:  [3.04367713e-02 3.60250158e-07] [[5.43076616e-06 4.01473982e-12]
 [4.01473982e-12 3.91364397e-16]]





Text(0, 0.5, 'wt_ads/SSA')
```

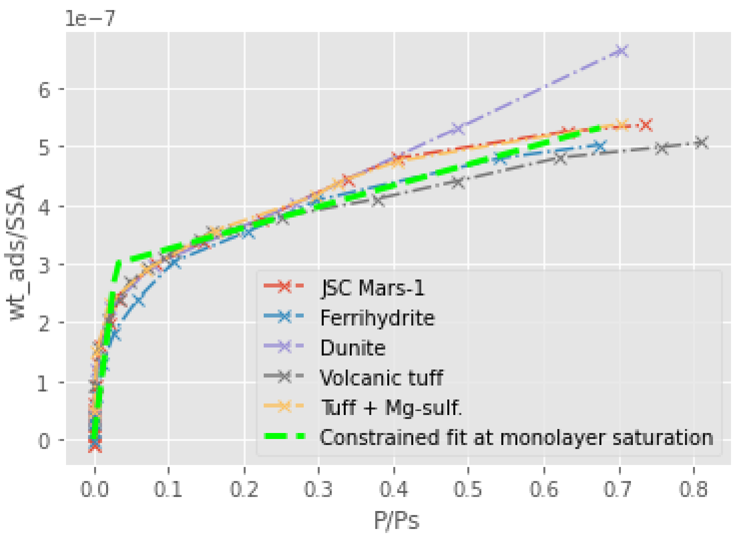


Figure 24: png

If the expressions above are correct, another useful relationship can be found:

$$\frac{1}{v_{\text{molecule H}_2\text{O}}} = (K0 + K1)\left[\frac{SSA}{S_{\text{H}_2\text{O}}}\right]$$

This expression is not currently used, but may interest the reader in the future. It is presented here as extra information.

3.2.1.3 Adding the dependance on temperature In sec. 2, it was stated that if the rate of change of the adsorption isotherm—with respect to temperature—equals that of $\frac{p}{p_0}$, the shape of the isotherms (see fig. 4) does not change in a n-vs-$\frac{p}{p_0}$ plot. The shape of the isotherm may change with respect to the temperature, so it may be interesting to ensure that the two-segment fit adapts to these changes.

The simple fit presented here is described by two segments, each of which with its own slope. The slope is given by an adsorption energy Q_{ads} (see fig. 13), i.e. the difference between E_1 (the heat of adsorption of the gas in the first adsorbed layer) and E_L (the heat of liquefaction of the gas) (**?TODO**)[8]. If Q_{ads} is equal to the enthalpy of sublimation H_{subl}, the slopes do not change shape with respect to $\frac{p}{p_0}$. In the BET theory, the variable that quantifies this energy Q_{ads} is named the BET adsorption constant c (C in Pommerol et al. (2009)) and is calculated as:

$$c = 1 + \frac{K_1}{K_0} = e^{\frac{E_1 - E_L}{RT}} \tag{5}$$

Note that the lower the energy, the higher the sublimation enthalpy.

[8]Note that if $E_1 - E_L = 10$ and $E_L = 50$ kJ mol^{-1}, then $E_1 = 50 + 10 = 60$ kJ mol^{-1}.

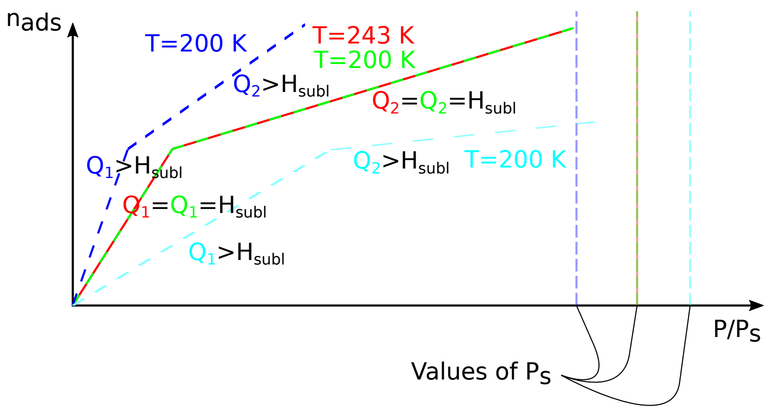


Figure 25: Change of slope for a set of temperatures

As seen shown in `soilwater.F90` (see sec. 6.2.2) and in Weinmann and Meslin (2019), Q_{ads} to be used in the LMDZ GCM may take the values 21 kJ mol^{-1}, 35 kJ mol^{-1}, and 60 kJ mol^{-1}. Although values around 21 kJ mol^{-1} have been found from data from Mars missions, values of $\sim$ 60 kJ mol^{-1} would rather be expected. For isothermes of type I, II, and IV, it can be seen that $Q_1 \geq Q_2$ (i.e. the slope of segment 1 is higher than the slope of segment 2), so the following combinations may be carried out:

- $Q_1 = Q_2 =$ 21 kJ mol^{-1}
- $Q_1 =$ 60 kJ mol^{-1} and $Q_2 =$ 21 kJ mol^{-1}
- $Q_1 =$ 60 kJ mol^{-1} and $Q_2 =$ 55 kJ mol^{-1}
- $Q_1 > Q2_{\geq} H_{\text{subl}}$

Note that for isotherms of type III and V $Q_2 \geq Q_1$, so the values above may be exchanged (Q_1 takes the value of Q_2 above, and conversely for Q_2) for the simulation.

The expression in eq. 5 is valid for the range of temperatures of 150 $<$ T $<$ 273 K and at least for type 2 and type 3 isotherms. Therefore, a number of temperatures is selected:

```
Ntemps = 5 # Number of temperatures to plot/evaluate
# Ts = np.linspace(150, 273, Ntemps)
Ts = np.linspace(230, 273, Ntemps-1) # Init vector of Ntemps-1 values so as to add Tref
    in the Ts array
Ts = np.append(Ts, Tref)
Ts = np.round(np.sort(Ts),1)
```

The Q_{ads} is computed with data from Pommerol et al. (2009) using the aforementioned formula:

```
# Qadss = [21e3, 35e3, 60e3] # [J/mole] Enthalpy of adsorption. soilwater.F90
    alternative values: 21e3, 35e3, and 60e3
# Q1ads = random.choice(Qadss)
# Q2ads = random.choice([Qads for Qads in Qadss if Qads >= Q1ads])
```

```
for material in materials:
    # material.Q1ads = Q1ads
    # material.Q2ads = Q2ads
    material.Qads = R * material.T * np.log(material.C) # The temperature to be used is
        that of the experiments
    material.Q1ads = material.Qads # First segment loosely corresponds to the linear
        part
    material.Q2ads = material.Qads * 0.5

    print(material.name + ": Qads=" + str(round(material.Qads,2)) + " Q1ads=" + str(
        round(material.Q1ads,2)) + " Q1ads=" + str(round(material.Q2ads,2)))
```

```
Q1ads = np.mean([material.Q1ads for material in materials])
Q2ads = np.mean([material.Q2ads for material in materials])
```

```
JSC Mars-1: Qads=9371.9 Q1ads=9371.9 Q1ads=4685.95
Ferrihydrite: Qads=8150.84 Q1ads=8150.84 Q1ads=4075.42
Dunite: Qads=9686.14 Q1ads=9686.14 Q1ads=4843.07
Volcanic tuff: Qads=10115.46 Q1ads=10115.46 Q1ads=5057.73
Tuff + Mg-sulf.: Qads=10663.98 Q1ads=10663.98 Q1ads=5331.99
```

The function `temp_piecewise_linear` defines a piecewise linear function—just like `piecewise_linear`—in terms of the slopes and intersection point:

```
def temp_piecewise_linear(x, m1, m2, yp):
    xp = yp / m1 # From yp = m1 * xp as first segment ends at point (xp, yp)
    return piecewise_linear(x, xp, yp, m2)
```

By taking into account eq. 29 and eq. 30, which describe each of the segments, a piecewise function can be written:

$$\begin{cases} n_{\text{ads}} = \underbrace{k_{\text{ads}} \exp\left(\frac{Q_1}{RT}\right)}_{m_1} n_{\text{vap}} & n_{\text{ads}} \leq n_{\text{sat}} \\ n_{\text{ads}} = \underbrace{k'_{\text{ads}} \exp\left(\frac{Q_2}{RT}\right)}_{m_2} n_{\text{vap}} + \underbrace{C_0}_{n_2} & n_{\text{ads}} > n_{\text{sat}} \end{cases}$$

Note that eq. 31 may be used if the values are given for a reference temperature.

```
print(Tref, Ts)
```

```
243.0 [230.  243.  244.3 258.7 273. ]
```

For each material:

```
for material in materials:

    xtemp = material.ads_iso["x"].values # x values
    xtemp = xtemp[~np.isnan(xtemp)] # x values without nans

    ytemp = material.ads_iso["wt_ads_SSA"].values # y values
    ytemp = ytemp[~np.isnan(ytemp)] # y values without nans

    # Optimize for each material
    [material.xp, material.m2], etempm = optimize.curve_fit(mod_piecewise_linear, xtemp,
         ytemp, p0 = p0_mod_piecewise)

    material.m1 = material.yp / material.xp
```

If the values of the fitting using `mod_piecewise_linear` are used:

```
xp = pam[0] # From mod_piecewise_linear
m2m = pam[1] # From mod_piecewise_linear
m1m = yp / xp # Derived from valeus of mod_piecewise_linear
```

With $k_{\text{ads}}^{\text{ref}} = m_1 \cdot SSA \cdot \rho_{\text{regolith}}$, $k_{\text{ads}}^{\prime,\text{ref}} = m_2 \cdot SSA \cdot \rho_{\text{regolith}}$, and $n_2 = y_p - m_2 x_p = y_p \left(1 - \frac{m_2}{m_1}\right)$, with $m_1 = \frac{y_p}{x_p}$ as segments join at (x_p, y_p).

```
for material in materials:
    material.kads_ref  = material.m1 * material.SSA * material.dens # In [kg ads / m^3
        regolith]
    material.kpads_ref = material.m2 * material.SSA * material.dens # In [kg ads / m^3
        regolith]
    print(material.name + " at " + str(material.T) + " K"+ ": kads_ref=" + str(material.
       kads_ref) + " kpads_ref=" + str(material.kpads_ref) + ")")
```

```
JSC Mars-1 at 243.0 K: kads_ref=1320.57401243346 kpads_ref=52.1940271758197)
Ferrihydrite at 243.0 K: kads_ref=1299.609245194286 kpads_ref=58.93641745905516)
Dunite at 243.0 K: kads_ref=106.62520986169116 kpads_ref=1.8334467636703906)
Volcanic tuff at 243.0 K: kads_ref=145.18581893940504 kpads_ref=5.18679398012704)
Tuff + Mg-sulf. at 243.0 K: kads_ref=508.8942706053109 kpads_ref=5.524879846745572)
```

IMPORTANT: Below, instead of working with the aforementioned $k_{\text{ads}}^{\text{ref}}$ and $k_{\text{ads}}^{\prime,\text{ref}}$, the proportional values $k_{\text{ads}}^{\text{ref}} = m_1$ and $k_{\text{ads}}^{\prime,\text{ref}} = m_2$ are used to scale the result in the y–`wt_ads`/`SSA` axis.

```
from decimal import Decimal, ROUND_HALF_UP
```

```
n = len(materials) + 1 # Plot for each material plus the plot with the fit
cols = 2
rows = int(Decimal(n/cols).quantize(Decimal('1'), rounding=ROUND_HALF_UP)) # Python's
    round() implements IEEE standard

figtemp, axtemp = plt.subplots(rows, cols, figsize=(10,10), constrained_layout=True)
figtemp.suptitle("Isotherm dependance on temperature")

if n % 2 != 0:
    figtemp.delaxes(axtemp[-1, -1]) # Delete last subplot if n is odd

i = 0

colors = ['r','b','y','m','c','k'] # color list
opacities = np.geomspace(0.3, 1, num=Ntemps) # As many alpha/transparency values as
    Ntemps. geomspace is similar to logscale

for material in materials:

    color = colors[0]
    colors = rotate(colors,-1) # shift colors

    # Current plot
    ci = i%rows
    cj = int(i/rows)

    print("i=" + str(i), "ci=" + str(ci), "cj=" + str(cj))
    op = 0 # For opacities

    for T in Ts: # Ts has length Ntemps

        xtemp = material.ads_iso["x"].values # x values
        xtemp = xtemp[~np.isnan(xtemp)] # x values without nans

        opacity = opacities[op]

        # m1 = material.kads_ref*np.exp(material.Q1ads / R * (1/T - 1/material.T))
        # m2 = material.kpads_ref*np.exp(material.Q2ads / R * (1/T - 1/material.T))
        # axtemp[ci, cj].plot(xtemp, temp_piecewise_linear(xtemp, m1, m2, yp), "-",
            color=color, alpha=opacity, label = material.name + " at " + str(T) + " K (
            Q1 != Q2)")

        m1 = material.m1*np.exp(material.Qads / R * (1/T - 1/material.T))
        m2 = material.m2*np.exp(material.Qads / R * (1/T - 1/material.T))
        axtemp[ci, cj].plot(xtemp, temp_piecewise_linear(xtemp, m1, m2, material.yp), "-
            ", color=color, alpha=opacity, label = str(T) + " K ")

        if(T == Tref):
            axtemp[ci, cj].plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], "
                .", color=color, alpha=opacity, label ="Data at " + str(material.T) + "
                K")

        kads  = m1 * material.SSA * material.dens # In [kg ads / m^3 regolith]
        kpads = m2 * material.SSA * material.dens # In [kg ads / m^3 regolith]
        print(material.name + " at " + str(T) + " K"+ ": kads=" + str(kads) + " kpads="
            + str(kpads) + ")")

        axtemp[ci, cj].set_ylim([0, 6e-7])
        axtemp[ci, cj].legend(loc="lower right")
        axtemp[ci, cj].set_xlabel("P/Ps")
        axtemp[ci, cj].set_ylabel("wt_ads/SSA")
        axtemp[ci, cj].set_title(material.name)

        op = op + 1

    i = i + 1

# Last plot
ci = i%rows
cj = int(i/rows)

op = 0

for T in Ts:

    opacity = opacities[op]

    m1 = m1m*np.exp(Q1ads / R * (1/T - 1/Tref))
    m2 = m2m*np.exp(Q2ads / R * (1/T - 1/Tref))
    axtemp[ci, cj].plot(xd, temp_piecewise_linear(xd, m1, m2, yp), "-", color='firebrick
        ', alpha=opacity, label = "Fit at " + str(T) + " K") # Plot constrained fit

    op = op + 1

 # Overlay isotherms
[axtemp[ci, cj].plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], ".", color='
    firebrick', label = "Data at " + str(material.T) + " K") for material in materials]

axtemp[ci, cj].set_ylim([0, 6e-7])
axtemp[ci, cj].set_title("Constrained fit")
axtemp[ci, cj].set_xlabel("P/Ps")
axtemp[ci, cj].set_ylabel("wt_ads/SSA")
axtemp[ci, cj].legend(bbox_to_anchor=(1, 1))
```

```
i=0 ci=0 cj=0
JSC Mars-1 at 230.0 K: kads=1716.4333764766814 kpads=67.83987073335045)
JSC Mars-1 at 243.0 K: kads=1320.57401243346 kpads=52.1940271758197)
JSC Mars-1 at 244.3 K: kads=1288.3765915445617 kpads=50.921464604510476)
JSC Mars-1 at 258.7 K: kads=996.564498727854 kpads=39.38795859931374)
JSC Mars-1 at 273.0 K: kads=793.207593575098 kpads=31.350532651202904)
i=1 ci=1 cj=0
Ferrihydrite at 230.0 K: kads=1632.4565000620407 kpads=74.03081974614715)
Ferrihydrite at 243.0 K: kads=1299.609245194286 kpads=58.93641745905516)
Ferrihydrite at 244.3 K: kads=1272.0072035945545 kpads=57.684683176262496)
Ferrihydrite at 258.7 K: kads=1017.3827703262449 kpads=46.13763397676792)
Ferrihydrite at 273.0 K: kads=834.218843818052 kpads=37.83127137120432)
i=2 ci=2 cj=0
Dunite at 230.0 K: kads=139.8111934955621 kpads=2.4040879316611865)
Dunite at 243.0 K: kads=106.62520986169116 kpads=1.8334467636703906)
Dunite at 244.3 K: kads=103.93948043081902 kpads=1.7872649841501982)
Dunite at 258.7 K: kads=79.7082556328629 kpads=1.370603101437748)
Dunite at 273.0 K: kads=62.95950933455481 kpads=1.0826042807460732)
i=3 ci=0 cj=1
Volcanic tuff at 230.0 K: kads=192.67362832867798 kpads=6.883305978812678)
Volcanic tuff at 243.0 K: kads=145.18581893940504 kpads=5.18679398012704)
Volcanic tuff at 244.3 K: kads=141.36886451567509 kpads=5.050432478900225)
Volcanic tuff at 258.7 K: kads=107.14381070035611 kpads=3.827735218275138)
Volcanic tuff at 273.0 K: kads=83.74995310283164 kpads=2.9919847252505307)
i=4 ci=1 cj=1
Tuff + Mg-sulf. at 230.0 K: kads=685.7879677322817 kpads=7.445350323079749)
Tuff + Mg-sulf. at 243.0 K: kads=508.8942706053109 kpads=5.524879846745572)
Tuff + Mg-sulf. at 244.3 K: kads=494.8000319633303 kpads=5.3718638284365445)
Tuff + Mg-sulf. at 258.7 K: kads=369.41540102872096 kpads=4.010608533268313)
Tuff + Mg-sulf. at 273.0 K: kads=284.9254808068227 kpads=3.0933322256929303)




<matplotlib.legend.Legend at 0x7f900a9fb700>
```

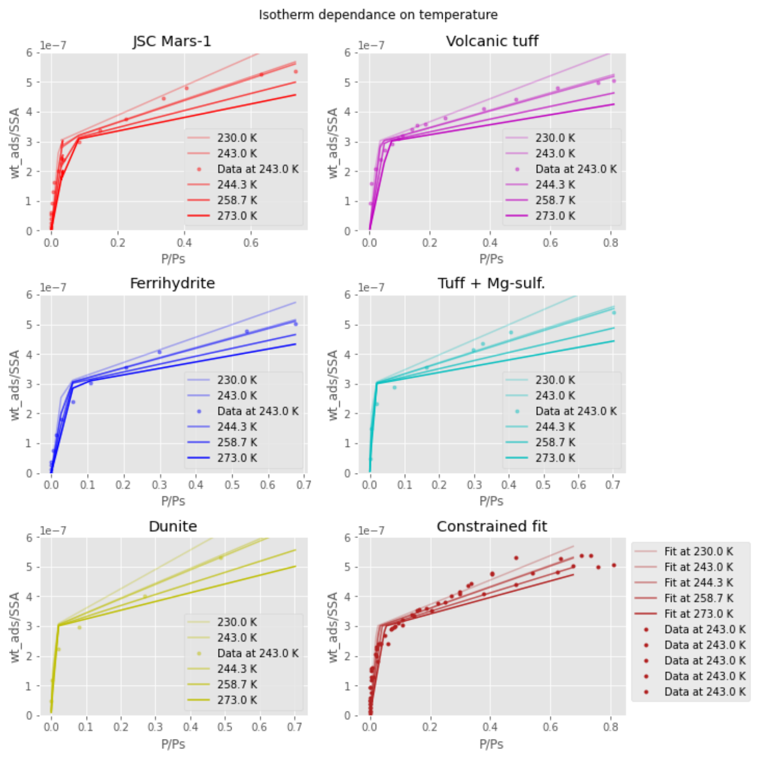


Figure 26: png

3.2.2 Testing alternative functions

3.2.2.1 The alpha function P.-Y. Meslin also suggested to use the formula below (from Zent et al. (1986), Zent et al. (1993) and also from the appendices of this paper), where α is the vapour adsorbed onto regolith grains in kg/m^3 (volume of regolith), ρ_r is the regolith density in kg/m^3, $\beta = 2.043 \times 10^{-8}$ Pa^{-1}, $\delta = -2679.8$ K, T is the temperature, k_B is the Boltzmann constant, $p_{H_2O} = \frac{nk_BT}{m_w}$ is the partial pressure of water, and m_w is the mass of a water molecule.

$$\alpha(n,T) = \frac{\rho_r\beta}{e^{\delta/T}}(p_{H_2O})^{0.51} = \frac{\rho_r\beta\sqrt{n}}{e^{\delta/T}}\left(\frac{k_BT}{m_w}\right)^{0.51} = F(T)\sqrt{n}$$

```python
def alpha(n, T, dens_rego):
    m_w = 2.988e-26 # Alternatively MH2O/Avogadro
    beta = 2.043e-8 # [Pa^(-1)]
    delta = -2679.8 # [K]
    return dens_rego * beta * np.sqrt(n) / np.exp(delta/T) * (Boltzmann*T/m_w)**0.51
```

Note that this function **may ONLY be used on basalt** and basalt-like materials.

```
def dens(M,P,T):
    return P / ( (R/M) * T )
```

Do not mistake the density of the water vapour (in kg/m^3) with the vapour density of water (non dimensional):

```
Matmo = 43.34e-3 # [kg/mol] Mean molecular weight of the Martian atmosphere from https
    ://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html
# Matmo = 28.97e-3 # [kg/mol] Mean molecular weight of the air  (Earth)
vapour_dens_H2O = MH2O / Matmo
print(vapour_dens_H2O)
```

```
0.41567328103368717
```

Plots of adsorbed water y against pressure x:

```
n = len(materials) # Number of plots
cols = 2
rows = int(Decimal(n/cols).quantize(Decimal('1'), rounding=ROUND_HALF_UP)) # Python's
    round() implements IEEE standard

figalpha, axalpha = plt.subplots(rows, cols, figsize=(8,8), constrained_layout=True)
figalpha.suptitle("Isotherms with function alpha")

if n % 2 != 0:
    figalpha.delaxes(axalpha[-1, -1]) # Delete last subplot if n is odd

colors = ['r','b','y','m','c','k'] # color list

i = 0

for material in materials:

    color = colors[0]
    colors = rotate(colors,-1) # shift colors

    # Current plot
    ci = i%rows
    cj = int(i/rows)

    print("i=" + str(i), "ci=" + str(ci), "cj=" + str(cj))

    # Data to be plotted
    x = material.ads_iso["x"]
    y = material.ads_iso["wt_ads_SSA"]

    axalpha[ci, cj].plot(x, y, ".-", color=color, label = "Data")

    # Plot of alpha()
    y_alpha = alpha(dens(MH2O, x*material.Ps, material.T), material.T, material.dens) /
        ( material.dens * material.SSA ) # [kg ads/m^3 regolith / ( (kg regolith / m^3
        regolith) * SSA ) = kg ads / (kg regolith * SSA)]
    axalpha[ci, cj].plot(x, y_alpha, "--", color=color, label = "Function alpha") # Plot
         alpha

    axalpha[ci, cj].legend(loc="right")
    axalpha[ci, cj].set_xlabel("P/Ps")
    axalpha[ci, cj].set_ylabel("wt_ads/SSA")
    axalpha[ci, cj].set_title(material.name)
    i=i+1
```

```
i=0 ci=0 cj=0
i=1 ci=1 cj=0
i=2 ci=2 cj=0
i=3 ci=0 cj=1
i=4 ci=1 cj=1
```

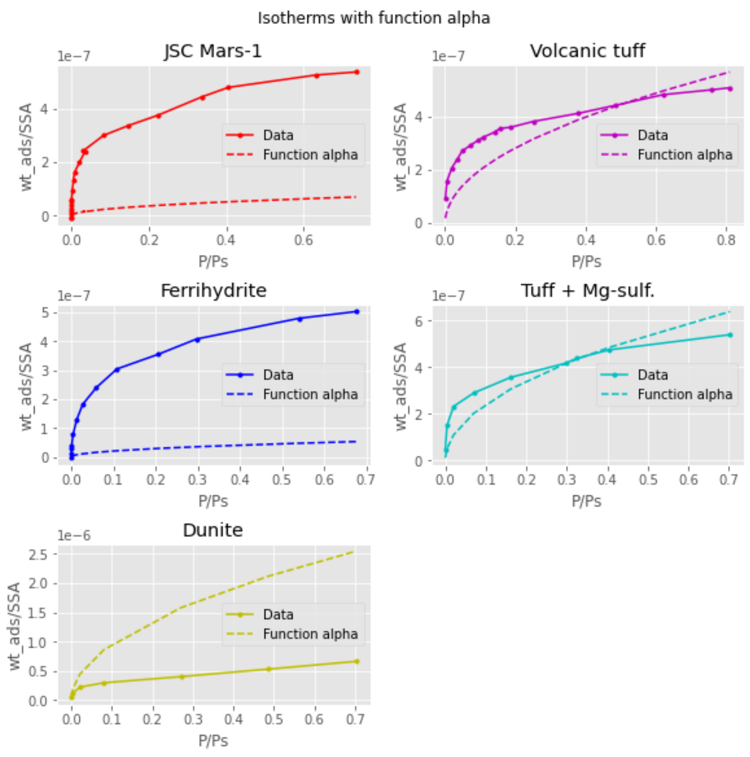


Figure 27: png

- **TODO Apparently β is given for a specific SSA, and this was not taken into account**
- As stated above, the `alpha` function may only be used on basalt-like materials, and maybe this is why discrepances arise.

The average regolith may be defined as having an SSA equal to the arithmetic or **harmonic mean**—`SSA` and `SSAh`, respectively—of the list of SSA (of each material). Temperature T and P_s are set according to the values in Pommerol et al. (2009). The value of 1300 kg/m^3 is given by $0.45 \cdot$ 3000 kg/m^3 (accounting for a 45% of porosity in a rock of density 3000 kg/m^3). Other values can be found in papers, such as 870 kg/m^3 for JSC Mars-1, as suggested in this paper.

```
SSAh = stats.hmean(SSAs) # Harmonic mean
print([SSA, SSAh])
```

```
[53566.0, 9403.05963661315]
```

```
axalphaall = plt.gca(ylim=[0, 1.5e-6], title="Function alpha for the average regolith")

for material in materials:

    # Plot material isotherm
    axalphaall.plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"], "-.x", label
        = material.name)
```

```
aa = alpha(dens(MH2O, xd*Ps, Tref), Tref, dens_rego)

y_alphah = aa / ( dens_rego * SSAh )
axalphaall.plot(xd, y_alphah, "k-", lw=3, label = "Alpha (harmonic mean SSA)") # Plot
    alpha

y_alpha = aa / ( dens_rego * SSA )
axalphaall.plot(xd, y_alpha, "r-", lw=3, label = "Alpha (arithmetic mean SSA)") # Plot
    alpha

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

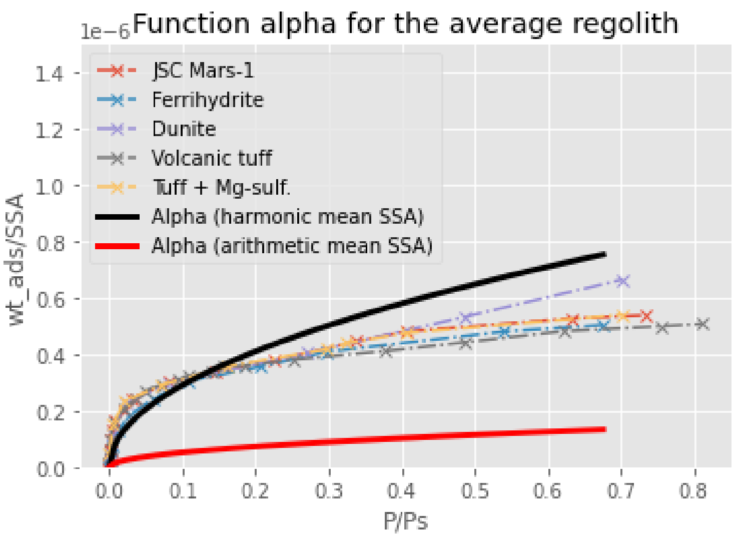


Figure 28: png

Below there the data has been fit with the following function, where $F\left(T\right)$ is a constant at a given temperature:

$$\alpha_{\text{fit}} = F\left(T\right)\sqrt{n}$$

```
def alphafit(n, F): # independent variable and then the fitting params
    return F * np.sqrt(n)
```

The current regolith data is plotted in water adsorbed in regolith grains in kg/m^3 against pressure in Pa

```
axalphafit = plt.gca(title="Fitting an alpha-like function to the isotherms")

x = []
y = []

for material in materials:

    xa = material.ads_iso["x"].values # x values
    xa = xa[~np.isnan(xa)] # x values without nans
    x = np.append(x,xa)

    ya = material.ads_iso["wt_ads_SSA"].values # y values in kg ads / m^3 regolith
    ya = ya[~np.isnan(ya)] # y values without nans
    y = np.append(y,ya)

    material.ads_iso.plot(kind="line", x="x", y="wt_ads_SSA", label=material.name, style
        ="-.x", ax=axalphafit)

palphafit, ealphafit = optimize.curve_fit(alphafit, x*Ps, y, p0 = [SSAh])
axalphafit.plot(xd, alphafit(xd*Ps, *palphafit), '-', label = "Alpha fit", lw=3, c="k",
    ls="solid") # Plot alpha fit
print("Alpha fit: ", palphafit, ealphafit)

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Alpha fit:  [1.19578066e-07] [[1.08599541e-17]]





Text(0, 0.5, 'wt_ads/SSA')
```

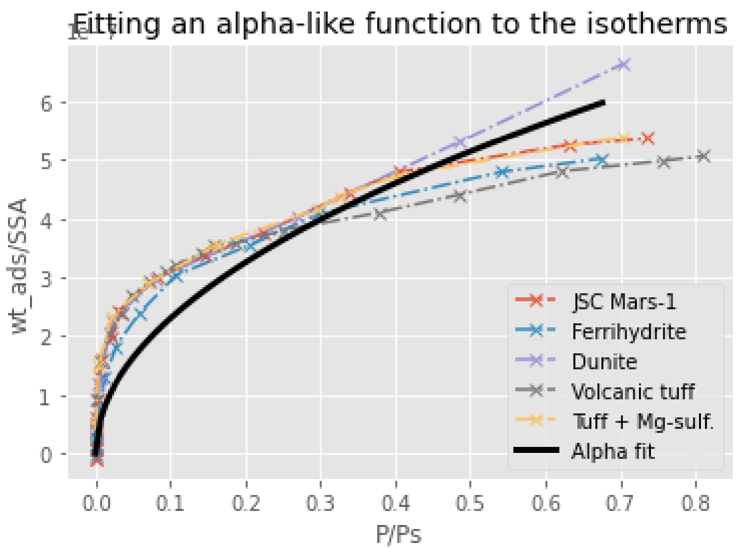


Figure 29: png

```
# data = sorted(zip(x, y), key = lambda elem: elem[0]) # Sort by values of x. elem[0] is
     the first element of the tuple (x,y), so y
# x,y = list(zip(*data)) # "unzip" data as tuple

# x = np.array(x) # tuple to array
# y = np.array(y) # tuple to array
```

3.2.2.1.1 Associated error (RMSE)

Create an array containing data arrays:

```
x = []
y = []

for material in materials:

    xa = material.ads_iso["x"].values # x values
    xa = xa[~np.isnan(xa)] # x values without nans
    x.append(xa) # np.append(x,xa) is not to be used here
    ya = material.ads_iso["wt_ads_SSA"].values # y values in kg ads / m^3 regolith
    ya = ya[~np.isnan(ya)] # y values without nans
    y.append(ya) # np.append(y,ya) is not to be used here
```

A function to compute the quality of the fit is created (also partially inspired by this):

```
def get_fit_info(xdata, ydata, yfun, p): # xdata and ydata are arrays of arrays, e.g
    xdata = [[...x_iso1...], [...x_iso2...],...]

    assert len(xdata) == len(ydata) # basic data consistency check

    N = len(ydata) # number of observations [of the ground]/isotherms available
    ssr = []

    for i in range(N):
        y = ydata[i]
        x = xdata[i]
        residuals = y - yfun(x, *p) # residuals for each isotherm with respect to the
            fit
        ssr.append(np.sum(residuals**2))

    # This rmse is used to compare differences between the fit and the isotherms,
        neither of which is accepted as the "standard" (the true Martian underground
        isotherm is not known)
    rmse = np.sqrt(np.sum(ssr)/N) # root mean square error

    # The following is not applicable because the true isotherm is not known
    # sst = np.sum((y - np.mean(y))**2) # sum of squares or mean square error (mse)
    # ssr = np.sum(residuals**2) # residual sum of squares
    # sst = np.sum((ydata - np.mean(ydata))**2) # total sum of squares
    # r2 = 1 - ssr/sst

    return dict(zip(["residuals", "ssr", "rmse"], [residuals, ssr, rmse]))
```

The RMSE is found for α_{fit} and `piecewise_linear()` (and all desired fits):

```
alphafit_info = get_fit_info(x, y, alphafit, palphafit)
piecewise_linear_info = get_fit_info(x, y, piecewise_linear, pa)
mod_piecewise_linear_info = get_fit_info(x, y, mod_piecewise_linear, pm) # It obviously
    depends on the constraint
```

They can be compared as follows:

```
rmse = dict([("alphafit", alphafit_info["rmse"]), ("piecewise_linear",
    piecewise_linear_info["rmse"]), ("mod_piecewise_linear", mod_piecewise_linear_info["
    rmse"])]) # Store data in a dict
print(sorted(rmse.items(), key = lambda x: x[1])) # Sort by RMSE value
```

```
[('piecewise_linear', 1.5855169512707795e-07), ('mod_piecewise_linear',
    1.7587826971207154e-07), ('alphafit', 9.879120321119228e-07)]
```

The fits are listed above according to their error. The lower the error, the better the fit.

3.2.2.2 Superimposing additional data

The additional data is treated in the `additional_data.ipynb` notebook. Herebelow the data is imported. All variables in the other notebook will be available here. The notebook is attached in the appendices in sec. 10.6.

```
%%capture
# %%capture magic from https://nbviewer.jupyter.org/github/ipython/ipython/blob/1.x/
    examples/notebooks/Cell%20Magics.ipynb
%run additional_data.ipynb
```

Herebelow is a plot with data from `additional_data.ipynb` and from Pommerol et al. (2009), without taking into account that they vary with the temperature:

```
Ts = [i for i in Pss.keys()]
if Tref not in Ts: # Only append if Tref present in Ts
    Ts = np.append(Ts, Tref)
Ts = np.round(np.sort(Ts),1)
Ntemps = len(Ts) # Number of temperatures
print(Ts)
```

```
[220 240 243 253 255 257 260 263 268 273 275 280 293 298 302 313 333]
```

In the plot below, the `winter` colormap is for data from Pommerol et al. (2009), and the `autumn` colormap is for data from `additional_materials`.

```
n = len(Ts) # Number of plots=every material + all materials
cols = 4
rows = int(Decimal(n/cols).quantize(Decimal('1'), rounding=ROUND_HALF_UP)) # Python's
    round() implements IEEE standard

if cols*rows < n:
    rows=rows + 1

figaddtemp, axaddtemp = plt.subplots(rows, cols, figsize=(15,15), constrained_layout=
    True)
figaddtemp.suptitle("Additional materials")

if cols*rows-n != 0:
    [figaddtemp.delaxes(axaddtemp[-i, -1]) for i in range(1,cols*rows-n+1)] # Delete
        last subplots if n is odd

# colors = list(mcolors.TABLEAU_COLORS.keys()) # color list https://matplotlib.org
    /3.1.0/gallery/color/named_colors.html

i = 0

for T in Ts: # Ts has length Ntemps

    colors1 = [ cm.winter(x) for x in np.linspace(0, 1, len(materials)) ]
    colors2 = [ cm.autumn(x) for x in np.linspace(0, 1, len(additional_materials)) ]

    # Current plot
    ci = i%rows
    cj = int(i/rows)

    # print("i=" + str(i), "ci=" + str(ci), "cj=" + str(cj))

    # Pommerol2009 data
    for material in materials:
        xtemp = material.ads_iso["x"].values # x values
        xtemp = xtemp[~np.isnan(xtemp)] # x values without nans

        # color = 'gray' # Reset color
        # opacity = 1 # Reset opacity

        color = colors1[0]
        colors1 = rotate(colors1,-1) # shift colors

        m1 = material.m1*np.exp(material.Qads / R * (1/T - 1/material.T))
        m2 = material.m2*np.exp(material.Qads / R * (1/T - 1/material.T))
        axaddtemp[ci, cj].plot(xtemp, temp_piecewise_linear(xtemp, m1, m2, yp), color=
            color, alpha=opacity, label = material.name + " (fit) at " + str(T) + " K ",
             zorder=0)
        kads  = m1 * material.SSA * material.dens # In [kg ads / m^3 regolith]
        kpads = m2 * material.SSA * material.dens # In [kg ads / m^3 regolith]
        # print(material.name + " at " + str(T) + " K"+ ": kads=" + str(kads) + " kpads
            =" + str(kpads) + ")")

        if(T == Tref):
            axaddtemp[ci, cj].plot(material.ads_iso["x"], material.ads_iso["wt_ads_SSA"
                ], ".", color=color, alpha=opacity, zorder=1, label=material.name + "
                data at " + str(material.T) + " K")

        # Pommerol_patch, a = axaddtemp[ci, cj].get_legend_handles_labels()
        Pommerol_patch = mpatches.Patch(color=color, label='Pommerol2009 data')

    # Additional data
    for curr_add_material, additional_material in additional_materials.items():
        for labl, vals in additional_material.items():
            if str(T) in labl.split("_"): # Check if the isotherm temperature is in Pss

                # color = 'r' # Reset color
                # opacity = 1 # Reset opacity

                color = colors2[0]
                colors2 = rotate(colors2,-1) # shift colors

                axaddtemp[ci, cj].plot(vals["x"], vals["y"], color=color, alpha=opacity,
                     label=curr_add_material + " " + labl, zorder=1)
                # axaddtemp[ci, cj].plot(vals["x"], vals["y"], label=curr_add_material +
                     " " + labl, zorder=1)

        additional_patch = mpatches.Patch(color=color, label='Additional data')

    axaddtemp[ci, cj].set_yscale('log')
    axaddtemp[ci, cj].set_ylim([1e-8, 3e-6])
    axaddtemp[ci, cj].set_title("Isotherms at " + str(T) + "K")
    axaddtemp[ci, cj].set_xlabel("P/Ps")
    axaddtemp[ci, cj].set_ylabel("wt_ads/SSA")
    # axaddtemp[ci, cj].legend(handles=[Pommerol_patch, additional_patch], loc="lower
        right")
    axaddtemp[ci, cj].legend(loc="lower right")

    i=i+1
```

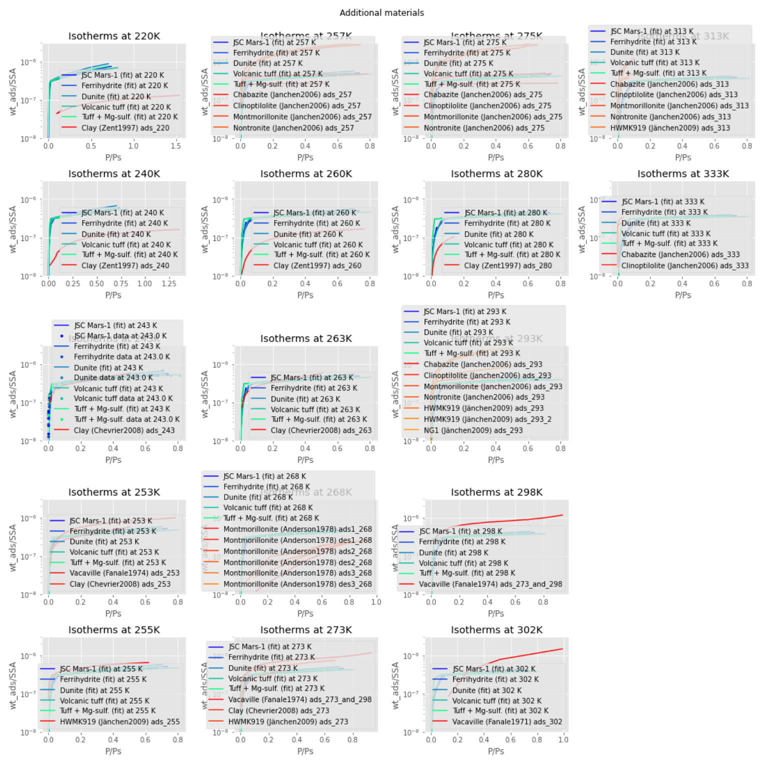


Figure 30: png

In order to plot `additional_materials` at a wide range of temperatures one could take into account the effect of temperature with data from figures such as #fig:q_vs_wt_ads_nontronite.

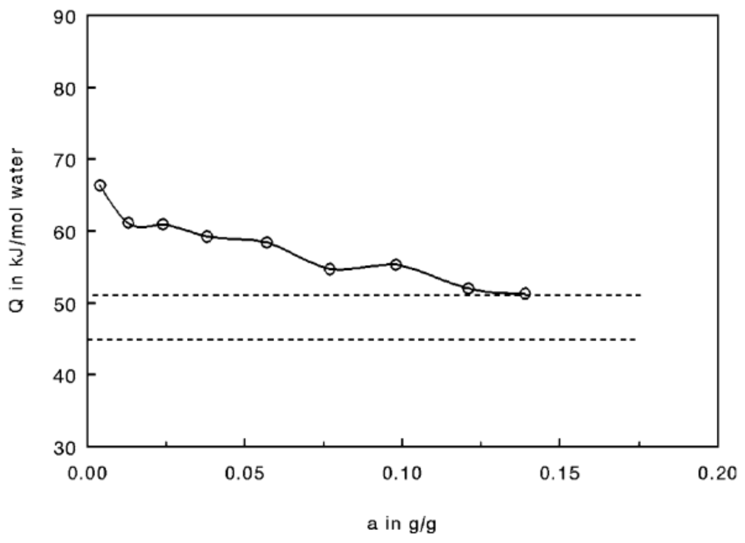


Fig. 9. Differential molar heats of adsorption as function of the adsorbed amount for water/nontronite at 303 K; dashed lines indicate the heat of vaporization of water and the heat of sublimation of water ice.

Figure 31: Differential molar heats of adsorption as function of the adsorbed amount for water/nontronite at 303 K

However, in this study—which is meant to be relatively simple—only the isotherms closest to `Tref` will be used.

```
Tref
```

```
243.0
```

Therefore, the isotherms that will be used are the following:

```
a = vcvlle_1
vcvlle_1 = Material()
vcvlle_1.simple_material("Vacaville (Fanale1971)", "vcvlle_1", Fanale1971_SSA, T=302,
    dens=1300)
vcvlle_1.msample = 6e-3
vcvlle_1.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_302"]["x"]),
    "wt_ads_SSA": np.array(a["ads_302"]["y"])
})

a = vcvlle
vcvlle = Material()
vcvlle.simple_material("Vacaville (Fanale1974)", "vcvlle", Fanale1974_SSA, T=253, dens
    =1300)
vcvlle.msample = 1e-3
vcvlle.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_253"]["x"]),
    "wt_ads_SSA": np.array(a["ads_253"]["y"])
})

a = mmllote_1
mmllote_1 = Material()
mmllote_1.simple_material("Montmorillonite (Anderson1978)", "mmllote_1",
    Anderson1978_SSA, T=268, dens=1300)
mmllote_1.msample = 73.8e-6 # Anderson1978
mmllote_1.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads1_268"]["x"]),
    "wt_ads_SSA": np.array(a["ads1_268"]["y"])
})

a = chbzite
chbzite = Material()
chbzite.simple_material("Chabazite (Janchen2006)", "chbzite", Janchen2006_SSA_chbzite, T
    =257, dens=1300)
chbzite.msample = 13e-6 # 12-14 mg # alt: 200e-6 # Janchen2006
chbzite.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_257"]["x"]),
    "wt_ads_SSA": np.array(a["ads_257"]["y"])
})

a = clolite
clolite = Material()
clolite.simple_material("Clinoptilolite (Janchen2006)", "clolite",
    Janchen2006_SSA_clolite, T=257, dens=1300)
clolite.msample = 13e-6 # 12-14 mg # alt: 200e-6 # Janchen2006
clolite.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_257"]["x"]),
    "wt_ads_SSA": np.array(a["ads_257"]["y"])
})

a = mmllote_2
mmllote_2 = Material()
mmllote_2.simple_material("Vacaville (Fanale1974)", "mmllote_2",
    Janchen2006_SSA_mmllote_2, T=257, dens=1300)
mmllote_2.msample = 13e-6 # 12-14 mg # alt: 200e-6 # Janchen2006
mmllote_2.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_257"]["x"]),
    "wt_ads_SSA": np.array(a["ads_257"]["y"])
})
```

```
a = nntrite
nntrite = Material()
nntrite.simple_material("Nontronite (Janchen2006)", "nntrite", Janchen2006_SSA_nntrite,
    T=257, dens=1300)
nntrite.msample = 13e-6 # 12-14 mg # alt: 200e-6 # Janchen2006
nntrite.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_257"]["x"]),
    "wt_ads_SSA": np.array(a["ads_257"]["y"])
})

a = clay_chevrier
clay_chevrier = Material()
clay_chevrier.simple_material("Clay (Chevrier2008)", "clay_chevrier",
    Chevrier2008_SSA_chevrier, T=243, dens=1300)
# clay_chevrier.msample =
clay_chevrier.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_243"]["x"]),
    "wt_ads_SSA": np.array(a["ads_243"]["y"])
})

a = clay_zent
clay_zent = Material()
clay_zent.simple_material("Clay (Zent1997)", "clay_zent", Chevrier2008_SSA_zent, T=240,
    dens=1300)
# clay_zent.msample =
clay_zent.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_240"]["x"]),
    "wt_ads_SSA": np.array(a["ads_240"]["y"])
})

a = HWMK919
HWMK919 = Material()
HWMK919.simple_material("HWMK919 (Jänchen2009)", "HWMK919", Janchen2009_SSA_HWMK919, T
    =253, dens=1300)
HWMK919.msample = 13e-6 # 12-14 mg Janchen2009
HWMK919.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_255"]["x"]),
    "wt_ads_SSA": np.array(a["ads_255"]["y"])
})

a = NG1
NG1 = Material()
NG1.simple_material("NG1 (Jänchen2009)", "NG1", Janchen2009_SSA_NG1, T=253, dens=1300)
NG1.msample = 13e-6 # 12-14 mg Janchen2009
NG1.ads_iso = pd.DataFrame.from_dict({
    "x": np.array(a["ads_293"]["x"]),
    "wt_ads_SSA": np.array(a["ads_293"]["y"])
})
```

```
additional_materials = [vcvlle_1, vcvlle, mmllote_1, chbzite, clolite, mmllote_2,
    nntrite, clay_chevrier, clay_zent, HWMK919, NG1]
```

```
for material in additional_materials:
    material.Psmbar = Pss[material.T]*1e3
    material.ads_iso["P"] = np.array(material.ads_iso["x"])*material.Ps
```

3.2.2.3 Fitting the alpha function to Vacaville basalt

The object of this study is to see which was the data used to obtain the function α from Zent et al. (1993).

The alpha function for a given SSA is presented in Zent and Quinn (1997), with $\gamma = 2.0847 \times 10^{-12}$ kg/m^2/Pa,

$\delta = -2679.8$ K:

$$\frac{\alpha}{SSA} = \frac{\rho_r \gamma}{e^{\delta/T}} (p_{\mathrm{H_2O}})^{0.51} = \frac{\rho_r \gamma \sqrt{n}}{e^{\delta/T}} \left(\frac{k_B T}{m_w}\right)^{0.51}$$

```
def alpha_SSA(n, T, dens_rego):
    m_w = 2.988e-26 # Alternatively MH2O/Avogadro
    gamma = 2.0847e-12 # [kg * m^(-2) * Pa^(-1)]
    delta = -2679.8 # [K]
    return dens_rego * gamma * np.sqrt(n) / np.exp(delta/T) * (Boltzmann*T/m_w)**0.51
```

Obviously, with $SSA = 9800\,\mathrm{m^2/kg}$ for basalt (from Fanale and Cannon (1974)), the expression $\frac{\alpha}{SSA}$–`alpha_SSA` above leads to the function α–`alpha` described previously, which used the parameter $\beta = \gamma \cdot SSA = 2.0847 \times 10^{-12} \cdot 9800 = 2.043 \times 10^{-8}\,\mathrm{Pa^{-1}}$:

```
beta = 2.043e-8
print(beta)
gamma = 2.0847e-12
print(gamma*vcvlle.SSA)
```

```
2.043e-08
2.043006e-08
```

Next fittings to Fanale and Cannon (1974) data are presented. With de density of water at the STP point:

```
dens_H2O
```

```
1000.0
```

The following from FANALE and CANNON (1971) may be used:

```
def Vm2SSA(Vm):
    # n = MH2O / dens_H2O * (1e2)**3
    # return 1e-20 * Vm * Avogadro * SH2O * (1e10)**2 / n * 1e6
    n = MH2O / dens_H2O
    return Vm * Avogadro * SH2O / n
```

```
def SSA2Vm(SSA):
    # n = MH2O / dens_H2O * (1e2)**3
    # return SSA * n / (1e-20 * Avogadro * SH2O * (1e10)**2) * 1/1e6
    n = MH2O / dens_H2O
    return SSA * n / (Avogadro * SH2O )
```

It is easy to see that this comes from assuming that $\frac{SSA}{S_{\mathrm{H_2O}}}$ is the number of molecules covering the surface per kilogram of regolith, and therefore $\frac{SSA}{S_{\mathrm{H_2O}}} \cdot \frac{1}{N_A} \cdot M_{\mathrm{H_2O}}$ is the mass of the monolayer per kilogram of regolith.

Basalt materials are defined below:

```
basalt_materials = [vcvlle_1, vcvlle]
```

```
axFanalealphafit = plt.gca(title="Alpha function and Vacaville basalt")

colors = ['r','b','y','m','c','k'] # color list

for material in basalt_materials + [t1]:

    color = colors[0]
    colors = rotate(colors, -1) # shift colors

    xP = material.ads_iso["x"]*material.Ps # Same as material.ads_iso["P"] for
        additional materials
    xP = xP[~np.isnan(xP)]
    x = material.ads_iso["x"]
    x = x[~np.isnan(x)]
    y = material.ads_iso["wt_ads_SSA"]
    y = y[~np.isnan(y)]

    axFanalealphafit.plot(x, y, "-", color=color, label=material.name + " (data)")

    y_alpha = alpha(dens(MH2O, xP, material.T), material.T, material.dens) / ( material.
        dens * material.SSA )
    axFanalealphafit.plot(x, y_alpha, "-.", color=color, label=material.name + " (alpha,
         Zent1993)")

    y_alpha_SSA = alpha_SSA(dens(MH2O, xP, material.T), material.T, material.dens) / (
        material.dens )
    axFanalealphafit.plot(x, y_alpha_SSA, "--", color=color, label=material.name + " (
        alpha_SSA, Zent1997)")

    pFanalealphafit, eFanalealphafit = optimize.curve_fit(alphafit, xP, y, p0 = [SSAh])
    b = axFanalealphafit.plot(x, alphafit(xP, *pFanalealphafit), ":", color=color, label
         =material.name + " (alpha fit)") # Plot alpha fit
    # print("Alpha fit: ", pFanalealphafit, eFanalealphafit)

    # Missing Vm
    if(material not in materials):
        material.Vm = SSA2Vm(material.SSA)
        material.yp = dens_H2O * material.Vm / material.SSA # Set monolayer saturation
            line for each material
    print(material.name, "yp:", material.yp, "SSA:", material.SSA, "Vm:", material.Vm)
    # axFanalealphafit.axhline(y = material.yp, color=color, alpha=0.5, linestyle='-',
        label = "Monolayer sat. for " + material.name)

axFanalealphafit.axhline(y = yp, color='k', alpha=0.5, linestyle='-', label = "Avg.
    Monolayer sat. for Pommerol2009")

axFanalealphafit.set_yscale('log')
plt.legend(loc="best", bbox_to_anchor=(1, 1))
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Vacaville (Fanale1971) yp: 2.136791160433976e-17 SSA: 5800.0 Vm: 1.2393388730517062e-16
Vacaville (Fanale1974) yp: 2.1367911604339758e-17 SSA: 9800.0 Vm: 2.0940553372252964e-16
Volcanic tuff yp: 2.992700729927007e-07 SSA: 13700.0 Vm: 4.1e-06





Text(0, 0.5, 'wt_ads/SSA')
```

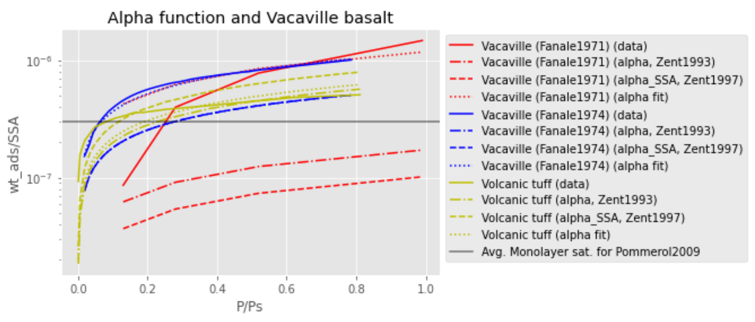


Figure 32: png

Note that `alpha` and `alpha_SSA` are the same function for Vacaville basalt from Fanale and Cannon (1974).

```
xP = vcvlle.ads_iso["x"]*material.Ps # Same as material.ads_iso["P"] for additional
    materials
xP = xP[~np.isnan(xP)]
x = vcvlle.ads_iso["x"]
x = x[~np.isnan(x)]
y = vcvlle.ads_iso["wt_ads_SSA"]
y = y[~np.isnan(y)]

plt.plot(x, y, "-", label=vcvlle.name + " (data)")

y_alpha = alpha(dens(MH2O, xP, vcvlle.T), vcvlle.T, vcvlle.dens) / ( vcvlle.dens *
    vcvlle.SSA )
plt.plot(x, y_alpha, "-.", label=vcvlle.name + " (alpha, Zent1993)")
vcvlle
y_alpha_SSA = alpha_SSA(dens(MH2O, xP, vcvlle.T), vcvlle.T, vcvlle.dens) / ( vcvlle.dens
    )
plt.plot(x, y_alpha_SSA, "--", label=vcvlle.name + " (alpha_SSA, Zent1997)")
plt.axhline(y = yp, color='k', alpha=0.5, linestyle='-', label = "Avg. Monolayer sat.
    for Pommerol2009")

plt.legend(loc="best")
plt.xlabel("P/Ps")
plt.ylabel("wt_ads/SSA")
```

```
Text(0, 0.5, 'wt_ads/SSA')
```

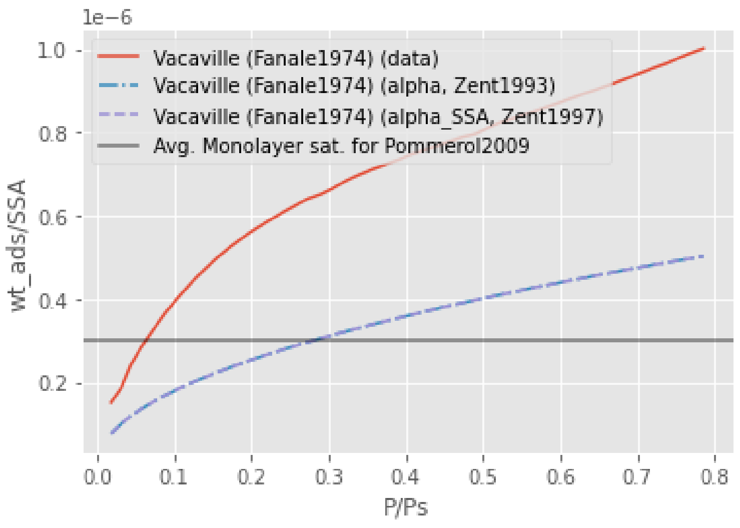


Figure 33: png

3.2.2.3.1 Applying the BET theory to Vacaville basalt The adsorption energy may be found using the BET theory. This value may be used to confirm which data was used.

BET theory only applies to a subset of the data. As in in sec. 3.1.1 for Vacaville basalt:

```
for material in additional_materials:
    material.BET = material.ads_iso[(material.ads_iso["x"] >= BETlim[0]) & (material.
        ads_iso["x"] <= BETlim[1])]
```

Using the same method as previously:

```
axbetfit = plt.gca(title="BET fitting for basalt") # ylim=[0, 1.5e-6]

# material = jsc1
```

```
colors = ['r','b','y','m','c','k']

for material in basalt_materials + [t1]:

    color = colors[0]
    colors = rotate(colors, -1) # shift colors

    xbet = material.BET["x"]

    # Linearize data
    ybet = 1 / (1 / xbet - 1) * 1 / ( material.BET["wt_ads_SSA"] * material.SSA *
        material.msample / dens_H2O )
    axbetfit.plot(xbet, ybet, '.', color=color, label=material.name + " (linearized data
        )")

    K1reg, K0reg, *paramsreg  = stats.linregress(xbet, ybet)
    axbetfit.plot(xbet, K0reg + K1reg*xbet, '--', color=color, label=material.name + " (
        regression)")

    material.K0 = K0reg
    material.K1 = K1reg
    material.C = 1 + K1reg/K0reg

    print(material.name + ":")
    print("K0 regression: " + str(K0reg))
    print("K1 regression: " + str(K1reg))
    print("C Basalt (from regression): " + str(1 + K1reg/K0reg))
    print()

plt.legend(loc="best", bbox_to_anchor=(1, 1))
plt.xlabel("P/Ps")
plt.ylabel("1/v*1/(Ps/P-1) = 1/v*1/(1/x-1)")
```

```
Vacaville (Fanale1971):
K0 regression: 68552760.65670869
K1 regression: -144187165.67642105
C Basalt (from regression): -1.103302103302103

Vacaville (Fanale1974):
K0 regression: 9563094.573455475
K1 regression: 186294962.00535744
C Basalt (from regression): 20.480614833869897

Volcanic tuff:
K0 regression: 812774.5910276473
K1 regression: 363192942.35825235
C Basalt (from regression): 447.85567975131005






Text(0, 0.5, '1/v*1/(Ps/P-1) = 1/v*1/(1/x-1)')
```

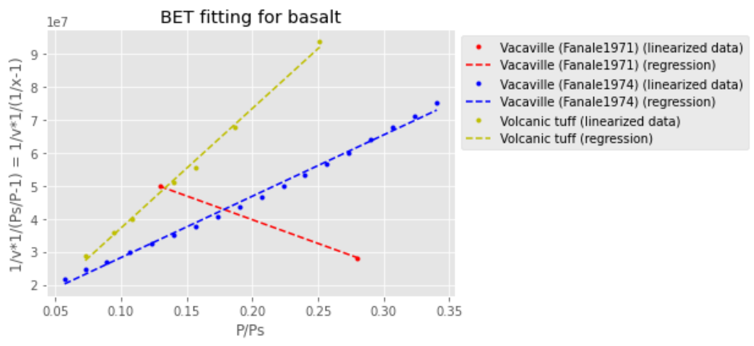


Figure 34: png

Notice that data does not appear to be properly linearised (e.g. negative slope?). Because $m = \rho v$:

$$\frac{1}{v}\frac{x}{1-x} = \underbrace{\frac{1}{v_m c}}_{K0} + \underbrace{\frac{c-1}{v_m c}}_{K1} x \equiv \frac{1}{m}\frac{x}{1-x} = \frac{1}{m_m c} + \frac{c-1}{m_m c} x$$

and with the following:

$$\frac{1}{m}\frac{x}{1-x} = \frac{1}{m_m c} + \frac{c-1}{m_m c} x \equiv \frac{m_{\text{sample}} \cdot SSA}{m}\frac{x}{1-x} = \underbrace{\frac{m_{\text{sample}} \cdot SSA}{m_m} \cdot \frac{1}{c}}_{K0'} + \underbrace{\frac{m_{\text{sample}} \cdot SSA}{m_m} \cdot \frac{c-1}{c}}_{K1'} x$$

Therefore, the following can be stated: $c = 1 + \frac{K1}{K0} = 1 + \frac{K1'}{K0'}$, which mean that from a regression in the `wt_ads_SSA`-vs-`x` space,[9] the adsorption energy can be found. This removes the need of m_{sample}.

```
axbetfit = plt.gca(title="BET fitting for basalt (alternative)") # ylim=[0, 1.5e-6]

# material = jsc1
colors = ['r','b','y','m','c','k']

for material in basalt_materials + [t1]:

    color = colors[0]
    colors = rotate(colors, -1) # shift colors

    xbet = material.BET["x"]

    # Linearize data
    ybet = 1 / (1 / xbet - 1) * 1 / material.BET["wt_ads_SSA"]
    axbetfit.plot(xbet, ybet, '.', color=color, label=material.name + " (linearized data
        )")

    K1preg, K0preg, *paramsPreg  = stats.linregress(xbet, ybet)
    axbetfit.plot(xbet, K0preg + K1preg*xbet, '--', color=color, label=material.name + "
         (regression)")

    print(material.name + ":")
    print("K0p regression: " + str(K0preg))
    print("K1p regression: " + str(K1preg))
    print("C Basalt (from regression): " + str(1 + K1preg/K0preg))
    print()
```

[9]`wt_ads_SSA` is, in fact, $\frac{m}{m_{\text{sample}}}\frac{1}{SSA}$, with $SSA = \frac{\text{true surface}}{m_{\text{sample}}}$.

```
# plt.yscale('log')
plt.legend(loc="best", bbox_to_anchor=(1, 1))
plt.xlabel("P/Ps")
plt.ylabel("SSA*m/msample*1/(1/x-1)")
```

```
Vacaville (Fanale1971):
K0p regression: 2385636.0708534624
K1p regression: -5017713.365539452
C Basalt (from regression): -1.1033021033021027

Vacaville (Fanale1974):
K0p regression: 93718.32681986375
K1p regression: 1825690.6276525026
C Basalt (from regression): 20.480614833869872

Volcanic tuff:
K0p regression: 7658.066095937043
K1p regression: 3422050.3308804105
C Basalt (from regression): 447.8556797513103




Text(0, 0.5, 'SSA*m/msample*1/(1/x-1)')
```

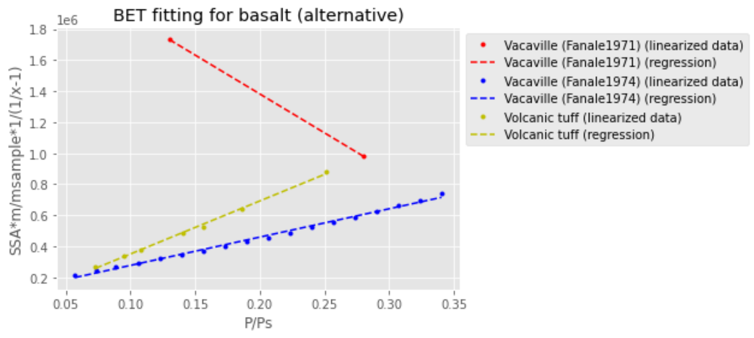


Figure 35: png

Obvously, the value of c is the same as before.

`Qads` may be now computed:

```
for material in basalt_materials:
    material.Qads = R * material.T * np.log(material.C) # The temperature to be used is
        that of the experiments
    print(material.name + ": Qads=" + str(round(material.Qads,2)))
```

```
Vacaville (Fanale1971): Qads=nan
Vacaville (Fanale1974): Qads=6351.65


<ipython-input-87-78bbf2d7a2f4>:2: RuntimeWarning: invalid value encountered in log
  material.Qads = R * material.T * np.log(material.C) # The temperature to be used is
      that of the experiments
```

Data from FANALE and CANNON (1971) appears to have a negative slope and positive intercept point ((i.e. $K0'$ and $K1'$), which means that c—dependant on $\frac{K1'}{K0'}$—is negative and therefore $Q_{\text{ads}} = R \cdot T \ln c$ cannot be computed.

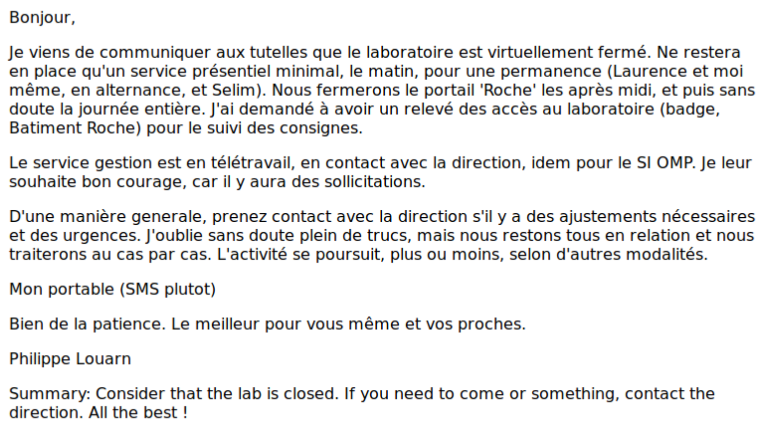
4 Logistics of the internship

4.1 Impact of COVID-19 on the course of the intership

IRAP was virtually closed on 2020-03-16 due to COVID-19. The message in fig. 36 was the last of a series of messages the researchers received during the weekend. The virus was spreading fast in Italy, Catalonia, and Spain, and it was a matter of time before it arrived in France, Occitanie, and eventually Toulouse. As an intern from ISAE-SUPAERO, it was obvious that I would have to work from home, but I still had faith I would have the chance to work at IRAP on Monday 2020-03-16 and a couple of days more. After hearing that there were some potential cases in ISAE-SUPAERO, I decided not to go to IRAP (telecommuting). This happened after three weeks after the start of the internship, two after I started working on the project *per se*.

This made accessing resources located at IRAP a little bit more challenging, as it can be seen in sec. 4.2.1. Guidance from the tutor was limited to what emailing and telecommunication tools could offer. The means used were:

- Emails
- Mobile calls
- Meetings through RENAvisio


Bonjour,

Je viens de communiquer aux tutelles que le laboratoire est virtuellement fermé. Ne restera en place qu'un service présentiel minimal, le matin, pour une permanence (Laurence et moi même, en alternance, et Selim). Nous fermerons le portail 'Roche' les après midi, et puis sans doute la journée entière. J'ai demandé à avoir un relevé des accès au laboratoire (badge, Batiment Roche) pour le suivi des consignes.

Le service gestion est en télétravail, en contact avec la direction, idem pour le SI OMP. Je leur souhaite bon courage, car il y aura des sollicitations.

D'une manière generale, prenez contact avec la direction s'il y a des ajustements nécessaires et des urgences. J'oublie sans doute plein de trucs, mais nous restons tous en relation et nous traiterons au cas par cas. L'activité se poursuit, plus ou moins, selon d'autres modalités.

Mon portable (SMS plutot)

Bien de la patience. Le meilleur pour vous même et vos proches.

Philippe Louarn

Summary: Consider that the lab is closed. If you need to come or something, contact the direction. All the best !

Figure 36: The closure of IRAP due to COVID-19

4.2 Available machines at CNRS

The GCM models are available in the following servers/supercomputers of the CNRS:

- *Hyperion*, located at IRAP, has the subsurface model installed along the compatible GCM version. **The subsurface model developed by P.-Y. Meslin is not compatible with the newest LMDZ GCM versions, and one of the goals of the internship is to make it available for newer versions**. LMDZ can only be run sequentally.
- *Hyperion2*, located at IRAP, has the newest LMDZ GCM verison. Only the effects of water vapor[10] are taken into account, but the files that allow simulations of CH_3 and Rn in Hyperion should be easy to include in the version found in Hyperion2. LMDZ can only be run sequentally.

[10] Current LMDZ GCM versions do not simulate the presence of ice water, **and do not take into account gases other than water vapor(? TODO).**

- *Titan*, found at LMD (Paris), has the newest LMDZ GCM version as it is the system that LMD uses. Only the effects of water vapor are taken into account. LMDZ can be run sequentally or in parallel, but the subsurface model is sequential (see **TODO**).

4.2.1 Accessing Hyperion and Hyperion2 from outside IRAP

If a user wants to access Hyperion or Hyperion2 (and maybe Titan) from outside IRAP, a gateway has to be used. Herebelow, the steps to connect to the any remote machine are presented:

1. From the local machine, run `ssh -X aprat-gasull@gw.irap.omp.eu`
2. From the gateway machine, run the following:
   - `ssh -X aprat-gasull@hyperion.irap.omp.eu` to access Hyperion
   - `ssh -X aprat-gasull@hyperion2.irap.omp.eu` to access Hyperion2
   - **TODO:** Add Titan procedure

4.2.2 Moving files between machines

4.2.2.1 From a remote machine to local One cannot access e.g. Hyperion2 from outside IRAP. The fastest way is to use the following:

1. Open a terminal, run `ssh <user>@gw.irap.omp.eu -L <localPort>:hyperion2.irap.omp.eu:22` (e.g. `ssh aprat-gasull@gw.irap.omp.eu -L 4567:hyperion2.irap.omp.eu:22`), and log in to the gateway machine.
2. Open another terminal, run one of the following, and log in to Hyperion2.
   - `scp -P <localPort> <user>@localhost:/path/to/file /path/to/destination` (e.g. `scp -P 4567 aprat-gasull@localhost:/home/STAGIAIRES/aprat-gasull/Models/model.tar.gz ~/Desktop/`)
   - `rsync -azv -e "ssh -p <localPort>"<user>@localhost:/path/to/file /path/to/destination` (e.g `rsync -azv -e "ssh -p 4567"aprat-gasull@localhost:/home/STAGIAIRES/aprat-gasull/Models/model.tar.gz ~/Desktop/`)

Alternatively, because the files need to be copied to the gateway machine before being accessible from the exterior one can copy a file stored in Hyperion2 to local using the following lines. *This is only to be done if the user is allowed to copy files in the gateway machine, which is not the case for the gateway machine.*

1. From the gateway machine (run `ssh -X aprat-gasull@gw.irap.omp.eu`), execute one of the following to copy the files from Hyperion2 to the gateway machine:
   - `scp aprat-gasull@hyperion2.irap.omp.eu:/path/to/your/file/file_name .` for a single or a low quantity of files
   - `rsync -aP aprat-gasull@hyperion2.irap.omp.eu:/path/to/your/file/file_name .` for multiple and heavy files
2. Run `exit` from the gateway machine to close the connection to Hyperion2
3. Run locally either one of this commands to retrieve the files from the gateway machine:

- `scp aprat-gasull@gw.irap.omp.eu:/path/to/your/file/file_name .` for a single or a low quantity of files
- `rsync -aP aprat-gasull@gw.irap.omp.eu:/path/to/your/file/file_name .` for multiple and heavy files

4.2.2.2 From local to a remote machine The procedure to copy a local file to Hyperion2 is exactly the inverse to the one presented above in sec. 4.2.2.1:

1. Open a terminal, run `ssh <user>@gw.irap.omp.eu -L <localPort>:hyperion2.irap.omp.eu:22` (e.g. `ssh aprat-gasull@gw.irap.omp.eu -L 4567:hyperion2.irap.omp.eu:22`), and log in to the gateway machine.
2. Open another terminal, run one of the following, and log in to Hyperion2.
   - `scp -P <localPort> /path/to/file <user>@localhost:/path/to/destination` (e.g. `scp -P 4567 /home/aeronau/Desktop/file.txt aprat-gasull@localhost:/home/STAGIAIRES/aprat-gasull/Models/`)
   - `rsync -azv -e "ssh -p <localPort>"/path/to/file <user>@localhost:/path/to/destination` (e.g `rsync -azv -e "ssh -p 4567"/home/aeronau/Desktop/file.txt aprat-gasull@localhost:/home/STAGIAIRES/aprat-gasull/Models/`)

Alternatively, *if the gateway machine allows the storage of files (see sec. 4.2.2.1)*:

1. Run locally either one of this commands to upload the files to the gateway machine:
   - `scp . aprat-gasull@gw.irap.omp.eu:/path/to/your/file/file_name` for a single or a low quantity of files
   - `rsync . -aP aprat-gasull@gw.irap.omp.eu:/path/to/your/file/file_name` for multiple and heavy files
2. From the gateway machine (run `ssh -X aprat-gasull@gw.irap.omp.eu`), execute one of the following to copy the files from the gateway machine to Hyperion2:
   - `scp . aprat-gasull@hyperion2.irap.omp.eu:/path/to/your/file/file_name` for a single or a low quantity of files
   - `rsync -aP . aprat-gasull@hyperion2.irap.omp.eu:/path/to/your/file/file_name` for multiple and heavy files

5 Integrating the fit in the subsurface model

5.1 The physics behind the subsurface model—stability and exchange of subsurface ice on Mars

A good introduction to the physics is provided by Schorghofer (2005), whose abstract is the following:

> We seek a better understanding of the distribution of subsurface ice on Mars, based on the physical processes governing the exchange of vapor between the atmosphere and the subsurface. Ground ice is expected down to $\sim 49°$ latitude and lower latitudes at poleward facing slopes. The diffusivity of the regolith also leads to seasonal accumulation of atmospherically derived frost at latitudes poleward of

> $\sim 30°$. The burial depths and zonally averaged boundaries of subsurface ice observed from neutron emission are consistent with model predictions for ground ice in equilibrium with the observed abundance of atmospheric water vapor. Longitudinal variations in ice distribution are due mainly to thermal inertia and are more pronounced in the observations than in the model. These relations support the notion that the ground ice has at least partially adjusted to the atmospheric water vapor content or is atmospherically derived. Changes in albedo can rapidly alter the equilibrium depth to the ice, creating sources or sinks of atmospheric H_2O while the ground ice is continuously evolving toward a changing equilibrium. At steady state humidity and temperature oscillations, the net flux of vapor is uninhibited by adsorption. The occurrence of temporary frost is characterized by the isosteric enthalpy of adsorption.

The following sections are extracts of Schorghofer (2005).

5.1.1 Subsurface ice in diffusive contact with atmospheric vapor

The Martian atmosphere has low absolute humidity, but due to its low temperature, it is often close to saturation. When water vapor diffuses into the ground, the vapor can freeze out as frost, and ground ice can develop even when in diffusive contact with a *dry* atmosphere. The basic physical processes responsible for ground ice formation are vapor diffusion and deposition. These physical principles alone predict ground ice covered by a dry layer of regolith, and some have argued that vapor exchange between the atmosphere and the subsurface accounts for the geographic distribution of high-latitude ground ice on Mars.

There are two possible end-member views of subsurface water ice on Mars, which are distinguished by the rate of vapor diffusion between the atmosphere and the subsurface:

- An atmosphere which has no contact with the subsurface ice buried under an impermeable layer of dry material. Low diffusivities would isolate the subsurface from changes in atmospheric humidity over obliquity variations or other climate changes. **The subsurface model uses the atmospheric data provided by the GCM to model subsurface changes**.
- The ground ice freely exchanges vapor with the present atmosphere, and is protected by a dry layer of regolith only from large temperature excursions. Ground ice and atmospheric vapor eventually reach an equilibrium. **The goal of the project is to model these exchanges, as it is supported by a number of scientists**.

Taking into account the porosity of the subsurface, the average change per Mars year is about the equivalent of the atmospheric water column abundance. If there is substantial redistribution of subsurface ice by diffusion over obliquity cycles, the vapor exchange with the ground ice could have a major impact on the annual atmospheric water budget.

5.1.2 Concepts of *stability* and *exchange*

5.1.2.1 Vapor diffusion and ice formation Frost can form directly from vapor by deposition, without a transient liquid phase. The pertinent pressure is the partial pressure of H_2O which can be lower than the triple point pressure, irrespective of the much heavier atmosphere. On Mars, unlike on Earth, the frost point temperature (typically $\sim$ 198 K) differs significantly from the melting point ($\sim$ 273 K).

The frost point temperature is determined by the water vapor content, according to the phase diagram of H_2O (see eq. 6), where sv is short for *solid vapor*, $p_t = 611$ Pa and $T_t = 273.16$ K are the triple point pressure and temperature respectively, $H_{\text{subl.}} = 51058\ \text{J mol}^{-1}$ the molar heat of sublimation (depends weakly on temperature), $R = 8.314\ \text{J K}^{-1}\ \text{mol}^{-1}$:

$$p_{sv}(T) = p_t \exp\left(-\frac{H_{\text{subl.}}}{R}\left(\frac{1}{T} - \frac{1}{T_t}\right)\right) \tag{6}$$

Diffusion can be written in terms of mass flux (see eq. 7), which can be simplified to eq. 8 if ρ_{air} is assumed constant with depth, and thermodiffusion and barodiffusion ratios—k_T and k_p, respectively—negligible. D is the diffusion coefficient while ρ_v is the mass density of water vapor. z is the vertical coordinate into the ground.

$$J = -D\rho_{\text{air}}\left(\frac{\partial}{\partial z}\frac{\rho_v}{\rho_{\text{air}}} + \frac{k_T}{T}\frac{\partial T}{\partial z} + \frac{k_p}{p_{\text{air}}}\frac{\partial p_{\text{air}}}{\partial z}\right) \tag{7}$$

$$J = -D\frac{\partial \rho_v}{\partial z} \tag{8}$$

The last expression shows that vapor diffuses into regions of low vapor density, irrespective of the relative humidity or saturation of the air. For example, the vapor density of warm, unsaturated air may be higher than that of cold, saturated air. In this case, vapor will flow toward the saturated region and produce frost.

Mass transfer of H_2O may also be caused by advection. A difference in air pressure causes the gas to move as a whole. Differences in air (CO_2) pressure are caused by thermal expansion and contraction or surface winds. At least on a seasonal basis the contribution of thermal expansion to the vapor transport is small, because the speed of diffusion is high. Advection is potentially more important at depths comparable to diurnal skin depths[11].

Assumptions appear least certain in regions of very low thermal inertia, especially for large temperature oscillations.

It is assumed that the partial pressure of H_2O does not change diurnally unless the temperature falsl blow the frost point, in which case the partial pressure equals the saturation vapor pressure, $p_{\text{surf}} = \min(p_0, p_{sv}(T_{\text{surf}}))$, with p_0 being the partial pressure estimated from afternoon observations and assumed to be constant throughout the day.

As water vapor diffuses through the subsurface, it can undergo phase transitions to free ice or adsorbed H_2O.

For a dry regolith of density ρ, thermal conductivity k, and thermal inertia $I = \sqrt{k\rho c}$, and for a temperature varying sinusoidally with an annual period, it can be seen that each year a layer of frost is transported deep into the subsurface, where it contributes to a growing ground ice reservoir. Note that the partial pressure at the surface never exceeds p_0 and, at times when the atmosphere is saturated, it is lower than p_0.

[11] From Titus and Cushing (2012), the thermal skin depth is a measure of how far either a diurnal or annual surface temperature cycle penetrates the regolith.

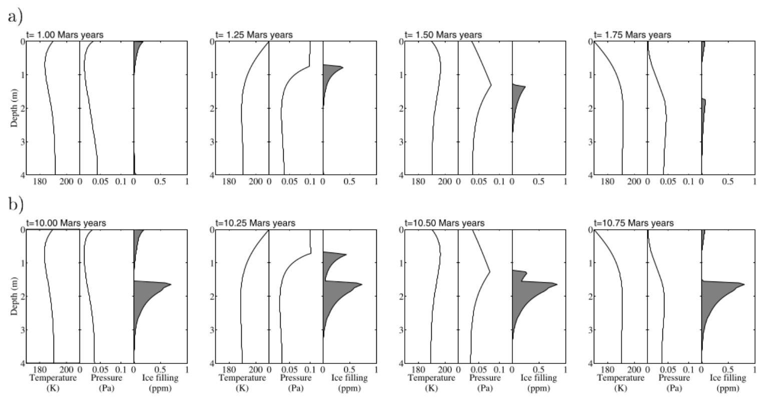


Figure 1. The downward migration of frost over two annual temperature cycles according to a one-dimensional model which simulates the conduction of heat and the diffusion, sublimation, and deposition of water vapor. Each panel shows instantaneous vertical profiles of temperature, partial pressure, and the fraction of pore space filled with ice. (a) The second year and (b) the tenth year after starting with an ice-free regolith. A layer of frost migrates inward during the warming part of the cycle and begins to fill the subsurface with ice. A sinusoidal surface temperature and constant daytime partial pressure are assumed in this model calculation. Model parameters: p_0 = 0.1 Pa, $D = 0.1(T/200\ \mathrm{K})^{3/2}\ \mathrm{cm^2 s^{-1}}$, $I = 280\ \mathrm{J\ m^{-2} K^{-1} s^{-1/2}}$, $\rho c = 1.28 \times 10^6\ \mathrm{J\ m^{-3} K^{-1}}$.

Figure 37: Migration of frost as computed in the model presented in Schorghofer (2005)

TBasic physical processes lead to a dry layer over an icy layer. The top layer is dry because the large temperature oscillations lead to high vapor densities if any ice is present. The layer *protects* the underlying ice not by reducing vapor transport, but by attenuating temperature oscillations, which would cause a large outward flux of vapor.

5.1.2.2 Net diffusion and permanent ice stability The mean annual vapor flux is given in terms of the gradient of the mean annual vapor density from averaging the equation of the mass flux (see eq. 9). Note that the correlation of D with $\frac{\partial \rho_v}{\partial z}$ is assumed to be small.

$$\langle J \rangle = -\left\langle D \frac{\partial \rho_v}{\partial z} \right\rangle \approx -\langle D \rangle \left\langle \frac{\partial \rho_v}{\partial z} \right\rangle = -\langle D \rangle \frac{\partial \langle \rho_v \rangle}{\partial z} \tag{9}$$

In any depth interval without net accumulation of H_2O, the averaged flux $\langle J \rangle$ must be constant with depth, due to mass conservation, and therefore is constant between the surface and a growing or receding ice reservoir. The mean vapor density $\langle \rho_v \rangle$ must decrease or increase linearly—respectively—with depth, even with in the presence of temporary frost or adsorbate at intermediate depths over an annual cycle. The gradient is given by the difference of vapor densities at the boundaries.

Concerning the amount of subsurface ice as a function of time, one can see that, after a long time, adsorption has a negligible impact on the amount of accumulated pore ice, but the vertical vapor density profile and the amount of periodically exchanged H_2O—*breathing*—are strongly affected by adsorption. The density of adsorbed H_2O is typically large compared to the density of vapor, and adsorption can therefore have a strong influence on the mass balance. However, adsorption has no influence on net diffusion in a stationary environment as derived from the mass balance (see eq. 10, where ϕ is the porosity of the regolith and $\bar{\rho}_a$ the density of adsorbed water):

$$\frac{\partial}{\partial t}\left(\phi \rho_v + \bar{\rho}_a\right) = \frac{\partial}{\partial z}\left(\phi D \frac{\partial}{\partial z} \rho_v\right) \tag{10}$$

Considering that the temperature is independent of time t and depth z, the equation eq. 10 becomes eq. 11 which states that for stationary diffusion between two prescribed boundary conditions (the surface and the ground ice), the mean annual flux is given purely by the difference between the mean vapor densities on the surface and the air in immediate contact with any subsurface frost, not involving $\bar{\rho}_a$ in any case. The factor $\frac{D}{1+\frac{RT}{18\phi}\frac{\partial\bar{\rho}_a}{\partial p}}$ is the rescaled diffusion coefficient. Adsorption changes the instantaneous vertical profile $\rho_v\left(z\right)$ but not the mean profile $\langle\rho_v\left(z\right)\rangle$.

$$\left(1+\frac{RT}{18\phi}\frac{\partial\bar{\rho}_a}{\partial p}\right)\frac{\partial p}{\partial t}=D\frac{\partial^2 p}{\partial z^2} \tag{11}$$

In fig. 38, the results of the model presented in Schorghofer (2005) are shown. In the left-most panel, the breaks in slope in the temperature profiles are caused by the thermal conductivity of ice. The right-most panel shows that the vapor density changes indeed linearly with depth, as required by mass conservation, although temporary frost and a temperature dependent diffusion coefficient are taken into account in these model calculations.

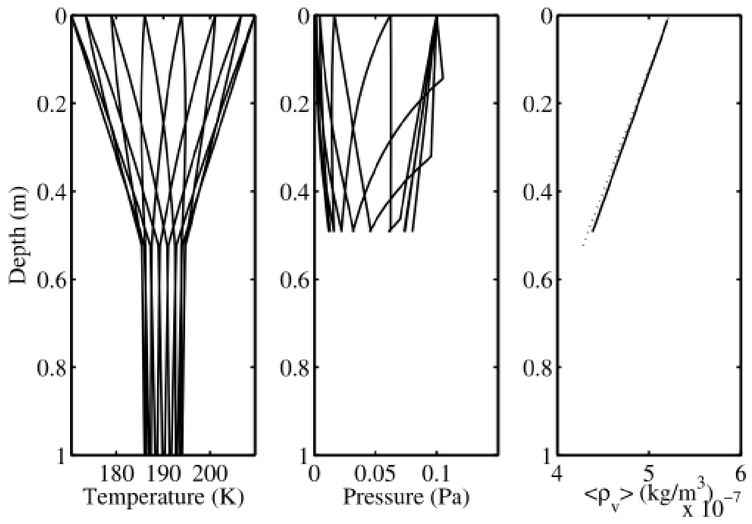


Figure 3. Temperature, partial pressure, and mean vapor density when the ice table is lower (0.5 m) than at equilibrium. A sinusoidal surface temperature with annual period and constant daytime partial pressure are assumed in this model calculation. The vapor density decreases linearly with depth, and the gradient in vapor density drives the ice level to its equilibrium. The dotted lines are estimates of the vapor density using boundary values only. Model parameters: I = 280 J m^{-2}K^{-1}s$^{-1/2}$, ρc = 1280000 J m^{-3}K^{-1}, p_0 = 0.1 Pa, D = 1(T/200 K)$^{3/2}$ cm^{2}s^{-1}, ϕ = 0.3.

Figure 38: Results of the diffusion model presented in Schorghofer (2005)

From eq. 9, the following in eq. 12 can be derived, where $\Delta\left\langle\rho_v\right\rangle$ i is the difference between the mean vapor density at the ice and at the surface, and Δz is the depth of perennial ice below the surface. At the ice, the air is saturated, because it is in contact with the ice. If no frost is present, the vapor pressure can only be lower than ρ_{sv}:

$$\langle J\rangle\approx-\langle D\rangle\frac{\Delta\left\langle\rho_v\right\rangle}{\Delta z} \tag{12}$$

This line of reasoning leads to a simple and widely valid condition for ground ice stability. The vapor flux between the surface and any point at depth z_0 where frost is present is proportional to the vapor density difference between these two points, $\langle J\rangle \propto \left\langle \frac{p_{\text{surf}}}{T_{\text{surf}}}\right\rangle - \left\langle \frac{p_{sv}(z_0)}{T(z_0)}\right\rangle$. Ice grows from year to year when eq. 13 is satisfied and is unstable when this inequality is reversed. Equilibrium is reached when the two terms are equal, and the vapor densities balance.

$$\left\langle \frac{p\,(\text{surface})}{T\,(\text{surface})}\right\rangle > \left\langle \frac{p_{sv}\,(\text{at depth})}{T\,(\text{at depth})}\right\rangle \tag{13}$$

A very cold, fully saturated atmosphere can draw vapor out of the regolith, because the flux is determined by differences in vapor density irrespective of the relative humidity of the air. When the surface is colder than frost in the ground, the absolute atmospheric humidity is lower than the saturated vapor near the ice and vapor will diffuse outward. Since the humidity of a saturated atmosphere cannot increase, the excess water will form surface frost, fogs, or clouds.

The depth to the equilibrium ice table is set by the temperature profile and the annual mean vapor density at the surface. The details of the diffusion and adsorption processes do not influence the long-term H_2O transport and properties of the regolith enter only insofar as they change the thermal profile.

5.1.2.3 Temporary frost: isosteric enthalpy If the stability condition is satisfied instantaneously, there must be a downward flux of vapor at some depth. For any realistic diffusivity, this will soon lead to the accumulation of frost or adsorbate in the ground.

Adsorption typically decreases rapidly with temperature (at constant pressure) and it typically increases with vapor pressure (at constant temperature). The adsorption "isotherm" describes the amount of adsorbed water as a function of pressure, at constant temperature.

Consider a fixed volume of vapor in contact with an adsorbent. If the amount of available water vapor is large compared to the change of adsorbed H_2O mass, then the partial pressure will remain constant with temperature and adsorption moves along isobars. If, on the other hand, the amount of vapor is small, then little adsorption or desorption can take place and pressures and temperatures follow isosteres.Without additional sources or sinks of H_2O, the pressure cannot change faster with temperature than the slope of the isostere, because this would require simultaneous desorption and decrease in partial pressure. Adsorption is self-limiting because reduction of the partial pressure limits the ability of the soil to adsorb.

If the isosteric enthalpies are significantly smaller than the enthalpy of sublimation, the frost point temperature is reduced little compared to a regolith without adsorption. If the enthalpy of adsorption is higher than for sublimation, free ice can form on warming from water released by desorption.

This principle can be applied to adsorption in the ground, although diffusion contributes to the H_2O balance. Beneath the breathing skin depth, the partial pressure indeed follows isosteres.

5.1.3 Conclusions

These conclusions have been drawn in Schorghofer (2005):

- Diffusion and deposition of water vapor produce subsurface frost beneath a dry layer of porous regolith.
- The soil layer protects underlying ice not primarily by reducing vapor transport, but by attenuating temperature amplitudes. When H_2O transport is predominantly due to gradients in vapor density, it is possible to derive a simple condition for permanent stability of ground ice, shown in eq. 13.

- The equilibrium depth to the ice table is set only by the thermal profile and the mean vapor density on the surface. Permanent ice is typically buried at depths in between the diurnal and seasonal thermal skin depths. Adsorption of H_2O does not affect the net (mean) vapor transfer beyond a transient time period.
- The relations between present-day humidities and the ground ice distribution indicates that the diffusivity of the regolith is sufficiently large to allow the subsurface ice to respond to the atmospheric humidity. Atmospheric water vapor appears to be the major controlling factor for the observed present-day ground ice distribution.
- Since the ratio of ice density to vapor density is enormous on Mars, any redistribution of ice requires rapid transfer of water vapor between the atmosphere and the regolith. This exchange could be a sizable contribution to the atmospheric water budget, and is potentially observable today.
- The depth to the ice table is sensitive to albedo. Given the ongoing albedo changes on Mars, it is unlikely the ice table has enough time to fully equilibrate.
- Seasonal subsurface frost is another consequence of a diffusive regolith. Adsorption isosteres determine the partial pressure in the subsurface. Unless the isosteric enthalpy of adsorption is larger than or comparable to the enthalpy of sublimation, adsorption has little capacity to suppress the formation of free ice.

5.2 Understanding the code used to model the subsurface

5.2.1 Derivation of the formulae used

The basic equations that the model implements are the following:

$$\begin{cases} \frac{\partial}{\partial t}\left(\varepsilon \hat{n}_{\text{vap}}\right) = \frac{\partial}{\partial z}\left(D_b \frac{\partial}{\partial z}\hat{n}_{\text{vap}}\right) - K_a \hat{n}_{\text{vap}} + K_d \hat{n}_{\text{ads}} \\ \frac{\partial}{\partial t}\left(\hat{n}_{\text{ads}}\right) = K_a \hat{n}_{\text{vap}} - K_d \hat{n}_{\text{ads}} \end{cases} \tag{14}$$

From the system of equations above, and assuming $\hat{n}_{\text{vap}} = n_1$ and $\hat{n}_{\text{ads}} = n_2$, these can be written as shown in eq. 15 and eq. 16:

$$\frac{\partial}{\partial t}\left(\varepsilon n_1\right) = \frac{\partial}{\partial z}\left(D_b \frac{\partial}{\partial z} n_1\right) - K_a n_1 + K_d n_2 \tag{15}$$

$$\frac{\partial}{\partial t}\left(n_2\right) = K_a n_1 - K_d n_2 \tag{16}$$

Being Δt the timestep between instant $t - dt$ and t, eq. 16 can be discretized to eq. 17:

$$\frac{n_2^t - n_2^{t-dt}}{\Delta t} = K_a^t n_1^t - K_d^t n_2^t \tag{17}$$

By isolating n_2^t from eq. 17, eq. 18 is found:

$$n_2^t = \frac{\Delta t K_a^t n_1^t + n_2^{t-dt}}{1 + \Delta t K_d} \tag{18}$$

5.2.1.1 For the standard case (no saturation, and not including the boundaries) eq. 15 can be discretized as shown below by using eq. 18. Note the use of past values for ε.

$$\frac{\varepsilon^{t-dt}_{k+\frac{1}{2}} n^t_{1,k+\frac{1}{2}} - \varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} =$$
$$\frac{1}{z_{k+1}-z_k}\left(D^t_{b,k+1}\frac{n^t_{1,k+\frac{3}{2}} - n^t_{1,k+\frac{1}{2}}}{z_{k+\frac{3}{2}} - z_{k+\frac{1}{2}}} - D^t_{b,k}\frac{n^t_{1,k+\frac{1}{2}} - n^t_{1,k-\frac{1}{2}}}{z_{k+\frac{1}{2}} - z_{k-\frac{1}{2}}}\right)$$
$$-K^t_a n^t_{1,k+\frac{1}{2}} + \left(\frac{K^t_d}{1+\Delta t K^t_d}\right)\left(\Delta t K^t_a n^t_{1,k+\frac{1}{2}} + n^{t-dt}_{2,k+\frac{1}{2}}\right)$$

This can be grouped as follows:

$$\frac{\varepsilon^{t-dt}_{k+\frac{1}{2}} n^t_{1,k+\frac{1}{2}} - \varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} =$$
$$\frac{1}{z_{k+1}-z_k}\left(D^t_{b,k+1}\frac{n^t_{1,k+\frac{3}{2}} - n^t_{1,k+\frac{1}{2}}}{z_{k+\frac{3}{2}} - z_{k+\frac{1}{2}}} - D^t_{b,k}\frac{n^t_{1,k+\frac{1}{2}} - n^t_{1,k-\frac{1}{2}}}{z_{k+\frac{1}{2}} - z_{k-\frac{1}{2}}}\right)$$
$$+\underbrace{\left(\frac{\Delta t K^t_d K^t_a}{1+\Delta t K^t_d} - K^t_a\right)}_{-G^t = -\frac{K^t_a}{1+\Delta t K^t_d}} n^t_{1,k+\frac{1}{2}} + \underbrace{\frac{K^t_d}{1+\Delta t K^t_d}}_{H^t} n^{t-dt}_{2,k+\frac{1}{2}}$$

From the expression above, the following is found:

$$\frac{\varepsilon^{t-dt}_{k+\frac{1}{2}} n^t_{1,k+\frac{1}{2}} - \varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} =$$
$$\frac{1}{z_{k+1}-z_k}\left[D^t_{b,k+1} d_{k+1}\left(n^t_{1,k+\frac{3}{2}} - n^t_{1,k+\frac{1}{2}}\right) - D^t_{b,k} d_k\left(n^t_{1,k+\frac{1}{2}} - n^t_{1,k-\frac{1}{2}}\right)\right]$$
$$-G^t n^t_{1,k+\frac{1}{2}} + H^t n^{t-dt}_{2,k+\frac{1}{2}}$$

Note that in order to calculate $n_{1,k+\frac{3}{2}}$, values of $n_{1,k-\frac{1}{2}}$ and $n_{1,k+\frac{1}{2}}$ are needed. This system of equations can be re-arranged for $k+1$ to be of the form:

$$\boxed{n^t_{1,k+\frac{1}{2}} = \alpha^t_{k+1} n^t_{1,k+\frac{3}{2}} + \beta^t_{k+1}} \tag{19}$$

Note that α^t_{k+1} and β^t_{k+1} take into account the *effect* of $n_{1,k-\frac{1}{2}}$, which is calculated using the same expression in eq. 19, but evaluated at k, as shown in eq. 20:

$$n^t_{1,k-\frac{1}{2}} = \alpha^t_k n^t_{1,k+\frac{1}{2}} + \beta^t_k \tag{20}$$

By substituting this last equation into the expresion found previously:

$$\frac{\varepsilon_{k+\frac{1}{2}}^{t-dt} n_{1,k+\frac{1}{2}}^{t} - \varepsilon_{k+\frac{1}{2}}^{t-2dt} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta t} =$$

$$\frac{1}{z_{k+1}-z_k}\left\{ D_{b,k+1}^t d_{k+1}\left(n_{1,k+\frac{3}{2}}^t - n_{1,k+\frac{1}{2}}^t\right) - D_{b,k}^t d_k \left[n_{1,k+\frac{1}{2}}^t - \underbrace{\left(\alpha_k^t n_{1,k+\frac{1}{2}}^t + \beta_k^t\right)}_{n_{1,k-\frac{1}{2}}^t} \right] \right\}$$

$$-G^t n_{1,k+\frac{1}{2}}^t + H^t n_{2,k+\frac{1}{2}}^{t-dt}$$

With $\Delta z = \Delta z_{k+\frac{1}{2}} = z_{k+1} - z_k$, the following is obtained:

$$\underbrace{\left[\varepsilon_{k+\frac{1}{2}}^{t-dt}\frac{\Delta z}{\Delta t} + G^t\Delta z + D_{b,k+1}^t d_{k+1} + D_{b,k}^t d_k\left(1-\alpha_k^t\right)\right]}_{\Delta_{k+\frac{1}{2}}^t} n_{1,k+\frac{1}{2}}^t =$$

$$D_{b,k+1}^t d_{k+1} n_{1,k+\frac{3}{2}}^t + D_{b,k}^t d_k \beta_k^t + \Delta z H^t n_{2,k+\frac{1}{2}}^{t-dt} + \varepsilon_{k+\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,k+\frac{1}{2}}^{t-dt}$$

And re-arranging the equation in the same form of eq. 19:

$$n_{1,k+\frac{1}{2}}^t = \underbrace{\frac{D_{b,k+1}^t d_{k+1}}{\Delta_{k+\frac{1}{2}}^t}}_{\alpha_{k+1}^t} n_{1,k+\frac{3}{2}}^t + \underbrace{\frac{D_{b,k}^t d_k \beta_k^t + \Delta z H^t n_{2,k+\frac{1}{2}}^{t-dt} + \varepsilon_{k+\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta_{k+\frac{1}{2}}^t}}_{\beta_{k+1}^t} \tag{21}$$

Therefore, in this case, with $C_{k+\frac{1}{2}} = \frac{\Delta z}{\Delta t}$, the expressions to be used are the following:

Table 1: Variables for the standard case

Variable	Expression
$\Delta_{k+\frac{1}{2}}^t$	$\varepsilon_{k+\frac{1}{2}}^{t-dt} C_{k+\frac{1}{2}} + G^t\Delta z + D_{b,k+1}^t d_{k+1} + D_{b,k}^t d_k\left(1-\alpha_k^t\right)$
α_{k+1}^t	$\frac{D_{b,k+1} d_{k+1}}{\Delta_{k+\frac{1}{2}}^t}$
β_{k+1}^t	$\frac{D_{b,k}^t d_k \beta_k^t + \Delta z H^t n_{2,k+\frac{1}{2}}^{t-dt} + \varepsilon_{k+\frac{1}{2}}^{t-2dt} C_{k+\frac{1}{2}} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta_{k+\frac{1}{2}}^t}$

5.2.1.1.1 Initialization with mass exchange with the atmosphere To initialize the subsurface ($t=0$), a flux between the soil/surface (regolith) and the atmosphere can be introduced and considered constant along the domain. The non-dimensional flux F—arising from convective mass transfers and derived from $f = \rho C_d\left(q - q_\infty\right)$—is written as shown below, with n being the mass fraction of water—mass of water per mass of carbon dioxide[^mh2omco2]—in, n the mass of per volume of regolith in, and $\varepsilon = \varepsilon\left(0\right)$ the porosity at $t = 0$:

[^mh2omco2]: Note that the martian atmosphere is majoritarily composed of ($\sim 95\%$).

$$\begin{aligned}
F &= \varepsilon \rho_{\mathsf{atm}} C_d \left(q_{\mathsf{soil}} - q_\infty\right) \\
&= \varepsilon \rho_{\mathsf{atm}} C_d \left(q_{\mathsf{vap}} - q_{\mathsf{atm}}\right) \\
&= \varepsilon\left(0\right) \rho_{\mathsf{atm}} C_d V \left(\mathtt{n}_{\mathsf{vap}} - \mathtt{n}_{\mathsf{atm}}\right) \\
&= \varepsilon\left(0\right) \rho_{\mathsf{atm}} C_d V \left(\mathtt{n}^t_{1,k+\frac{1}{2}} - \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}}\right) \\
F_k &= \varepsilon_{k+\frac{1}{2}}\left(0\right) \rho_{\mathsf{atm}} C_d V \left(\frac{n^t_{1,k+\frac{1}{2}}}{\rho_{\mathsf{vap},k+\frac{1}{2}}} - \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}}\right)
\end{aligned}$$

Note that$\mathtt{n}_{\mathsf{atm}}$ is a boundary condition. By introducing this flux into the main formula (see eq. 5.2.1.1), the following is obtained:

$$\frac{\varepsilon^{t-dt}_{k+\frac{1}{2}} n^t_{1,k+\frac{1}{2}} - \varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} =$$

$$\frac{1}{z_{k+1}-z_k}\left[D^t_{b,k+1} d_{k+1}\left(n^t_{1,k+\frac{3}{2}} - n^t_{1,k+\frac{1}{2}}\right) - \varepsilon_{k+\frac{1}{2}}\left(0\right)\rho_{\mathsf{atm}} C_d V\left(\frac{n^t_{1,k+\frac{1}{2}}}{\rho_{\mathsf{vap},k+\frac{1}{2}}} - \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}}\right)\right]$$

$$-G^t n^t_{1,k+\frac{1}{2}} + H^t n^{t-dt}_{2,k+\frac{1}{2}}$$

$$\underbrace{\left[\varepsilon^{t-dt}_{k+\frac{1}{2}}\frac{\Delta z}{\Delta t} + G^t\Delta z + D^t_{b,k+1}d_{k+1} + \varepsilon_{k+\frac{1}{2}}\left(0\right)\frac{\rho_{\mathsf{atm}}}{\rho_{\mathsf{vap},k+\frac{1}{2}}} C_d V\right]}_{\Delta^t_{k+\frac{1}{2}}} n^t_{1,k+\frac{1}{2}} =$$

$$D^t_{b,k+1} d_{k+1} n^t_{1,k+\frac{3}{2}} + \varepsilon_{k+\frac{1}{2}}\left(0\right)\rho_{\mathsf{atm}} C_d V \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}} + \Delta z H^t n^{t-dt}_{2,k+\frac{1}{2}} + \varepsilon^{t-2dt}_{k+\frac{1}{2}}\frac{\Delta z}{\Delta t} n^{t-dt}_{1,k+\frac{1}{2}}$$

If the result is rearranged:

$$n^t_{1,k+\frac{1}{2}} = \underbrace{\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}}_{\alpha^t_{k+1}} n^t_{1,k+\frac{3}{2}} + \underbrace{\frac{\varepsilon_{k+\frac{1}{2}}\left(0\right)\rho_{\mathsf{atm}} C_d V \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}} + \Delta z H^t n^{t-dt}_{2,k+\frac{1}{2}} + \varepsilon^{t-2dt}_{k+\frac{1}{2}}\frac{\Delta z}{\Delta t} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}}_{\beta^t_{k+1}}$$

The initialization of the standard case with flux uses the following variables:

Table 2: Initialization of the standard case with flux

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + G^t\Delta z + D^t_{b,k+1} d_{k+1} + \varepsilon_{k+\frac{1}{2}}\left(0\right)\frac{\rho_{\mathsf{atm}}}{\rho_{\mathsf{vap},k+\frac{1}{2}}} C_d V$
α^t_{k+1}	$\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{\varepsilon_{k+\frac{1}{2}}(0)\rho_{\mathsf{atm}} C_d V \mathtt{n}^t_{\mathsf{atm},k+\frac{1}{2}} + \Delta z H^t n^{t-dt}_{2,k+\frac{1}{2}} + \varepsilon^{t-2dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}$

5.2.1.1.2 Initialization with no mass exchange with the atmosphere Considering no flux at the surface $F = 0$, the initilialization of the standard case becomes:

Table 3: Initialization of the standard case with no flux

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + G^t \Delta z + D^t_{b,k+1} d_{k+1}$
α^t_{k+1}	$\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{\Delta z H^t n^{t-dt}_{2,k+\frac{1}{2}} + \varepsilon^{t-2dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}} + \cancelto{0}{F}}{\Delta^t_{k+\frac{1}{2}}}$

5.2.1.2 For the case with saturation (not including the boundaries) eq. 16 becomes eq. 22 with $n^t_{2,k+\frac{1}{2}} = n_{\text{sat}}$:

$$\frac{n_{\text{sat}} - n^{t-dt}_{2,k+\frac{1}{2}}}{\Delta t} = K^t_a n^t_{1,k+\frac{1}{2}} - K^t_d n_{\text{sat}} \tag{22}$$

eq. 15 becomes eq. 23

$$\frac{\partial}{\partial t}\left(\varepsilon n_1\right) = \frac{\partial}{\partial z}\left(D_b \frac{\partial}{\partial z} n_1\right) - K_a n_1 + K_d n_{\text{sat}} \tag{23}$$

The equation above can be discetized, and by using the result in eq. 22 ($\frac{1}{\Delta z}\left[\ldots\right]$ is fully shown in sec. 5.2.1.1):

$$\begin{aligned}\frac{\varepsilon^{t-dt}_{k+\frac{1}{2}} n^t_{1,k+\frac{1}{2}} - \varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} &= \frac{1}{\Delta z}\left[\ldots\right] - \frac{n_{\text{sat}} - n^{t-dt}_{2,k+\frac{1}{2}}}{\Delta t} \\ &= \frac{1}{\Delta z}\left[\ldots\right] + \frac{n^{t-dt}_{2,k+\frac{1}{2}} - n_{\text{sat}}}{\Delta t}\end{aligned}$$

This can be rearranged in the form shown in eq. 19, with $\Delta^t_{k+\frac{1}{2}} = \varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + D^t_{b,k+1} d_{k+1} + D^t_{b,k} d_k \left(1 - \alpha^t_k\right)$:

$$n^t_{1,k+\frac{1}{2}} = \underbrace{\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}}_{\alpha^t_{k+1}} n^t_{1,k+\frac{3}{2}} + \underbrace{\frac{D^t_{b,k} d_k \beta^t_k + \left(\varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}} + n^{t-dt}_{2,k+\frac{1}{2}} - n_{\text{sat}}\right) C_{k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}}_{\beta^t_{k+1}}$$

Therefore, the variables are:

Table 4: Variables for the case with saturation

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + D^t_{b,k+1} d_{k+1} + D^t_{b,k} d_k \left(1 - \alpha^t_k\right)$
α^t_{k+1}	$\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$

Variable	Expression
β^t_{k+1}	$\dfrac{D^t_{b,k} d_k \beta^t_k + \left(\varepsilon^{t-2dt}_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}} + n^{t-dt}_{2,k+\frac{1}{2}} - n_{\mathsf{sat}}\right) C_{k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}$

5.2.1.3 Soil-atmosphere interface with mass exchange with the atmosphere ($k = 0$) An expression like eq. 19 for $k = 0$ is desired, i.e.:

$$\boxed{n^t_{1,\frac{1}{2}} = \alpha^t_1 n^t_{1,\frac{3}{2}} + \beta^t_1} \tag{24}$$

Next follows the derivation for the cases with and without saturation.

5.2.1.3.1 With no saturation From

$$\frac{\varepsilon^{t-dt}_{\frac{1}{2}} n^t_{1,\frac{1}{2}} - \varepsilon^{t-2dt}_{\frac{1}{2}} n^{t-dt}_{1,\frac{1}{2}}}{\Delta t} =$$
$$\frac{1}{z_1 - z_0}\left[D^t_{b,1} d_1 \left(n^t_{1,\frac{3}{2}} - n^t_{1,\frac{1}{2}}\right) - \varepsilon_{\frac{1}{2}}(0)\, \rho_{\mathsf{atm}} C_d V \left(\frac{n^t_{1,\frac{1}{2}}}{\rho_{\mathsf{vap},\frac{1}{2}}} - \mathtt{n}^t_{\mathsf{atm},\frac{1}{2}}\right)\right]$$
$$-G^t n^t_{1,\frac{1}{2}} + H^t n^{t-dt}_{2,\frac{1}{2}}$$

And assuming the following:

$$\mathtt{n}^t_{\mathsf{atm},\frac{1}{2}} = \alpha^t_0 \frac{n^t_{1,\frac{1}{2}}}{\rho_{\mathsf{vap},\frac{1}{2}}} + \beta^t_0 \tag{25}$$

Then

$$\underbrace{\left[\varepsilon^{t-dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} + G^t \Delta z + D_{b,1} d_1 + \varepsilon_{\frac{1}{2}}(0) \frac{\rho_{\mathsf{atm}}}{\rho_{\mathsf{vap},\frac{1}{2}}} C_d V \left(1 - \alpha^t_0\right)\right]}_{\Delta^t_{\frac{1}{2}}} n^t_{1,\frac{1}{2}} =$$
$$D^t_{b,1} d_1 n^t_{1,\frac{3}{2}} + \varepsilon_{\frac{1}{2}}(0)\, \rho_{\mathsf{atm}} C_d V \beta^t_0 + \Delta z H^t n^{t-dt}_{2,\frac{1}{2}} + \varepsilon^{t-2dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} n^{t-dt}_{1,\frac{1}{2}}$$

Therefore:

$$n^t_{1,\frac{1}{2}} = \underbrace{\frac{D^t_{b,1} d_1}{\Delta^t_{\frac{1}{2}}}}_{\alpha^t_1} n^t_{1,\frac{3}{2}} + \underbrace{\frac{\varepsilon_{\frac{1}{2}}(0)\, \rho_{\mathsf{atm}} C_d V \beta^t_0 + \Delta z H^t n^{t-dt}_{2,\frac{1}{2}} + \varepsilon^{t-2dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}}_{\beta^t_1}$$

The variables are:

Table 5: Variables for the soil-atmosphere interface with mass flux but with no saturation

Variable	Expression
$\Delta^t_{\frac{1}{2}}$	$\varepsilon^{t-dt}_{\frac{1}{2}} C_{\frac{1}{2}} + G^t \Delta z + D^t_{b,1} d_1 + \varepsilon_{\frac{1}{2}}(0) \frac{\rho_{\mathsf{atm}}}{\rho_{\mathsf{vap},\frac{1}{2}}} C_d V \left(1 - \alpha^t_0\right)$

Variable	Expression
α_1^t	$\frac{D_{b,1}^t d_1}{\Delta^t_{\frac{1}{2}}}$
β_1^t	$\frac{\varepsilon_{\frac{1}{2}}(0)\rho_{\text{atm}}C_dV\beta_0^t+\Delta z H^t n^{t-dt}_{2,\frac{1}{2}}+\varepsilon^{t-2dt}_{\frac{1}{2}}C_{\frac{1}{2}}n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}$

5.2.1.3.2 With saturation

From

$$\frac{\varepsilon^{t-dt}_{\frac{1}{2}}n^t_{1,\frac{1}{2}}-\varepsilon^{t-2dt}_{\frac{1}{2}}n^{t-dt}_{1,\frac{1}{2}}}{\Delta t}=$$
$$\frac{1}{z_1-z_0}\left[D^t_{b,1}d_1\left(n^t_{1,\frac{3}{2}}-n^t_{1,\frac{1}{2}}\right)-\varepsilon_{\frac{1}{2}}(0)\,\rho_{\text{atm}}C_dV\left(\frac{n^t_{1,\frac{1}{2}}}{\rho_{\text{vap},\frac{1}{2}}}-\mathtt{n}^t_{\text{atm},\frac{1}{2}}\right)\right]$$
$$+\frac{n^{t-dt}_{2,\frac{1}{2}}-n_{\text{sat}}}{\Delta t}$$

And assuming eq. 25, then

$$\underbrace{\left[\varepsilon^{t-dt}_{\frac{1}{2}}\frac{\Delta z}{\Delta t}+D^t_{b,1}d_1+\varepsilon_{\frac{1}{2}}(0)\,\frac{\rho_{\text{atm}}}{\rho_{\text{vap},\frac{1}{2}}}C_dV\left(1-\alpha^t_0\right)\right]}_{\Delta^t_{\frac{1}{2}}}n^t_{1,\frac{1}{2}}=$$
$$D^t_{b,1}d_1n^t_{1,\frac{3}{2}}+\varepsilon_{\frac{1}{2}}(0)\,\rho_{\text{atm}}C_dV\beta^t_0+\left(n^{t-dt}_{2,\frac{1}{2}}-n_{\text{sat}}\right)\frac{\Delta z}{\Delta t}+\varepsilon^{t-2dt}_{\frac{1}{2}}\frac{\Delta z}{\Delta t}n^{t-dt}_{1,\frac{1}{2}}$$

Therefore:

$$n^t_{1,\frac{1}{2}}=\underbrace{\frac{D^t_{b,1}d_1}{\Delta^t_{\frac{1}{2}}}}_{\alpha^t_1}n^t_{1,\frac{3}{2}}+\underbrace{\frac{\varepsilon_{\frac{1}{2}}(0)\,\rho_{\text{atm}}C_dV\beta^t_0+\left(n^{t-dt}_{2,\frac{1}{2}}-n_{\text{sat}}\right)\frac{\Delta z}{\Delta t}+\varepsilon^{t-2dt}_{\frac{1}{2}}\frac{\Delta z}{\Delta t}n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}}_{\beta^t_1}$$

The variables are:

Table 6: Variables for the soil-atmosphere interface with mass flux and saturation

Variable	Expression
$\Delta^t_{\frac{1}{2}}$	$\varepsilon^{t-dt}_{\frac{1}{2}}C_{\frac{1}{2}}+D^t_{b,1}d_1+\varepsilon_{\frac{1}{2}}(0)\,\frac{\rho_{\text{atm}}}{\rho_{\text{vap},\frac{1}{2}}}C_dV\left(1-\alpha^t_0\right)$
α_1^t	$\frac{D_{b,1}^t d_1}{\Delta^t_{\frac{1}{2}}}$
β_1^t	$\frac{\varepsilon_{\frac{1}{2}}(0)\rho_{\text{atm}}C_dV\beta_0^t+\left(n^{t-dt}_{2,\frac{1}{2}}-n_{\text{sat}}\right)C_{\frac{1}{2}}+\varepsilon^{t-2dt}_{\frac{1}{2}}C_{\frac{1}{2}}n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}$

5.2.1.4 Soil-atmosphere interface with no mass exchange with the atmosphere

Note that the desired result is the expression eq. 24, which is eq. 19 for the case $k = 0$. The expression eq. 19 *is written* for $k + 1$, so the first cell to be computed in the aforementioned expression would be cell $\overset{0}{\cancel{k}} + 1 = 1$, missing the cell $k = 0$. The bounding cell (i.e. the cell which includes the point that is exactly on the interface between the soil and the atmosphere) is $\overset{0}{\cancel{k}} = 0$. Therefore, the derivation of the expression must include a flux F towards cell 1 due to possible existing gradients between cell 0—the boundary/surface—and cell 1, which is below the surface.

5.2.1.4.1 With no saturation

From

$$\frac{\varepsilon_{\frac{1}{2}}^{t-dt} n_{1,\frac{1}{2}}^{t} - \varepsilon_{\frac{1}{2}}^{t-2dt} n_{1,\frac{1}{2}}^{t-dt}}{\Delta t} = \frac{1}{z_1 - z_0}\left[D_{b,1}^t d_1 \left(n_{1,\frac{3}{2}}^t - n_{1,\frac{1}{2}}^t\right) - F\right] - G^t n_{1,\frac{1}{2}}^t + H^t n_{2,\frac{1}{2}}^{t-dt}$$

Then,

$$\underbrace{\left[\varepsilon_{\frac{1}{2}}^{t-dt}\frac{\Delta z}{\Delta t} + G^t \Delta z + D_{b,1}^t d_1\right]}_{\Delta_{\frac{1}{2}}^t} n_{1,\frac{1}{2}}^t = D_{b,1}^t d_1 n_{1,\frac{3}{2}}^t - F + \Delta z H^t n_{2,\frac{1}{2}}^{t-dt} + \varepsilon_{\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,\frac{1}{2}}^{t-dt}$$

Therefore:

$$n_{1,\frac{1}{2}}^t = \underbrace{\frac{D_{b,1}^t d_1}{\Delta_{\frac{1}{2}}^t}}_{\alpha_1^t} n_{1,\frac{3}{2}}^t + \underbrace{\frac{-F + \Delta z H^t n_{2,\frac{1}{2}}^{t-dt} + \varepsilon_{\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,\frac{1}{2}}^{t-dt}}{\Delta_{\frac{1}{2}}^t}}_{\beta_1^t}$$

The variables are:

Table 7: Variables for the soil-atmosphere interface with no mass flux or saturation (generic)

Variable	Expression
$\Delta_{\frac{1}{2}}^t$	$\varepsilon_{\frac{1}{2}}^{t-dt} C_{\frac{1}{2}} + G^t \Delta z + D_{b,1}^t d_1$
α_1^t	$\frac{D_{b,1}^t d_1}{\Delta_{\frac{1}{2}}^t}$
β_1^t	$\frac{-F + \Delta z H^t n_{2,\frac{1}{2}}^{t-dt} + \varepsilon_{\frac{1}{2}}^{t-2dt} C_{\frac{1}{2}} n_{1,\frac{1}{2}}^{t-dt}}{\Delta_{\frac{1}{2}}^t}$

If the following is considered, with $d_0 = \frac{1}{z_{\frac{1}{2}}}$:

$$F = D_0^t d_0 \left[n_{1,\frac{1}{2}}^t - \mathtt{n}_{1,\text{sat}}\left(T_s\right)\rho_{\text{vap}}\left(T_s\right)\right] \tag{26}$$

Then the variables are:

Table 8: Variables for the soil-atmosphere interface with no mass flux or saturation

Variable	Expression
$\Delta^t_{\frac{1}{2}}$	$\varepsilon^{t-dt}_{\frac{1}{2}} C_{\frac{1}{2}} + G^t \Delta z + D^t_{b,1} d_1 + D^t_0 d_0$
α^t_1	$\frac{D_{b,1} d_1}{\Delta^t_{\frac{1}{2}}}$
β^t_1	$\frac{D^t_0 d_0 \mathtt{n}_{1,\mathsf{sat}}(T_s)\rho_{\mathsf{vap}}(T_s) + \Delta z H^t n^{t-dt}_{2,\frac{1}{2}} + \varepsilon^{t-2dt}_{\frac{1}{2}} C_{\frac{1}{2}} n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}$

5.2.1.4.2 With saturation

From

$$\frac{\varepsilon^{t-dt}_{\frac{1}{2}} n^t_{1,\frac{1}{2}} - \varepsilon^{t-2dt}_{\frac{1}{2}} n^{t-dt}_{1,\frac{1}{2}}}{\Delta t} = \frac{1}{z_1 - z_0}\left[D^t_{b,1} d_1 \left(n^t_{1,\frac{3}{2}} - n^t_{1,\frac{1}{2}}\right) - F\right] + \frac{n^{t-dt}_{2,\frac{1}{2}} - n_{\mathsf{sat}}}{\Delta t}$$

Then,

$$\underbrace{\left[\varepsilon^{t-dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} + D^t_{b,1} d_1\right]}_{\Delta^t_{\frac{1}{2}}} n^t_{1,\frac{1}{2}} = D^t_{b,1} d_1 n^t_{1,\frac{3}{2}} - F + \frac{n^{t-dt}_{2,\frac{1}{2}} - n_{\mathsf{sat}}}{\Delta t} + \varepsilon^{t-2dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} n^{t-dt}_{1,\frac{1}{2}}$$

Therefore:

$$n^t_{1,\frac{1}{2}} = \underbrace{\frac{D^t_{b,1} d_1}{\Delta^t_{\frac{1}{2}}}}_{\alpha^t_1} n^t_{1,\frac{3}{2}} + \underbrace{\frac{-F + \left(n^{t-dt}_{2,\frac{1}{2}} - n_{\mathsf{sat}}\right)\frac{\Delta z}{\Delta t} + \varepsilon^{t-2dt}_{\frac{1}{2}} \frac{\Delta z}{\Delta t} n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}}_{\beta^t_1}$$

The variables are:

Table 9: Variables for the soil-atmosphere interface with no mass flux but with saturation (generic)

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{\frac{1}{2}} C_{k+\frac{1}{2}} + D^t_{b,k+1} d_{k+1}$
α^t_{k+1}	$\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{-F + \left(n^{t-dt}_{2,k+\frac{1}{2}} - n_{\mathsf{sat}}\right) C_{k+\frac{1}{2}} + \varepsilon^{t-2dt}_{\frac{1}{2}} C_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}$

Using eq. 26, the variables are:

Table 10: Variables for the soil-atmosphere interface with no mass flux but with saturation

Variable	Expression
$\Delta^t_{\frac{1}{2}}$	$\varepsilon^{t-dt}_{\frac{1}{2}} C_{\frac{1}{2}} + D^t_{b,1} d_1 + D^t_0 d_0$
α^t_1	$\frac{D^t_{b,1} d_1}{\Delta^t_{\frac{1}{2}}}$
β^t_1	$\frac{D^t_0 d_0 \mathtt{n}_{1,\mathsf{sat}}(T_s)\rho_{\mathsf{vap}}(T_s)+\left(n^{t-dt}_{2,\frac{1}{2}}-n_{\mathsf{sat}}\right)C_{\frac{1}{2}}+\varepsilon^{t-2dt}_{\frac{1}{2}}C_{\frac{1}{2}}n^{t-dt}_{1,\frac{1}{2}}}{\Delta^t_{\frac{1}{2}}}$

5.2.1.5 Deep ground interface ($k = N - 1$ for eq. 19)

In this case, it is assumed that there is no flux coming from below $F = 0$.

5.2.1.5.1 With no saturation

From

$$\frac{\varepsilon^{t-dt}_{N-\frac{1}{2}} n^t_{1,N-\frac{1}{2}} - \varepsilon^{t-2dt}_{N-\frac{1}{2}} n^{t-dt}_{1,N-\frac{1}{2}}}{\Delta t} =$$

$$\frac{1}{z_N - z_{N-1}} \left\{ \cancelto{0}{F} - D^t_{b,N-1} d_{N-1} \left[n^t_{1,N-\frac{1}{2}} - \underbrace{\left(\alpha^t_{N-1} n^t_{1,N-\frac{1}{2}} + \beta^t_{N-1} \right)}_{n^t_{1,N-\frac{3}{2}}} \right] \right\}$$

$$-G^t n^t_{1,N-\frac{1}{2}} + H^t n^{t-dt}_{2,N-\frac{1}{2}}$$

The following is achieved:

$$n^t_{1,N-\frac{1}{2}} = \frac{D^t_{b,N-1} d_{N-1} \beta^t_{N-1} + \Delta z H^t n^{t-dt}_{2,N-\frac{1}{2}} + \varepsilon^{t-2dt}_{N-\frac{1}{2}} C_{N-\frac{1}{2}} n^{t-dt}_{1,N-\frac{1}{2}}}{\varepsilon^{t-dt}_{N-\frac{1}{2}} C_{N-\frac{1}{2}} + \Delta z G^t + D_{N-1} d_{N-1} \left(1 - \alpha^t_{N-1}\right)}$$

5.2.1.5.2 With saturation

From

$$\frac{\varepsilon^{t-dt}_{N-\frac{1}{2}} n^t_{1,N-\frac{1}{2}} - \varepsilon^{t-2dt}_{N-\frac{1}{2}} n^{t-dt}_{1,N-\frac{1}{2}}}{\Delta t} =$$

$$\frac{1}{z_N - z_{N-1}} \left\{ \cancelto{0}{F} - D^t_{b,N-1} d_{N-1} \left[n^t_{1,N-\frac{1}{2}} - \underbrace{\left(\alpha^t_{N-1} n^t_{1,N-\frac{1}{2}} + \beta^t_{N-1} \right)}_{n^t_{1,N-\frac{3}{2}}} \right] \right\}$$

$$+ \frac{n^{t-dt}_{2,N-\frac{1}{2}} - n_{\mathsf{sat}}}{\Delta t}$$

The following is achieved:

$$n^t_{1,N-\frac{1}{2}} = \frac{D^t_{b,N-1}d_{N-1}\beta^t_{N-1} + \left(n^{t-dt}_{2,N-\frac{1}{2}} - n_{\text{sat}}\right)C_{N-\frac{1}{2}} + \varepsilon^{t-2dt}_{N-\frac{1}{2}}C_{N-\frac{1}{2}}n^{t-dt}_{1,N-\frac{1}{2}}}{\varepsilon^{t-dt}_{N-\frac{1}{2}}C_{N-\frac{1}{2}} + D^t_{b,N-1}d_{N-1}\left(1-\alpha^t_{N-1}\right)} \tag{27}$$

Note that the code implements other phenomena not described in the equations above. Next follow some improvements on the equations above:

5.2.2 Improvements over sec. 5.2.1

5.2.2.1 The sublimation problem The implementation of the equations in sec. 5.2.1 make the sublimed quantity to be $\approx \varepsilon n_{\text{sat}}, \forall \Delta t$ for each time step. However, sublimation is dependent on Δt!

To solve this, one does not has to separate the diffusion/adsorption phase from the sublimation phase, but to treat everything implicitly (implicit method).

Departing from eq. 5.2.1.1, one can rewrite the expression to:

$$\frac{\varepsilon n^t_{1,k+\frac{1}{2}} - \varepsilon n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta t} = \underbrace{\frac{1}{z_{k+1}-z_k}\left(D^t_{b,k+1}\frac{n^t_{1,k+\frac{3}{2}} - n^t_{1,k+\frac{1}{2}}}{z_{k+\frac{3}{2}} - z_{k+\frac{1}{2}}} - D^t_{b,k}\frac{n^t_{1,k+\frac{1}{2}} - n^t_{1,k-\frac{1}{2}}}{z_{k+\frac{1}{2}} - z_{k-\frac{1}{2}}}\right)}_{F_{\text{net}}} + \underbrace{\left[-K^t_a n^t_{1,k+\frac{1}{2}} + \left(\frac{K^t_d}{1+\Delta t K^t_d}\right)\left(\Delta t K^t_a n^t_{1,k+\frac{1}{2}} + n^{t-dt}_{2,k+\frac{1}{2}}\right)\right]}_{\Phi_{\text{ads}}} \tag{28}$$

This is greater than 0 if water vapor is present in the cell, i.e. if $F_{\text{Net}} + \Phi_{\text{ads}} > 0$ there is accumulation of vapor in the cell.

Instead of writing eq. 20, which leads to eq. 19, (α^t_{k+1} and β^t_{k+1} depend on α^t_k and β^t_k, respectively), the following is established:

$$\text{ice}^{t-dt}_{k-\frac{1}{2}} \neq 0 \implies n^t_{1,k-\frac{1}{2}} = n^t_{\text{sat},k-\frac{1}{2}}$$

$n_{\text{sat},k-\frac{1}{2}}$ saturated in relation to ice. The recurrence equation can be reused as $n^t_{1,k-\frac{1}{2}} = \underbrace{0}_{\alpha^t_k} \cdot n^t_{1,k+\frac{1}{2}} + \underbrace{n^t_{\text{sat},k-\frac{1}{2}}}_{\beta^t_k}$, with the variables therefore being $\alpha^t_k = 0$ and $\beta^t_k = n^t_{\text{sat},k-\frac{1}{2}}$ for eq. 20, while the ones for eq. 19 are shown below:

Table 11: Sublimation improvement for the standard case

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon C_{k+\frac{1}{2}} + G^t\Delta z + D_{b,k+1}d_{k+1} + D_{b,k}d_k\left(1-\alpha^t_k\right)$

Variable	Expression
α^t_{k+1}	$\frac{D_{b,k+1}d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{f\left(\beta^t_k\right)}{\Delta^t_{k+\frac{1}{2}}} = \frac{f\left(n^t_{1,k-\frac{1}{2}}\right)}{\Delta^t_{k+\frac{1}{2}}} \overset{\text{ice}^{t-dt}_{k-\frac{1}{2}}\neq 0}{=} \frac{f\left(n^t_{\text{sat},k-\frac{1}{2}}\right)}{\Delta^t_{k+\frac{1}{2}}}$

If there is ice in the cell at $t - dt$, the scheme to be followed is the following:

1. At time $t - dt$ compute $Q^{t-dt}_{k+\frac{1}{2}} = \text{ice}^{t-dt}_{k+\frac{1}{2}} + \varepsilon^{t-2dt} n^{t-dt}_{\text{sat},k+\frac{1}{2}}$.
2. At time t' an intermediate value is computed: $Q^{t'} = \text{ice}^{t-dt} + \varepsilon^{t-dt} n^t_{\text{sat}}$, where $\varepsilon^{t-dt} = f\left(\text{ice}^{t-dt}\right)$.
3. Compute $(F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t$
4. At time t, compute the final value according to one of the following cases:

   1. If $\left|\underbrace{F_{\text{net}} + \Phi_{\text{ads}}}_{<0}\right| > Q^{t-dt}$—there is no ice left after transport and everything sublimed—it can be seen that $\text{ice}^t = 0$, so α_k and β_k are not modified (see sec. 5.2.1.1). The following is defined: $\varepsilon_0 n^{t-dt}_{k+\frac{1}{2}} = Q^{t-dt}$, as it is supposed that there is no more ice. In this expression, ε_0 is the real value of ε at time t, which can be justified by the conservation of mass: $Q^{t-dt} = \text{ice}^{t-dt} + \varepsilon^{t-2dt} n^{t-dt} = \varepsilon_0 n^{t-dt}$. The calculation is resumed with $n^{t-dt}_{k+\frac{1}{2}} = \frac{Q^{t-dt}}{\varepsilon_0}$.
   2. If there is some ice left after the transport step, i.e. $(F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t > 0$, the flux remains the same: $n^t_{1,k+\frac{1}{2}} = n^t_{\text{sat},k+\frac{1}{2}}$. Variable ice^t is computed according to the conservation of mass:

$$\text{ice}^t + \varepsilon^{t-dt} n^t_{\text{sat}} = Q^{t-dt} + (F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t$$

so

$$\text{ice}^t = Q^{t-dt} + (F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t - \varepsilon^{t-dt} n^t_{\text{sat}}$$

If $n^t_{1,k+\frac{1}{2}}$ exceeds n^t_{sat} ($n^t_{1,k+\frac{1}{2}} > n^t_{\text{sat},k+\frac{1}{2}}$) when there was no ice already present before, the following is established:

1. $\alpha_k = 0$, $\beta_k = n^t_{\text{sat},k+\frac{1}{2}}$
2. $n^t_{1,k+\frac{1}{2}} = n^t_{\text{sat},k+\frac{1}{2}}$
3. The profile is recalculated with these values, i.e. a new $F_{\text{net}} + \Phi_{\text{ads}}\Delta t$ is found
4. Find the new ice^t to ensure mass conservation:

$$\text{ice}^t + \varepsilon^{t-dt}\underbrace{n^t_{\text{sat}}}_{n^t_1} = \underbrace{\varepsilon^{t-2dt} n^{t-dt}_1}_{Q^{t-dt}} + \underbrace{(F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t}_{\frac{\Delta\varepsilon_1 n_1}{\Delta t}}$$

so

$$\text{ice}^t = \varepsilon^{t-2dt} n^{t-dt}_1 + (F_{\text{net}} + \Phi_{\text{ads}})\,\Delta t - +\varepsilon^{t-dt}\underbrace{n^t_{\text{sat}}}_{n^t_1}$$

5.2.2.2 A generalisation of the expressions The formulae can be arbitrarily complex if more and more phenomena/improvements are added. The general formula **for the general case** that the model uses is the following for eq. 19:

Table 12: Variables for the general case

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + A^t_{k+\frac{1}{2}} + D^t_{b,k+1} d_{k+1} + D^t_{b,k} d_k \left(1 - \alpha^t_k\right)$
α^t_{k+1}	$\frac{D^t_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{D^t_{b,k} d_k \beta^t_k + B^t_{k+\frac{1}{2}} + \varepsilon C_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}$

5.3 New derivation including the fitting

The fitting proposed in sec. 3.2 using a piecewise function defined by two lines, is to be included in the formulas of sec. 5.2. Therefore, a new derivation is carried out below.

The two-segment isotherm found (see sec. 3.2) fits data that is given at a certian temperature. Because the model requires isotherms for a whole spectrum of temperatures, it will be considered that n_{ads} is described by an [Arrhenius equation](https://en.wikipedia.org/wiki/Arrhenius_equation). Considering fig. 13 (Q_1 and Q_2 are heats of adsorption):

- Segment 1 is defined by

$$\hat{n}_{\text{ads}} = \underbrace{\frac{K_a}{K_d}}_{k_{\text{ads}}} \exp\left(\frac{Q_1}{RT}\right) \hat{n}_{\text{vap}} \tag{29}$$

- Segment 2 is defined by

$$\hat{n}_{\text{ads}} = \underbrace{\frac{K'_a}{K'_d}}_{k'_{\text{ads}}} \exp\left(\frac{Q_2}{RT}\right) \hat{n}_{\text{vap}} + \boldsymbol{C}_0 \tag{30}$$

Note that the functions above are written with respect to absolute temperature 0. If the fitted isotherms are obtained at a reference temperature T_{ref}, the equations above are rewritten in the form below:

$$\frac{\hat{n}_{\text{ads}}}{\hat{n}_{\text{vap}}} = k^{\text{ref}}_{\text{ads}} e^{\frac{Q^{\text{ref}}}{R}\left(\frac{1}{T} - \frac{1}{T_{\text{ref}}}\right)} = \underbrace{\left(k^{\text{ref}}_{\text{ads}}\right)^2 e^{\frac{Q^{\text{ref}}}{RT}}}_{\text{new factor}} \Big/ \underbrace{k^{\text{ref}}_{\text{ads}} e^{\frac{Q^{\text{ref}}}{RT_{\text{ref}}}}}_{\text{reference isotherm}} \tag{31}$$

It is easy to see that for values contained in segment 1, the equations to be used are the ones presented in sec. 5.2. The differences between the old derivation shown in sec. 5.2 and the new derivation—shown herebelow—is that the second segment had a slope of 0 (equation was $\hat{n}_{\text{ads}} = \hat{n}_{\text{sat}}$), while now it has a slope of $\frac{K'_a}{K'_d} \exp\left(\frac{Q_2}{RT}\right)$ and the line that defines it is assumed to be offset by C_0 (equation given above in eq. 30). Using the same naming convention and reasoning of eq. 14 leads to eq. 32 and eq. 33 for segment 2:

$$\frac{\partial}{\partial t}\left(\varepsilon n_1\right) = \frac{\partial}{\partial z}\left(D_b \frac{\partial}{\partial z} n_1\right) - K'_a n_1 + K'_d n_2 - K'_d \boldsymbol{C}_0 \tag{32}$$

$$\frac{\partial}{\partial t}\left(n_2\right) = K'_a n_1 - K'_d n_2 + K'_d \boldsymbol{C}_0 \tag{33}$$

Note that for a given temperature T, eq. 30 can be written as $\hat{n}_{\text{ads}} = \frac{K_a'}{K_d'}\hat{n}_{\text{vap}} + \boldsymbol{C}_0 \equiv n_2 = \frac{K_a'}{K_d'}n_1 + \boldsymbol{C}_0$ which arises from eq. 33 when considering $\frac{\partial}{\partial t}(n_2) = 0$, or equivalently, from eq. 32 assuming $\frac{\partial}{\partial t}(\varepsilon n_1) = \frac{\partial}{\partial z}\left(D_b \frac{\partial}{\partial z} n_1\right)$. This considerations mean that water is not adsorbed over time, or that the vapour formed or released over time (which is interdependent with porosity) spreads and is responsible for its distribution along the z axis, respectively (?TODO: is this correct?).

Being Δt the timestep between instant $t - dt$ and t, eq. 33 can be discretized to eq. 34:

$$\frac{n_2^t - n_2^{t-dt}}{\Delta t} = \left(K_a'\right)^t n_1^t - \left(K_d'\right)^t n_2^t + \left(K_d'\right)^t \boldsymbol{C}_0 \tag{34}$$

By isolating n_2^t from eq. 17, eq. 35 is found:

$$n_2^t = \frac{\Delta t \left(K_a'\right)^t n_1^t + n_2^{t-dt} + \Delta t \left(K_d'\right)^t \boldsymbol{C}_0}{1 + \Delta t \left(K_d'\right)^t} \tag{35}$$

The value of C_0 can be found as segments 1 and 2 must intersect at saturation, i.e. eq. 29 is equal to eq. 30 when $\hat{n}_{\text{ads}} = \hat{n}_{\text{sat}}$:

$$\begin{aligned}
\frac{K_a}{K_d}\exp\left(\frac{Q_1}{RT}\right)\left(\hat{n}_{\text{vap}}\right)_{\text{sat}} &= \frac{K_a'}{K_d'}\exp\left(\frac{Q_2}{RT}\right)\left(\hat{n}_{\text{vap}}\right)_{\text{sat}} + \boldsymbol{C}_0 \\
\boldsymbol{C}_0 &= \left[k_{\text{ads}}\exp\left(\frac{Q_1}{RT}\right) - k_{\text{ads}}'\exp\left(\frac{Q_2}{RT}\right)\right]\left(\hat{n}_{\text{vap}}\right)_{\text{sat}} \\
&= \left[1 - \frac{k_{\text{ads}}'}{k_{\text{ads}}}\exp\left(\frac{Q_2 - Q_1}{RT}\right)\right]\underbrace{k_{\text{ads}}\exp\left(\frac{Q_1}{RT}\right)}_{\text{de 29, } \frac{\hat{n}_{\text{sat}}}{(\hat{n}_{\text{vap}})}}\left(\hat{n}_{\text{vap}}\right)_{\text{sat}} \\
&= \left[1 - \frac{k_{\text{ads}}'}{k_{\text{ads}}}\exp\left(\frac{Q_2 - Q_1}{RT}\right)\right]\hat{n}_{\text{sat}}
\end{aligned}$$

5.3.1 Case for segment 2

Using eq. 35 in the discretized form of eq. 32:

$$\begin{aligned}
&\frac{\varepsilon_{k+\frac{1}{2}}^{t-dt} n_{1,k+\frac{1}{2}}^t - \varepsilon_{k+\frac{1}{2}}^{t-2dt} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta t} = \\
&\frac{1}{z_{k+1} - z_k}\left(D_{b,k+1}^t \frac{n_{1,k+\frac{3}{2}}^t - n_{1,k+\frac{1}{2}}^t}{z_{k+\frac{3}{2}} - z_{k+\frac{1}{2}}} - D_{b,k}^t \frac{n_{1,k+\frac{1}{2}}^t - n_{1,k-\frac{1}{2}}^t}{z_{k+\frac{1}{2}} - z_{k-\frac{1}{2}}}\right) \\
&- \left(K_a'\right)^t n_{1,k+\frac{1}{2}}^t + \left(\frac{\left(K_d'\right)^t}{1 + \Delta t \left(K_d'\right)^t}\right)\left[\Delta t \left(K_a'\right)^t n_{1,k+\frac{1}{2}}^t + n_{2,k+\frac{1}{2}}^{t-dt} + \Delta t \left(K_d'\right)^t \boldsymbol{C}_0^t\right] - \left(K_d'\right)^t \boldsymbol{C}_0^t
\end{aligned}$$

$$\frac{\varepsilon_{k+\frac{1}{2}}^{t-dt} n_{1,k+\frac{1}{2}}^{t} - \varepsilon_{k+\frac{1}{2}}^{t-2dt} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta t} =$$

$$\frac{1}{z_{k+1}-z_k}\left(D_{b,k+1}^t \frac{n_{1,k+\frac{3}{2}}^t - n_{1,k+\frac{1}{2}}^t}{z_{k+\frac{3}{2}} - z_{k+\frac{1}{2}}} - D_{b,k}^t \frac{n_{1,k+\frac{1}{2}}^t - n_{1,k-\frac{1}{2}}^t}{z_{k+\frac{1}{2}} - z_{k-\frac{1}{2}}}\right)$$

$$+\underbrace{\left(\frac{\Delta t\left(K_d'\right)^t\left(K_a'\right)^t}{1+\Delta t\left(K_d'\right)^t} - \left(K_a'\right)^t\right)}_{-(G')^t = -\frac{(K_a')^t}{1+\Delta t\left(K_d'\right)^t}} n_{1,k+\frac{1}{2}}^t + \underbrace{\frac{\left(K_d'\right)^t}{1+\Delta t\left(K_d'\right)^t}}_{(H')^t} n_{2,k+\frac{1}{2}}^{t-dt} + \underbrace{\underbrace{\frac{\left[\left(K_d'\right)^t\right]^2}{1+\Delta t\left(K_d'\right)^t}}_{(H')^t\left(K_d'\right)^t} \boldsymbol{C}_0^t\left(\Delta t - 1\right)}_{(M')^t}$$

Therefore:

$$\frac{\varepsilon_{k+\frac{1}{2}}^{t-dt} n_{1,k+\frac{1}{2}}^{t} - \varepsilon_{k+\frac{1}{2}}^{t-2dt} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta t} =$$

$$\frac{1}{z_{k+1}-z_k}\left[D_{b,k+1}^t d_{k+1}\left(n_{1,k+\frac{3}{2}}^t - n_{1,k+\frac{1}{2}}^t\right) - D_{b,k}^t d_k\left(n_{1,k+\frac{1}{2}}^t - n_{1,k-\frac{1}{2}}^t\right)\right]$$

$$-\left(G'\right)^t n_{1,k+\frac{1}{2}}^t + \left(H'\right)^t n_{2,k+\frac{1}{2}}^{t-dt} + \left(M'\right)^t$$

By using eq. 20 in the equation above:

$$\frac{\varepsilon_{k+\frac{1}{2}}^{t-dt} n_{1,k+\frac{1}{2}}^{t} - \varepsilon_{k+\frac{1}{2}}^{t-2dt} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta t} =$$

$$\frac{1}{z_{k+1}-z_k}\left\{D_{b,k+1}^t d_{k+1}\left(n_{1,k+\frac{3}{2}}^t - n_{1,k+\frac{1}{2}}^t\right) - D_{b,k}^t d_k\left[n_{1,k+\frac{1}{2}}^t - \underbrace{\left(\alpha_k^t n_{1,k+\frac{1}{2}}^t + \beta_k^t\right)}_{n_{1,k-\frac{1}{2}}^t}\right]\right\}$$

$$-\left(G'\right)^t n_{1,k+\frac{1}{2}}^t + \left(H'\right)^t n_{2,k+\frac{1}{2}}^{t-dt} + \left(M'\right)^t$$

$$\underbrace{\left[\varepsilon_{k+\frac{1}{2}}^{t-dt}\frac{\Delta z}{\Delta t} + \left(G'\right)^t \Delta z + D_{b,k+1}^t d_{k+1} + D_{b,k}^t d_k\left(1-\alpha_k^t\right)\right]}_{\Delta_{k+\frac{1}{2}}^t} n_{1,k+\frac{1}{2}}^t =$$

$$D_{b,k+1}^t d_{k+1} n_{1,k+\frac{3}{2}}^t + D_{b,k}^t d_k \beta_k^t + \Delta z\left(H'\right)^t n_{2,k+\frac{1}{2}}^{t-dt} + \Delta z\left(M'\right)^t + \varepsilon_{k+\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,k+\frac{1}{2}}^{t-dt}$$

And re-arranging the equation in the form of eq. 19:

$$n_{1,k+\frac{1}{2}}^t = \underbrace{\frac{D_{b,k+1}^t d_{k+1}}{\Delta_{k+\frac{1}{2}}^t}}_{\alpha_{k+1}^t} n_{1,k+\frac{3}{2}}^t$$

$$+\underbrace{\frac{D_{b,k}^t d_k \beta_k^t + \Delta z\left(H'\right)^t n_{2,k+\frac{1}{2}}^{t-dt} + \Delta z\left(M'\right)^t + \varepsilon_{k+\frac{1}{2}}^{t-2dt}\frac{\Delta z}{\Delta t} n_{1,k+\frac{1}{2}}^{t-dt}}{\Delta_{k+\frac{1}{2}}^t}}_{\beta_{k+1}^t}$$

Table 13: Variarbles for the standard case (fitted isotherm)

Variable	Expression
$\Delta^t_{k+\frac{1}{2}}$	$\varepsilon^{t-dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} + (G')^t \Delta z + D^t_{b,k+1} d_{k+1} + D^t_{b,k} d_k \left(1 - \alpha^t_k\right)$
α^t_{k+1}	$\frac{D_{b,k+1} d_{k+1}}{\Delta^t_{k+\frac{1}{2}}}$
β^t_{k+1}	$\frac{D^t_{b,k} d_k \beta^t_k + \Delta z (H')^t n^{t-dt}_{2,k+\frac{1}{2}} + \Delta z (M')^t + \varepsilon^{t-2dt}_{k+\frac{1}{2}} C_{k+\frac{1}{2}} n^{t-dt}_{1,k+\frac{1}{2}}}{\Delta^t_{k+\frac{1}{2}}}$

5.3.2 New `soilwater()` subroutine

5.3.2.1 Initialisation The first loop is carried out when `firstcall .eqv. .true.` (see sec. 6.2.2.1), and in it some variables are initizialised. `midlayer_dz` and `interlayer_dz` are initialised as before, as well as `D0`. The adsorption saturation level `adswater_sat` neither does change, $n_{\text{sat}} = \rho_{\text{soil}} \underbrace{\frac{SSA}{S_{\text{molecule H}_2\text{O}}}}_{\text{num. of molecules on the grain's surface}} m_w$, being m_w the mass of the water molecule as in sec. 3.2.2 (related: sec. **??**). Instead of being a limit value, in this case `adswater_sat` marks the transition between segment 1 and segment 2 because it marks the saturation **of the first layer**. Instead of being called `adswater_sat` it should be called `adswater_sat_monolayer` or something like that.

On the one hand, for the case 1D–1 column, where `ngrid .eq. 1`, `znsoil` is computed as $n_1 = MMR_{\text{H}_2\text{O}} M_{\text{atmo}} \frac{p_{\text{atmo}}}{RT_{\text{soil}}}$, where MMR is the water vapor mass mixing ratio, M_{atmo} is the mean molecular weight of the Martian atmosphere (also appeared in sec. 3.2.2), p_{atmo} are atmospheric pressure levels, and T_{soil} the subsurface temperatures. An alternative is to set $n_1 = 0$. Ice quantity in kilogram per metre cube of regolith (stored in `ice`) is initially set to 0. Variable `saturation_water_ice`—the ratio between the actual quantity of water ice and the quantity of ice needed to fill the void—is initialized the same way as in the current implementation and used to compute `porosity`. With `porosity`, the value of the adsorbed water in subsurface cells n_2 (named `adswater`) can be computed from adsorption and desorption constants, which are related by k_{ads}. These constants can be computed from $k^{\text{ref}}_{\text{ads}}$, but note that in the new implementation there are two regimes (i.e. two segments defined by Q_1 and Q_2). If $n_2 = \frac{K_a}{K_d} n_1 > n_{\text{sat}}$ ($\frac{K_a}{K_d}$ describe segment 1, which ends at $n_{\text{ads}} = n_{\text{sat}}$), then $n_2 = \frac{K'_a}{K'_d} n_1$ (segment 2) must be used.

On the other hand, in the the case of `ngrid` columns, all values of n_1 or `znsoil`, `ice`, and `adswater` are retrieved from the caller, and are used to compute `saturation_water_ice` and `porosity`.

5.3.2.2 Computation of transport coefficents This loop apparently does not need modifications.

5.3.2.3 Main loop (computation of adsorption constants and n_1 and n_2) The values of the adsorption and desorption constants depend on the region (i.e. segment) they are in. A simple solution is to initially compute all the constants are computed as though they were for segment 1. Imposing the steady state from eq. 16, $\frac{dn_2}{dt} = 0 \implies 0 = K_a n_1 - K_d n_2$, then $n_1 = \frac{K_d}{K_a} n_2$ (as seen previously). Variable $n_{1,\text{sat}} = \frac{K_d}{K_a} n_{\text{sat}}$—the amount of water vapor when the monolayer is fully saturated/filled by molecules—can be calculated along `C`, `E`, and `F`. One then performs the computation of `A` and `B`, assuming no saturation of the first monolayer.

The coefficients Δ_0, α_0, and β_0 of the surface layer are also computed. With coefficients `A` to `F`, it is easy to compute n_1 at the bottom layer. If $n_2 = \frac{K_a}{K_d} n_1 > n_{\text{sat}}$ (K_a and K_d describe segment 1, which ends at $n_{\text{ads}} = n_{\text{sat}}$), then $n_2 = \frac{K'_a}{K'_d} n_1$ (K'_a and K'_d describe segment 2) must be used. With these new values of K'_a and K'_d, variables `A` to `F` are recomputed, which are then used to find n_1 and n_2 of the bottom layer. When the lowest layer is computed, the same is carried out for layer $k = N - 1$. Variables n_1 and n_2 are computed assuming they are on segment 1 (K_a and K_d). The resulting n_2 value is compared to n_{sat}, and if the former is greater than the latter, K'_a and K'_d must be used in the computation of coefficients Δ_{N-1}, α_{N-1}, and β_{N-1}, as well as Δ_N, α_N, and β_N (β_N depends on β_{N-1}). The bottom layer is recomputed as if it was in the first segment and the result is checked again against the value of the monolayer. The procedure is then repeated for layer $k = N - 2$—which may affect layers $k = N - 1$ and $k = N$—and upwards up to $k = 1$.

In brief, every cell is assumed to be on segment 1, and if they are not by comparing the result to n_{sat}, its coefficients Δ_k, α_k, and β_k, and those of the cells below need to be changed. Every cell below—whose coefficients depend on the cell above—then needs to be initially recomputed as if they were in segment 1 and the result checked against the transition point n_{nsat} (after the recomputing, the result may be on segment 2 too). This is done for every `nsoil` cell. Although, it does not seem to be very efficient, the the `soilwater()` subroutine does not act as a bottleneck when coupled to the LMDZ GCM.

Another method is to invert the system of equations eq. 19, but setting the value of the coefficients for all `nsoil` may not be trivial.

6 Coupling the subsurface model in LMDZ GCM

The model is programmed in Fortran. A Fortran guide is found in sec. 10.4 and the model's coding conventions in sec. 6.1.6.

6.1 Short introduction to the LMDZ GCM

The LMDZ GCM "calculates the temporal evolution of a set of variables that control or describe the Martian meteorology and climate at different points of a 3D grid that covers the entire atmosphere".

For a variable X, a set of parametrizations—i.e. parametrizations $1, 2\ldots$ describing the variable X according to different parameters—are used to calculate its temporal evolution or tendency $\left(\frac{\partial X}{\partial t}\right)_1$, $\left(\frac{\partial X}{\partial t}\right)_2$. These are then used to compute the value of $X_{t+\delta t} = X_t + \delta t \left(\frac{\partial X}{\partial t}\right)_1 + \delta t \left(\frac{\partial X}{\partial t}\right)_2 + \ldots$ at time step $t + \delta t$ from a known state X_t at time t.

6.1.1 Dynamical-phyisical separation and variables

The model—handled by `gcm.F`—operates in two parts that are integrated according to different schemes:

- A *dynamical part* that contains the numerical solution of the general equations for atmospheric circulation[12]. This part is responsible for the horizontal exchanges in the grid. The dyamical variables are:

[12] This part is valid for all atmospheres of terrestrial type such as Earth or Mars.

- Potential temperature teta or $\theta = T\left(\frac{P}{P_{\text{ref}=610\,\text{Pa}\atop\text{(Mars)}}}\right)^{-\kappa}$, where $\kappa = \frac{R}{C_p}$ represented as kappa in the dynamical part and rcp in the physical part
- Surface pressure ps
- The atmosphere mass in each grid box masse
- Covariant meridional and zonal winds ucov = cu * u and vcov = cv * v[13], respectively
- Mixing ratio of tracers q

• A *physical part* that is specific to the planet, and can be seen as a juxtaposition of atmosphere "columns" that do not interact with each other. The physical timestep is iphysiq times longer than the dynamical timestep. The physical variables are:

- Winds u and v in m s^{-1}
- Temperature T in K
- Pressure at the middle of the layers play and at the interlayers plev, both in Pa
- CO_2 ice on the surface co2ice in kg/m^2
- Surface temperature tsurf in K
- Temperature at different layers under the surface tsoil in K
- Surface emissivity emis
- Wind variance q2 (square root of the turbulent kinetic energy)
- Tracer on the surface qsurf in kg/m^2
- Other tracers—stored in the three-dimensional array q and controlled by callphys.def—for dust particles, for other chemical species which depict the composition of the atmosphere, for water vapor and water ice particles... as configured in traceur.def

6.1.2 Basic algorithm

At every timestep, the following steps are carried out:

1. Calculation of $\left(\frac{\partial X}{\partial t}\right)$ (as it appears in caldyn.F)
2. Integration of the tendencies to retrieve $X_{t+\delta t}$ (as it appears in integrd.F)
3. After every iphysiq dynamical timesteps, calculation of the tendencies arising from the physical part (via calfis.F and then physic.F)
4. Integration of the physical variables (as it appears in addfi.F)
5. After every idissip dynamical timesteps, calculation and inegration of tendencies due to horizontal dissipation and the "sponge layer".

6.1.3 Grid boxes

A brief description of the horizontal grids and vertical grids is presented in this section.

6.1.3.1 Horizontal grids The coordinates of a variable are identified with rlonu, rlatu, rlonv, and rlatv (see fig. 39), longitudes and latitudes being expressed in radians.

[13]Variables cu and cv only depend on the latitude.

Note that *the physical grid and the dynamical grid are different* (as seen in fig. 39), even if a single parameter `-d 64x48x29`[14] was passed when compiling the model:

- The dynamical grid is very much a grid of `iim` $\times$ `jjm` $= IM \times JM$ points, where points at $i = 1$ and $j = 1$ are respectively the same as those at $i = IM + 1 =$ `iip1` and $j = JM + 1 =$ `jjp1` due to periodicity in longitude. Points at $j = 1, JM$ are all the same points, as these all respectively converge to the North and South poles. In other words, points at $j = 1, JM$ are repeated $IM + 1$ times (accounting for halo values).
- The physical grid does not contain repeated points as it is represented in an array in memory. "In practice, computations relative to the physics are made for a series of `ngrid` atmospheric columns, where NGRID $= IM \times (JM - 1) + 2$.

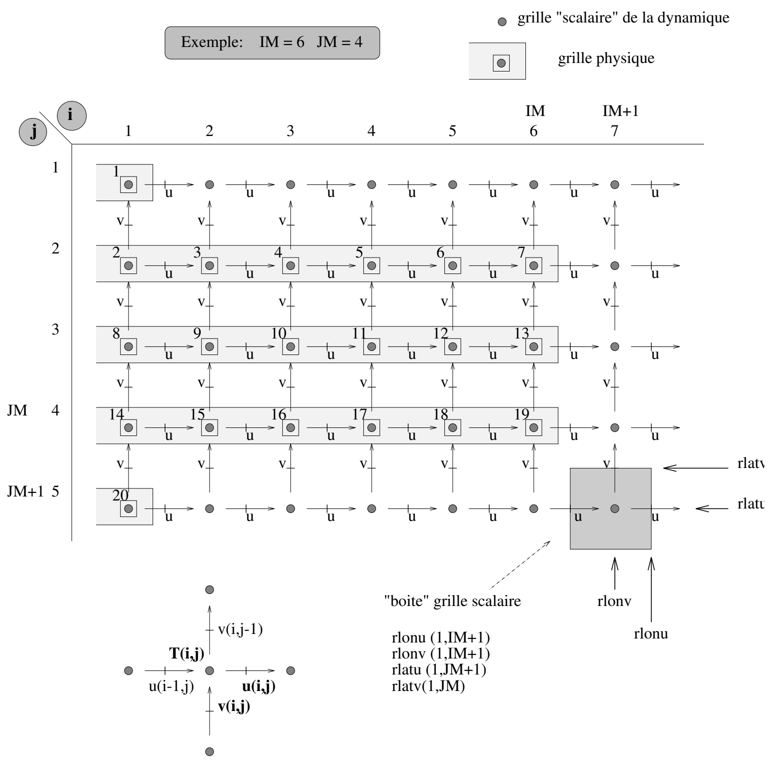


Figure 39: Dynamical and physical grids for a 6×7 horizontal resolution

6.1.3.2 Vertical grids The vertical grids are usually defined in hybrid coordinates (see fig. 40), which uses $\sigma = \frac{p}{ps=\text{surface pres.}}$ near the surface and gradually shifts to p coordinates with increasing altitude. For a height of 80 km, 25 levels are usually used, 32 levels for 120 km, and 50 for simulations rising up to the thermosphere. By setting `hybrid` to `False` in `run.def`, the model will only use σ coordinates.

Pressures at the interlayers `plev` and at the middle of the layers `plav` are given by `plev=ap + bp*ps` and `plav=aps + bs*ps`, where `ap`, `bp`, and `aps`, `bps` indicate the hybrid levels at the interlayer levels and at the middle of the layers, respectively. Sigma coordinates use `aps=0` and `bps=p/ps`, but `bps=0` above $\sim$ 50 km, giving purely pressure levels.

[14] Examples of typical grid values are longitude $\times$ latitude $\times$ altitude $= 64 \times 48 \times 25, 64 \times 48 \times 32$, or $32 \times 24 \times 25$. For Mars (radius $\sim$ 3400 km), a 64×48 horizontal grid corresponds to grid boxes of lengths 330 $\times$ 220 km near the equator.

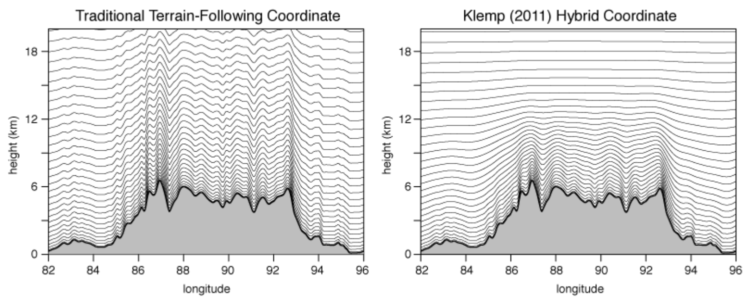


Figure 40: Example of non-hybrid and hybrid coordinates from MPAS–(Skamarock 2020)

6.1.4 A practice simulation

To start a simulation after installing the LMDZ GCM (see sec. 10.3) and moving the executable e.g. `gcm_64x48x29_phymars_seq.e` to the directory `~/Desktop/lmd/gcm_64x48x29_phymars_seq`, four files are still required to be copied to this same directory:

- `run.def` containing the simulation options
- `callphys.def`, containing the general options of the model
- `z2sig.def` containing the [uneven] vertical distribution of the atmospheric layers
- `traceur.def` containing the number of tracers (e.g. `7` in line 1) and the type (e.g. `co2`, `dust_mass`, `dust_number`, `h2o_vap`, `h2o_ice`, `ccn_mass`, `ccn_number`)

The contents of these files have to be *carefully* modified for the simulation to be carried out.

Example files can be found in `~/Desktop/lmd/trunk/LMDZ.MARS/deftank` (local) or `~/Models/trunk/LMDZ.MARS/deftank` (Hyperion2).

The initial condition files should also be in the same directory:

- `start.nc` containing the initial states of the dynamical variables
- `startfi.nc` containing the initial states of the physical variables

Example files can be retrieved form the LMDZ repository after converting the downloaded `start_archive` by means of the program `newstart` as explained in Millour and Forget (2018). If `newstart` is already compiled in the system (see sec. 10.3.7.1 for installation instructions), one can run e.g.:

1. `cd ~/Desktop/lmd/gcm_64x48x29_phymars_seq`
2. `wget https://www.lmd.jussieu.fr/~lmdz/planets/mars/starts/start_archive_64x48x49_MY27.nc` to download `start_archive.nc`
3. Rename this file to `start_archive.nc` via `mv start_archive_64x48x49_MY27.nc start_archive.nc` (`./newstart.e` specifies that it looks for `./start_archive.nc`)
4. Place `newstart.e` in '~/Desktop/lmd/gcm_64x48x29_phymars_seq"
5. Execute `./newstart.e`, which had been
    1. Choose option 0 to create the new initial state from `start_archive.nc`
    2. Follow the instructions presented in the terminal
6. Rename `restart.nc` to `start.nc` via `mv restart.nc start.nc`
7. Rename `restartfi.nc` to `startfi.nc` via `mv restart.nc startfi.nc`

8. Run `./gcm_64x48x29_phymars_seq.e`
9. Visualize results in `diagfi.nc` (instantaneous values) of `stats.nc` (statistics of the run) with ParaView (see sec. 6.1.7)

To run the program, one has to basically call `gcm_64x48x29_phymars_seq.e` (or `gcm.e`) with the files aforementioned in the same directory. To keep the log, the following can be used `gcm.e > gcm.out 2>&1`. If the error `error while loading shared libraries: libnetcdf.so.4: cannot open shared object file: No such file or directory`, this means that NetCDF is not in the path. One can add the line `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/netcdf/lib`[15] to `~/.bash_profile` or `~/.bashrc`, or run the line every time before launching the GCM, or run the GCM through a Bash script (see lst. 1)[16].

Listing 1: Script to launch any GCM

```
#!/bin/bash

NETCDF_LIBDIR=$HOME/Desktop/lmd/netcdf-4.0.1/lib

if [[ "$2" != "" ]];  then
    NETCDF_LIBDIR=$2
fi

echo 'NetCDF lib is in' ${NCDF_LIB}
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NETCDF_LIBDIR

echo 'Running ' ./$1
./$1 2>&1 | tee gcm.out
```

6.1.5 Organization of the code

The model is split between two directories:

- `LMDZ.COMMON` where the dynamics module is found (named `COMMON` because it is common to all planets). It is made up of:
  - `LMDZ.COMMON/libf` includes the source code of phenomena common to all planets
    * `LMDZ.COMMON/libf/grid` contains the grid description
    * `LMDZ.COMMON/libf/dyn3d` contains the code for the dynamical core, which shares the routines in `LMDZ.COMMON/libf/dyn3d_common` with the parallel version of the core, found in `LMDZ.COMMON/libf/dyn3dpar`
    * `LMDZ.COMMON/libf/dynphy_lonlat` contains `phymars` and the interfaces between the physics and dynamics core.
    * `LMDZ.COMMON/libf/filtrez` includes filters for polar latitudes to ensure numeric stability
    * `LMDZ.COMMON/libf/phy_common` contains routines to all physics packages
    * `LMDZ.COMMON/libf/aeronomars` and `LMDZ.COMMON/libf/phymars` link to `LMDZ.MARS/libf/aeronomars` and `LMDZ.MARS/libf/phymars`, respectively
    * `soilwater.F90` is included in `LMDZ.COMMON/libf/phymars`

[15] `/path/to/netcdf/lib` may be `/usr/lib` if NetCDF has been installed via the package manager (see sec. 10.3.4), `~/netcdf/lib` on Hyperion2 if sec. 10.3.2 has been followed, or something like `~/Desktop/lmd/netcdf-4.0.1/lib` according to the sec. 10.3.1.

[16] The installation method shown in sec. 10.3.1 also runs a benchmark test. This test is launched using the script in `~/Desktop/lmd/bench_64x48x29` named `run_gcm.sh`, on which lst. 1 is based.

  - `LMDZ.COMMON/bin` where the executables generated by `makelmdz_fcm` are placed
  - the script to compile the GCM `makelmdz_fcm`
  - `LMDZ.COMMON/arch`, `ioipsl`, `libo`,...

- `LMDZ.MARS` where the Martian physics are described. It is made up of:
  - `LMDZ.MARS/libf` includes the source code for Martian-only phenomena and legacy files which should not be used (see below)
    * `LMDZ.MARS/libf/phymars` contains the Martian physics routines
    * `LMDZ.MARS/libf/aeronomars` contains the Martian chemistry and thermosphere routines
    * `LMDZ.MARS/libf/dyn3d`, `LMDZ.MARS/libf/grid`, `LMDZ.MARS/libf/filtrez` should not be used in favour of the versions in `LMDZ.COMMON/libf`
  - `LMDZ.MARS/deftank` includes a collection of examples of parameter files required to run the GCM (see sec. 6.1.4)
  - `LMDZ.MARS/util` contains postprocessing tools
  - A *legacy* GCM build tool named `makegcm`

6.1.6 Coding conventions for LMDZ

The following good practices should be followed:

- The code is to be written in fixed-form Fortran (`.F` files) or free-form Fortran (`.F90` files).
- Constants are to be placed in `COMMON` declarations, in common *include* files (libraries/headers, `.h` files) or modules (in `/LMDZ.COMMON/libf` **? TODO**)
- In general, variables should be passed from subroutine to subroutine as arguments (never as blocks)
- For historical reasons, a variable e.g. `name` should be named `pname` if passed as an argument by the calling program, and `zname` if they are local variables (i.e. name of the variable in the definition of the routine)

6.1.7 Input/Outputs

LMDZ GCM input and output data use the NetCDF file format. To visualize NetCDF files use ParaView or any of the following alternatives:

- GrADS[17]
  - To install GrADS from the source code follow this or try:
    1. Download the code from here: `wget ftp://cola.gmu.edu/grads/2.2/grads-2.2.0-src.tar.gz`
    2. Extract the contents in `~/grads-2.2.1`
    3. `cd ~/grads-2.2.1`
    4. Create a symbolic link to `/usr/lib` with `ln -s /usr/lib /home/aeronau/grads-2.2.1/lib`
    5. Run `./configure`

[17] In Manjaro `grads` can be installed from the Arch User Repository using `pamac build grads`. Even fixing the Java installation using `archlinux-java set java-13-openjdk`, the installation did not succeed. The program had to be compiled from source.

6. `make`
7. The binaries are in `~/grads-2.2.1/bin`

- GMT[18]
- Panoply
- ncview[19]
  - To view the file run `ncview diagfi.nc`
- Ferret
  - To install follow this or this

As seen in sec. 6.1.4, the LMDZ GCM requires the input four ASCII text `.def` files and two NetCDF `.nc` files:

- Parameter files:
  - `run.def` containing the parameters of the dynamical part of the program, and the temporal integration of the model
  - `callphys.def` containing the parameters for calling the physical part
  - `traceur.def` containing the names of the tracers to use
  - `z2sig.def` containing the vertical distribution of the atmospheric layers
- Initialization files:
  - `start.nc` containing the initial states of the dynamical variables
  - `startfi.nc` containing the initial states of the physical variables

6.1.7.1 Input and parameter files

6.1.7.1.1 The `run.def` file The file is quite straightforward and flexible. Variables are commented or self explanatory, e.g. `nday` is the number of modeled days to run. The file `callphys.def` (see sec. 6.1.7.1.2) is included in this file using `INCLUDEDEF=callphys.def`.

Herebelow are some notes about the variables:

- A low value for `day_step` is 480, with the model's CFL stability improving the larger the number is.
- `tetagdiv`, `tetagrot`, `tetatemp` control the dissipation intensity. They should neither be too low or too high (condition for `idissip` also applies for `tetagrot` and `tetatemp` **?TODO**). Example used with `nitergdiv=1` and `nitergrot=2=niterh=2` :
  - using the 32-by-24 grid `tetagdiv=6000 s ; tetagrot=tetatemp=30000 s`
  - using the 64-by-48 grid: `tetagdiv=2500 s ; tetagrot=tetatemp=5000 s`
- Dissipation is computed and added every `idissip` dynamical time step and should verify `idissip< tetagdiv*daystep/88775`
- Physical tendencies/the physical core is computed every `iphysiq` dynamical timestep. Normally `iphysiq=day_step/48`

[18] In Manjaro, install with `pamac build gmt`

[19] In Manjaro, install with `pamac build ncview` and probably `pacman -S xorg-fonts-misc`, `pacman -S xorg-fonts-75dpi`, `pacman -S xorg-fonts-100dpi`

6.1.7.1.2 The `callphys.def` file The functions in `soilwater.F90`—which are the object of study for this stage—are called when the following is set in `callphys.def`: `callsoilwater=.true.`. The output of the physical module is written every `ecritphys` timesteps, i.e. `ecritphys=20` will force outputs every 20 dynamical timesteps (see sec. 6.1.7.2.2).

The routine that reads the parameters in `callphys.def` is `conf_phys()` found in `conf_phys.F`, which includes the variables in `callkeys.h` (the flag `callsoilwater` is listed in this header).

6.1.7.1.3 The `traceur.def` file The first line contains the number of tracers to load and use, followed by a list with the tracer names (one per line). If there are no tracers in the input files, the tracers are then initialized to zero.

`{#lst:traceur_def example .bash caption="Example of a traceur.def file: with CO2, dust distribution moments, (water ice)cloud con- densation nuclei moments, water vapour and water ice tracers"} 7 co2 dust_number dust_mass ccn_number ccn_mass h2o_ice h2o_vap`

Tracers in the input and output files are stored using these tracer names.

6.1.7.1.4 The `z2sig.def` file Contains the pseudo-altitudes (in km) at which the user wants to set the vertical levels. It is recommended to use the altitude levels of `z2sig.def` located in `LMDZ.MARS/deftank`.

6.1.7.1.5 The initialization files—`start.nc` and `startfi.nc` files They are constituted by:

- A header with a *control* variable followed by a series of variables defining the grid
- A series of non temporal variables that give information about surface conditions on the planet
- A *time* variable for time-dependant files, set to 0 for state files

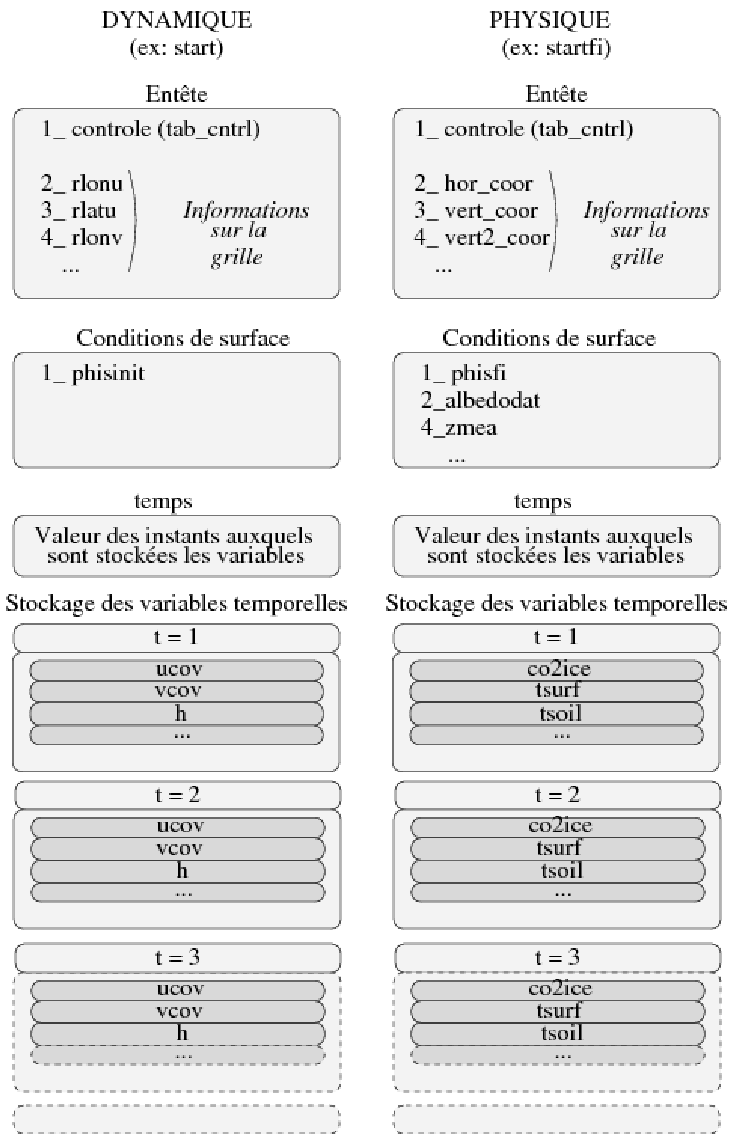


Figure 41: `start.nc` and `startfi.nc` file structure

6.1.7.2 Output files

6.1.7.2.1 The restart files—`restart.nc` and `restartfi.nc` files These allow for a restart of the simulation and follow the same structure as `start.nc` and `startfi.nc` file (see sec. 6.1.7.1.5).

6.1.7.2.2 The `digafi.nc` file This file stores the instantaneous values of physical variables throughout the siulation at a regular interval (set by `ecritphy` in `callphys.def`—see sec. 6.1.7.1.2—and multiple of `iphysiq` and divisor of `day_step`). Constituted by:

- Dimensions
- A *time* variable
- A *control* variable containing parameters
- A list of data describing geometrical coordinates of the data and the surface topography (form `rhonu` to `phisinit`)
- All the 2D or 3D data stored in the run

6.1.7.2.3 The `stats.nc` file Enabled in `callphys.def` via `stats=.true.`, the `stats.nc` file—similar in structure to `diagfi.nc`—contains statistical information of the whole simulation.

6.2 Current status of the subsurface module and environment

P.-Y. Meslin provided a version of the LMDZ GCM which accounts for the impact of subsurface ice on the atmosphere. The subsurface model is coded entirely in `soilwater.F90` (see sec. 6.2.2) and it is called whenever `callsoilwater = .true.` (and `water = .true` **TODO: And maybe more?**) in `callphys.def` (see sec. 6.1.7.1.2). The direct call to `soilwater()` is carried out in the subroutine `vdifc()` of `vdifc_mod.F90`, which performs operations such as the initializatin of the `exchange` array.

6.2.1 Compilation of the custom LMDZ GCM

P.-Y. Meslin supplied a custom version of the LMDZ GCM with support for the subsurface (`soilwater.F90`) module. The model is stored in the file `aprat-gasull@hyperion2:/home/STAGIAIRES/aprat-gasull/Models/model_soilwater.tar.gz` in Hyperion2. Obviously, this version contains the `soilwater.F90` file. Note that this `tar.gz` file also contains the script `install_model_soilwater.sh` for the quick install (sec. 6.2.1.1), which was not provided. The difference between `model_soilwater.tar.gz` and `model_soilwater_julian.tar.gz`, is that the first is bundled with the quick install script aforementioned. This script was created to easen the installation of the model.

6.2.1.1 Quick install of the custom LMDZ GCM After downloading `model_soilwater.tar.gz` to `$HOME`, the model can be installed using the `install_model_soilwater.sh` Bash script:

1. Extract the contents using `tar -xvf model_soilwater.tar.gz`. The directory `~/model_soilwater` will appear.
2. `cd model_soilwater`
3. `chmod +x install_model_soilwater.sh`
4. `./install_model_soilwater.sh`

```
# If you have model_soilwater_julian.tar.gz:
#
# 1. Extract the contents of model_soilwater_julian.tar.gz.
# 2. Place this script inside the directory model_H20, along with the directories UTIL,
   LMDZ.COMMON, LMDZ.MARS, and datagcm.
# 3. Open a terminal in model_H20 and run chmod +x install_model_soilwater.sh
# 4. Run this script with ./install_model_soilwater.sh in the terminal opened before
#
# If you have model_soilwater.tar.gz:
#
# 1. Extract the contents of model_soilwater.tar.gz.
# 2. Find this script inside the directory model_soilwater, .
# 3. Open a terminal in model_soilwater and run chmod +x install_model_soilwater.sh
# 4. Run this script with ./install_model_soilwater.sh in the terminal opened before
#
# This will replace the old variables of Julian's installation (i.e. paths with /home/
   julian/Documents/Uni/Toulouse/Internship/trunk/) and will hopefully recompile the
   model
# This only works for Mars
# Requirements:
# - GNU sed
```

```
BASE=$PWD # i.e. BASE=$HOME/model_H2O
echo $BASE

cd $BASE/UTIL
# Download and install NetCDF 4.0.1
./install_netcdf4.0.1.bash

# Make FCM available
export PATH=$PATH:$BASE/UTIL/FCM_V1.2/bin

# Delete old model compilations
rm -r $BASE/LMDZ.COMMON/libo

cd $BASE/LMDZ.COMMON/ioipsl/modipsl/util
cp AA_make.gdef AA_make.gdef.bak
# Restore original variables for install_ioipsl_gfortran.bash with the following
sed -i -e s:"gfortran  NCDF_INC = /home/julian/Documents/Uni/Toulouse/Internship/trunk/
    UTIL/netcdf-4.0.1/include":"gfortran  NCDF_INC = /usr/local/include":g AA_make.gdef
sed -i -e s:"gfortran  NCDF_LIB = -L/home/julian/Documents/Uni/Toulouse/Internship/trunk
    /UTIL/netcdf-4.0.1/lib -lnetcdf":"gfortran  NCDF_LIB = -L/usr/local/lib -lnetcdf":g
    AA_make.gdef

cd $BASE/LMDZ.COMMON/ioipsl
cp install_ioipsl_gfortran.bash install_ioipsl_gfortran.bash.bak
# Change setfolder path to NetCDF and avoid downloading the modispl
sed -i -e s:'setfolder=/home/julian/Documents/Uni/Toulouse/Internship/trunk/UTIL/netcdf
    -4.0.1':"setfolder=$BASE/UTIL/netcdf-4.0.1":g install_ioipsl_gfortran.bash
sed -i -e s:'svn co http\://forge.ipsl.jussieu.fr/igcmg/svn/modipsl/trunk modipsl':'#svn
     co http\://forge.ipsl.jussieu.fr/igcmg/svn/modipsl/trunk modipsl':g
    install_ioipsl_gfortran.bash # sed can use any separator
# Install model
./install_ioipsl_gfortran.bash

cd $BASE/LMDZ.COMMON/libf
# Repair symlinks
ln -rsf ../../LMDZ.MARS/libf/phymars
ln -rsf ../../LMDZ.MARS/libf/aeronomars
ln -rsf ../../LMDZ.MARS/libf/dynphy_lonlat/phymars dynphy_lonlat/phymars

cd $BASE/LMDZ.COMMON/arch
# Change compilation variables
cp arch-gfortran.env arch-gfortran.env.bak
sed -i -e s:'mpif90':'gfortran':g arch-gfortran.fcm
sed -i -e s:'export NETCDF=/home/julian/Documents/Uni/Toulouse/Internship/trunk/UTIL/
    netcdf-4.0.1/':"export NETCDF=$BASE/UTIL/netcdf-4.0.1/":g arch-gfortran.env

cd $BASE/LMDZ.COMMON
# Compile sample model
./makelmdz_fcm -arch gfortran -d 64x48x29 -p mars gcm
```

6.2.1.2 Manual restoration of the custom LMDZ GCM provided by P.-Y. Meslin

> The following describes the steps in lst. 6.2.1.1, which already contains brief comments.

For the model to be compiled, the NetCDF and the IOIPSL libraries need to be compiled. The former is downloaded and compiled by running the script `~/model_soilwater/UTIL/install_netcdf4.0.1.bash`. The IOIPSL libary is compiled by executing the file `~/model_soilwater/LMDZ.COMMON/ioipsl/install_ioipsl_gfortran.bash` after changing `setfolder=$HOME/model_soilwater/UTIL/netcdf-4.0.1` in `~/model_soilwater/LMDZ.COMMON/ioipsl/install_ioipsl_gfortran.bash` and modifying `~/model_soilwater/LMDZ.COMMON/ioipsl/modipsl/util/AA_make.gdef` to include the following:

```
#-Q- gfortran  NCDF_INC = /usr/local/include
```

```
#-Q- gfortran  NCDF_LIB = -L/usr/local/lib -lnetcdf
```

After this modification, `install_ioipsl_gfortran.bash` can be executed.

In order to compile the custom versions of the GCM and noGCM, the `fcm` tool needs to be available. Therefore, before calling e.g. `./makelmdz_fcm -arch gfortran -p mars -d 64x48x49 gcm` or `./makelmdz_fcm -arch gfortran -p mars -d 64x48x4 nogcm`, one needs to run `export PATH=$PATH:$HOME/model_soilwater/UTIL/FCM_V1.2/bin`.

Ensure that all previous compilations from other machines are deleted using `rm -rf ~/model_soilwater/LMDZ.COMMON/libo/*`.

Symbolic links may need to be restored. From `~/model_soilwater/LMDZ.COMMON/arch`, run `ln -rsf ../../LMDZ.MARS/libf/phymars`, `ln -rsf ../../LMDZ.MARS/libf/aeronomars`, and `ln -rsf ../../LMDZ.MARS/libf/dynphy_lonlat/phymars dynphy_lonlat/phymars`.

The `NETCDF` environment variable must be set to `~/model_soilwater/UTIL/netcdf-4.0.1` in `~/LMDZ.COMMON/arch/arch-gfortran.env`.

The model can now be compiled using i.e. `./makelmdz_fcm -arch gfortran -d 64x48x29 -p mars gcm`.

6.2.2 The `soilwater.F90` file

The subroutine `soilwater()` calculates the profile of water vapor, and the adsorbed water and ice in the regolith down to `depth(nsoil)` metres (e.g. $\sim$ 18 m) according to an improved version of the equations developed in sec. 5.2.1.

Extracted from the source file:

> The linearized adsorption–diffusion equation (see eq. 14) is solved first, then the water vapor pressure is compared to the saturation vapor pressure, and if the latter is reached, ice is formed, the water vapor pressure is fixed to its saturation value and the adsorption–diffusion equation is solved again. If ice was already present, its sublimation or growth is calculated.
>
> It returns the flux of water vapor that is injected into or out of the regolith, and which serves as a boundary condition to calculate the vertical atmospheric profile of water vapor in `vdifc()`. It also calculates the exchange between the subsurface and surface ice.
>
> It requires three subsurface tracers to be defined in `traceur.def`:
>
> - `h2o_vap_soil` (subsurface water vapor)
> - `h2o_ice_soil` (subsurface ice)
> - `h2o_ads_vap` (adsorbed water)

The following are snippets of a commented version of `soilwater.F90`:

Listing 2: Variables used in the Fortran subsurface model

```
! Arguments :
! ======================================================================

! Inputs :
! ----------------------------------------------------------------------
```

```
integer, intent(in) :: ngrid                    ! number of points in the model (all lat and long point in one 1D array)
integer, intent(in) :: nlayer                   ! number of layers
integer, intent(in) :: nq                       ! number of tracers
integer, intent(in) :: nsoil
integer, intent(in) :: nqsoil
logical, save :: firstcall = .true.             ! triggers the initialization
real, intent(in) :: tsoil(ngrid, nsoil)         ! Subsurface temperatures
real, intent(in) :: ptsrf(ngrid)                ! Surface temperature
real, intent(in) :: ptimestep                   ! length of the timestep (unit depends on run.def file)
logical, intent(in) :: exchange(ngrid)          ! logical :: array describing whether there is exchange with the atmosphere at the current timestep

real, intent(in) :: P_sat_surf(ngrid)           ! Saturation vapor pressure at the current surface temperature (formerly qsat)
real, intent(in) :: pq(ngrid, nlayer, nq)       ! Tracer atmospheric mixing ratio
real, intent(in) :: pa(ngrid, nlayer)           ! Coefficients za
real, intent(in) :: pb(ngrid, nlayer)           ! Coefficients zb
real, intent(in) :: pc(ngrid, nlayer)           ! Coefficients zc
real, intent(in) :: pd(ngrid, nlayer)           ! Coefficients zd
real, intent(in) :: pdqsdifpot(ngrid, nq)       ! Potential pdqsdif (without subsurface - atmosphere exchange)

real, intent(in) :: pplev(ngrid, nlayer+1)      ! Atmospheric pressure levels
real, intent(in) :: rhoatmo(ngrid)              ! Atmospheric air density (1st layer) (not used right now)
real, intent(in) :: co2ice(ngrid)               ! CO2 ice on the surface

! Variables modified :
! ----------------------------------------------------------------------
real, intent(inout) :: pqsoil(ngrid, nsoil, 3*nq) ! Subsurface tracers (water vapor and ice)
real, intent(in) :: pqsurf(ngrid, nq)           ! Water ice on the surface
                                                      ! (in kg.m - 3 : m - 3 of pore air
                                                          for water vapor and m - 3 of
                                                         regolith for water ice and
                                                         adsorbed water)
! Outputs :
! ----------------------------------------------------------------------
real, intent(out) :: zdqsdifrego(ngrid)         ! Flux from subsurface (positive pointing outwards)
real, intent(out) :: zq1temp2(ngrid)            ! Temporary atmospheric mixing ratio after exchange with subsurface (kg / kg)

! Outputs for the output files
real*8 :: preduite(ngrid, nsoil)                ! how close the water vapor density is to adsorption saturation
integer :: exch(ngrid)                          ! translates the logical variable exchange into a -1 or 1
real*8 :: mass_h2o(ngrid)                       ! Mass of subsurface water column at each point (kg.m - 2) (formerly mh2otot)
real*8 :: mass_ice(ngrid)                       ! Mass of subsurface ice at each point (kg.m - 2) (formerly micetot)
real*8 :: mass_h2o_tot                          ! Mass of subsurface water over the whole planet
real*8 :: mass_ice_tot                          ! Mass of subsurface ice over the whole planet
real*8 :: nsurf(ngrid)                          ! surface tracer density in kg/m^3
real*8 :: close_out(ngrid, nsoil)           ! output for close_top and close_bottom

! Local (saved) variables :
! ======================================================================

real*8 :: P_sat_soil(ngrid, nsoil)              ! water saturation pressure of subsurface cells (at miD-layers) (formerly Psatsoil)
```

```fortran
real*8 :: nsatsoil(ngrid, nsoil)                    ! gas number density at saturation pressure
real*8, allocatable, save :: znsoil(:, :)          ! Water vapor soil concentration (kg.m - 3 of pore air)
real*8 :: znsoilprev(ngrid, nsoil)                  ! value of znsoil in the previous timestep
real*8 :: znsoilprev2(ngrid, nsoil)                 ! second variable for the value of znsoil in the previous timestep
real*8 :: zdqsoil(ngrid, nsoil)                     ! Increase of pqsoil if sublimation of ice
real*8, allocatable, save :: ice(:, :)             ! Ice content of subsurface cells (at miD-layers) (kg / m3)
real*8 :: iceprev(ngrid, nsoil)
logical :: ice_index_flag(nsoil)                    ! flag for ice presence
real*8, allocatable, save :: adswater(:, :)        ! Adsorbed water in subsurface cells (at miD-layers) (...)
real*8 :: adswater_temp(ngrid, nsoil)               ! temprory variable for adsorbed water
logical :: ads_water_saturation_flag_2(nsoil)
real*8 :: adswprev(ngrid, nsoil)
logical :: ads_water_saturation_flag_1(nsoil)
real*8, save :: adswater_sat                        ! Adsorption saturation value (kg.m - 3 of regolith)
real*8 :: delta0(ngrid)                             ! Coefficient delta(0) modified
real*8 :: alpha0(ngrid)
real*8 :: beta0(ngrid)

! Adsorbtion/Desorption variables and parameters
real*8 :: Ka(ngrid, nsoil)                ! Adsorption time constant (s - 1)
real*8 :: Kd(ngrid, nsoil)                ! Desorption time constant (s - 1)
real*8 :: k_ads_eq(ngrid, nsoil)          ! Equilibrium adsorption coefficient (unitless) (formerly kads)
real*8, parameter :: kinetic_factor = 1      ! (inverse of) Characteristic time to reach adsorption equilibrium (s - 1):
                                                    ! fixed arbitrarily when kinetics factors are unknown
                                                    ! possible values: ! = 1 / 1800 s - 1 ! / 1.16D-6 /  !( =  10 earth days)! / 1.D0 /  ! / 1.2D-5 /  !

real*8, allocatable, save :: ztot1(:, :)  ! Total (water vapor +  ice) content (kg.m - 3 of soil) !? why does this not include adsorbed water?
real*8 :: dztot1(nsoil)
real*8 :: h2otot(ngrid, nsoil)      ! Total water content (ice +  water vapor + adsorbed water) (....)
real*8 :: flux(ngrid, 0:nsoil - 1)  ! Fluxes at interfaces (kg.m - 2.s - 1) (positive = upward)
real*8 :: rho(ngrid)                ! Air density (first subsurface layer)
real*8 :: rhosurf(ngrid)            ! Surface air density


! Porosity and tortuosity variables
real*8, allocatable, save :: porosity_ice_free(:, :)  ! Porosity with no ice present (formerly eps0)
real*8, allocatable, save :: porosity(:, :)           ! Porosity (formerly eps)
real*8 :: porosity_prev(ngrid, nsoil)                  ! Porosity from previous timestep
real*8, allocatable, save :: tortuosity(:, :)         ! Tortuosity factor (formerly tortuo)

real*8 :: saturation_water_ice(ngrid, nsoil)           ! Water pore ice saturation level (formerly Sw)
real*8 :: saturation_water_ice_inter(ngrid, nsoil)           ! Water pore ice saturation level at the interlayer

! Diffusion coefficients
real*8 :: D_mid(ngrid, nsoil)       ! Midlayer diffusion coefficients
real*8 :: D_inter(ngrid, nsoil)     ! Interlayer diffusion coefficients (formerly D)
```

```fortran
real*8, allocatable, save :: D0(:, :)      ! Diffusion coefficient prefactor :
                                           ! If (medium = 1) : D0 = value of D_mid for
                                             saturation_water_ice = 0, ie. poro /
                                             tortuo * Dm (in m2 / s)
                                           ! If (medium = 2) : D0 = porosity_tortuosity
                                             factor (dimensionless)
real*8 :: omega(ngrid, nsoil)        ! H2O - CO2 collision integral
real*8 :: vth(ngrid, nsoil)          ! H2O thermal speed
real*8, parameter :: Dk0 = 0.459D0   ! Knudsen diffusion coefficient (for
    saturation_water_ice = 0) Value for a (5 / 1) bidispersed random assembly of spheres
                                     ! (dimensionless, ie. divided by thermal speed and
                                       characteristic meshsize of the porous medium)
real*8 :: Dk(ngrid, nsoil)           ! Knudsen diffusion coefficient (divided by porosity
     / tortuosity factor)
real*8 :: Dk_inter(ngrid, nsoil)           ! Knudsen diffusion coefficient at the
    interlayer
real*8 :: Dm(ngrid, nsoil)           ! Molecular diffusion coefficient
real*8 :: Dm_inter(ngrid, nsoil)           ! Molecular diffusion coefficient at the
    interlayer

real*8, parameter :: choke_fraction = 0.8D0  ! fraction of ice that prevents further
    diffusion
logical, allocatable, save :: close_top(:, :)          ! closing diffusion at the top of
    a layer if ice rises over saturation
logical, allocatable, save :: close_bottom(:, :)       ! closing diffusion at the bottom
    of a layer if ice risies over saturation
logical, parameter :: print_closure_warnings = .true.

! Coefficients for the Diffusion calculations
real*8 :: A(ngrid, nsoil)            ! Coefficient in the diffusion formula
real*8 :: B(ngrid, nsoil)            ! Coefficient in the diffusion formula
real*8 :: C(ngrid, nsoil)            ! Coefficient in the diffusion formula
real*8 :: E(ngrid, nsoil)            ! Coefficient in the diffusion formula
real*8 :: F(ngrid, nsoil)            ! Coefficient in the diffusion formula
real*8, allocatable, save :: zalpha(:, :) ! Alpha coefficients
real*8, allocatable, save :: zbeta(:, :)  ! Beta coefficients
real*8 :: zdelta(ngrid, nsoil - 1)        ! Delta coefficients

! Distances between layers
real*8, allocatable, save :: interlayer_dz(:, :)       ! Distance between the interlayer
    points in m (formerly interdz)
real*8, allocatable, save :: midlayer_dz(:, :)         ! Distance between the midcell
    points in m (formerly middz)

real*8 :: nsat(ngrid, nsoil)                     ! amount of water vapor at adsoption
    saturation

real*8, allocatable, save :: meshsize(:, :)      ! scaling/dimension factor for the por
    size
real*8, allocatable, save :: rho_soil(:, :)      ! Soil density (bulk -  kg / m3) (
    formerly rhosoil)
real*8, allocatable, save :: cste_ads(:, :)      ! Prefactor of adsorption coeff. k

! general constants
real*8, parameter :: kb = 1.38065D-23     ! Boltzmann constant
real*8, parameter :: mw = 2.988D-26     ! Water molecular weight
!real*8, parameter :: rho_H2O_ice = 920.D0    ! Ice density

! adsorption coefficients
real*8, parameter :: enthalpy_ads = 21.D3 ! Enthalpy of adsorption (J.mole - 1) options
    21.D3, 35.D3, and 60.D3
real*8, parameter :: tau0 = 1D-14
real*8, parameter :: S = 17.D3            ! Soil specific surface area (m2.kg - 1 of
    solid) options: 17.D3 and 106.D3
real*8, parameter:: Sm = 10.6D-20         ! Surface of the water molecule (m2) (only
    needed in the theoretical formula which is not used right now)
```

```fortran

integer, parameter :: choice_ads = 1      ! Choice of adsorption - desorption constants 3: no adsorption

! Reference values for choice_ads = 2
real*8, parameter :: Tref = 243.15D0
real*8, parameter :: nref = 2.31505631D-6 ! calculated as 18.D-3 / (8.314D0 * Tref) * 0.26D0
real*8, parameter :: Kd_ref = 0.0206D0
real*8, parameter :: Ka_ref = 2.4D-4
real*8, parameter :: Kref = 1.23D7        ! Volcanic tuff @ 243.15 K (obtained at low P / Psat)

logical :: saturation_flag
logical :: sublimation_flag
logical :: condensation_flag(nsoil)

! variables and parameters for the H2O map
integer, parameter :: n_long_H2O = 66!180          ! number of longitudes of the new H2O map
integer, parameter :: n_lat_H2O = 50 !87            ! number of latitudes of the new H2O map
real*8, parameter :: rho_H2O_ice = 920.0D0      ! Ice density (formerly rhoice)
real :: latH2O(n_lat_H2O*n_long_H2O)            ! Latitude at H2O map points
real :: longH2O(n_lat_H2O*n_long_H2O)           ! Longitude at H2O map points
real :: H2O_depth_data(n_lat_H2O*n_long_H2O)    ! depth of the ice layer in in g/cm^2 at H2O map points
real :: H2O_cover_data(n_lat_H2O*n_long_H2O)    ! H2O content of the cover layer at H2O map points (not used yet)
real :: dataH2O(n_lat_H2O*n_long_H2O)           ! H2O content of the ice layer at H2O map points
real :: latH2O_temp(n_lat_H2O*n_long_H2O)       ! intermediate variable
real :: longH2O_temp(n_lat_H2O*n_long_H2O)      ! intermediate variable
real :: dataH2O_temp(n_lat_H2O*n_long_H2O)      ! intermediate variable
real :: H2O_depth_data_temp(n_lat_H2O*n_long_H2O)     ! intermediate variable
real, allocatable, save :: H2O(:)                           ! H2O map interpolated at GCM grid points (in wt%)
real, allocatable, save :: H2O_depth(:)                     ! H2O map ice depth interpolated at GCM in g/cm^2

! variables for the 1D case
real*8, parameter :: mmr_h2o = 0.6D-4     ! Water vapor mass mixing ratio (for initialization only)

real*8 :: diff(ngrid, nsoil)              ! difference between total water content and ice + vapor content
                                          ! (only used for output, inconstistent: should just be adswater)
real :: var_flux_soil(ngrid, nsoil)

! Loop variables and counters
integer :: ig, ik, i, j                   ! loop variables
logical :: output_trigger                 ! used to limit amount of written output
integer, save :: n                        ! number of runs of this subroutine
integer :: sublim_n                       ! number of sublimation loop runs
integer :: satur_n                        ! number of saturation loop runs
```

The file is constituted by different loops, which are explained below

6.2.2.1 The `firstcall` loop When `firstcall .eqv. .true.`, a loop on the `ngrid` columns—corresponding to the `ngrid` points in the physical grid—is carried out (see fig. 42) using the variable `ig`. Note that `layer` and `mlayer` are imported from the `comsoil_h` module along other variables (see lst. 3).

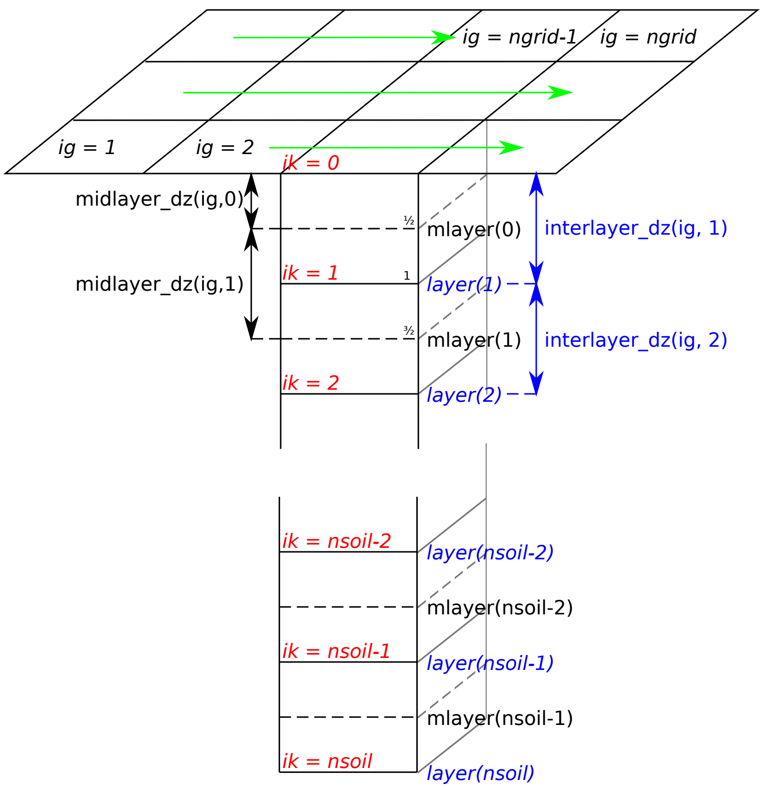


Figure 42: Grid and variables used by `soilwater()`

Listing 3: soilwater subroutine declaration and module imports

```fortran
subroutine soilwater(ngrid, nlayer, nq, tsoil, ptsrf, ptimestep,  &
      exchange, P_sat_surf, pq, pa, pb, pc, pd, pdqsdifpot, pqsurf,  &
      pqsoil, pplev, rhoatmo, co2ice, zdqsdifrego, zq1temp2, nsoil, nqsoil)


      use comsoil_h, only: igcm_h2o_vap_soil, igcm_h2o_ice_soil, igcm_h2o_vap_ads, layer
          , mlayer
      use comcstfi_h
      use tracer_mod
      use surfdat_h, only: watercaptag ! use mis par AP15 essai
      use geometry_mod, only: cell_area, latitude_deg

implicit none
```

The `interlayer_dz` and `midlayer_dz` are initialized in lst. 4 (paying special attention to the topmost layer), and are followed in the source code by `porosity_ice_free(ig, ik)= 0.45D0` and `rho_soil(ig, ik)= 1.3 D3`, initialization values of porosity and regolith density that are also used in sec. 3.2.2.

Listing 4: Initialization of soil parameters

```fortran
                  ! initialize the midlayer distances
                  midlayer_dz(ig, 0) = mlayer(0)

                  do ik = 1, nsoil - 1
                        midlayer_dz(ig, ik) = mlayer(ik) - mlayer(ik - 1)
                  enddo

                  ! initialize the interlayer distances
                  interlayer_dz(ig, 1) = layer(1)
                  do ik = 2, nsoil
                        interlayer_dz(ig, ik) = layer(ik) - layer(ik - 1)
```

```
         enddo
```

The source code is full of comments with alternative initialization values. For example, the variable `adswater_sat`, which holds the adsorption saturation value, is currently initialised using a mathematical operation that uses experimental values from Pommerol et al. (2009) of the volcanic tuff, but alternative definitions are provided as comments.

Then, the ice, water vapor, and adsorbed water profiles are initialised for each of the `ngrid` columns. Each of the `nsoil` ground points of a given `ig` column are walked and variables `znsoil`, `ice`, and `adswater` are initialised from `pqsoil`, the variable containing subsurface tracers which was passed from the subroutine caller (note the initial `p`).

For the case of `ngrid.eq.1` (1) column, these variables are initialised as shown in lst. 5 (`choice_ads` is initialised as shown in lst. 2). Note that `porosity` accounts for water content as it is computed subtracting `saturation_water_ice`, the part of water ice that fills the pores, from `porosity_ice_free`, the porosity without ice. Also note that `saturation_water_ice` cannot be greater than 1 as it would mean that more than the available pores are filled with water ice. Therefore, the variable is limited to `0.999D-0` (99%) and equal to $\frac{\frac{m_{\text{ice}}}{m_{\text{regolith}}}}{\rho_{\text{H}_2\text{O ice}}\varepsilon_{\text{ice free}}}$ or equivalently `ice(ig, ik)/ (rho_H2O_ice * porosity_ice_free(ig, ik)`.

Listing 5: Initialization of profiles when there is only one column

```
            if (ngrid.eq.1) then
               znsoil(ig, ik) = mmr_h2o * mugaz * 1.D-3 * pplev(ig, 1) &
               / (8.314D0 * tsoil(ig, nsoil - 4))
               !                 znsoil(ig, ik) = 0.
               ice(ig, ik) = 0.D0  ! 0.1D0 !414.D0

               saturation_water_ice(ig, ik) = min(ice(ig, ik) / (rho_H2O_ice * porosity_ice_free(ig, ik)), 0.999D-0)
               porosity(ig, ik) = porosity_ice_free(ig, ik) * (1.D0 - saturation_water_ice(ig, ik))

               if (choice_ads.eq.1) then
                  vth(ig, ik) = dsqrt(8.D0 * 8.314D0 * tsoil(ig, nsoil - 4) &
                     / (pi * 18.D-3))
                  k_ads_eq(ig, ik) = rho_soil(ig, ik) * S * vth(ig, ik) / 4.D0 &
                     / (dexp(-enthalpy_ads &
                     / (8.314D0 * tsoil(ig, nsoil - 4))) / tau0)
                  Kd(ig, ik) = kinetic_factor  &
                     / (1.D0 + k_ads_eq(ig, ik) / porosity(ig, ik))
                  Ka(ig, ik) = kinetic_factor * k_ads_eq(ig, ik) /  &
                     (1.D0 + k_ads_eq(ig, ik) / porosity(ig, ik))

               elseif (choice_ads.eq.2) then  ! par rapport \E0 valeur experimentale de reference
                  !                                        Kd(ig, ik) = Kd_ref * exp(-enthalpy_ads / 8.314 *
                  !     &                                        (1. / tsoil(ig, nsoil - 4) - 1. / Tref))
                  !                                        Ka(ig, ik) = rho_soil(ig, ik) * Ka_ref *
                  !     &                                        sqrt(tsoil(ig, nsoil - 4) / Tref) / nref

                  k_ads_eq(ig, ik) = Kref * dexp(enthalpy_ads / 8.314D0 * (1.D0 / tsoil(ig, ik) - 1.D0 / Tref))

```

```
                                    Kd(ig, ik) = kinetic_factor / (1.D0 + k_ads_eq(ig,
                                        ik) / porosity(ig, ik))

                                    Ka(ig, ik) = kinetic_factor * k_ads_eq(ig, ik) / (1.
                                        D0 + k_ads_eq(ig, ik) / porosity(ig, ik))
                              endif

                              adswater(ig, ik) = min(Ka(ig, ik) / Kd(ig, ik) * znsoil(ig
                                  , ik), adswater_sat)
```

Water content could also be initialisied from external data `H2O_data`, but the part is currently commented out. At the end of this first loop, variable `firstcall` is set to false.

6.2.2.2 A loop to compute transport coefficients In the second loop—carried out at every timestep—all the `nsoil` points of each of the `ngrid` columns that are not in the polar caps (strictly speaking when `watercaptag(ig).eqv. .true.`) are walked. The aim is to find the diffusion coefficients at the midlayers and the interlayers, named `D_mid` and `D_inter`.

The loop starts by computing the variable `porosity` from values of the previous timestep in a similar fashion as described above in sec. 6.2.2.1. The air density in the first layer `rho(ig)` is computed using the ideal gas law from the atmospheric pressure `pplev(ig, 1)` and the subsurface temperature `tsoil(ig, 1)` of the first layer.

The midlayer diffusion coefficient `D_mid` is computed from the harmonic mean of the Knudsen diffusion and the molecular diffusion, while the interlayer diffusion coefficient `D_inter` is computed from the interpolation of the molecular and Knudsten diffusions (as seen in lst. 6). Variables `close_bottom` and `close_top` are set to true when there is too much ice at the bottom or the top (see sec. 6.2.2.5), respectively (i.e. it is saturated to allow flux, and if it is saturated the value to be set is 99%).

Listing 6: Computation of the interlayer diffusion coefficient

```
             ! calculate the interlayer diffusion coefficient as interpolation of the
               midlayer coefficients
             do ik = 1, nsoil - 1
                   Dm_inter(ig, ik) = ( (layer(ik) - mlayer(ik - 1)) * Dm(ig, ik + 1) &
                         + (mlayer(ik) - layer(ik)) * Dm(ig, ik) ) / (mlayer(ik) - mlayer
                             (ik - 1))

                   Dk_inter(ig, ik) = ( (layer(ik) - mlayer(ik - 1)) * Dk(ig, ik + 1) &
                         + (mlayer(ik) - layer(ik)) * Dk(ig, ik) ) / (mlayer(ik) - mlayer
                             (ik - 1))

!                  saturation_water_ice_inter(ig, ik) = ( (layer(ik) - mlayer(ik - 1)) *
     saturation_water_ice(ig, ik + 1) &
!                        + (mlayer(ik) - layer(ik)) * saturation_water_ice(ig, ik) ) / (
     mlayer(ik) - mlayer(ik - 1))

                   if (close_bottom(ig, ik).or.close_top(ig, ik+1)) then
                         saturation_water_ice_inter(ig, ik) = 0.999D0
                   else
                         saturation_water_ice_inter(ig, ik) = min(saturation_water_ice(ig
                             , ik + 1), saturation_water_ice(ig, ik))  ! new diffusion
                             interaction
                   endif

                   saturation_water_ice_inter(ig, ik) = min(saturation_water_ice(ig, ik +
                        1), saturation_water_ice(ig, ik))  ! new diffusion interaction

```

```
                    D_inter(ig, ik) = D0(ig, ik) * (1.D0 - saturation_water_ice_inter(ig,
                        ik)) ** 2.D0 * 1.D0 / (1.D0 / Dm_inter(ig, ik) + 1.D0 / Dk_inter(
                        ig, ik))

!                    D_inter(ig, ik) = ( (layer(ik) - mlayer(ik - 1)) * D_mid(ig, ik + 1)
    &  ! old implementation with direct interpolation
!                          + (mlayer(ik) - layer(ik)) * D_mid(ig, ik) ) / (mlayer(ik) -
    mlayer(ik - 1))
            enddo
```

6.2.2.3 A loop to adsorption and desorption constants

The adsorption and desorption constants—`Ka` and `Kd`, respectively—are computed for the columns that are not in the polar caps using the same equations presented in sec. 6.2.2.1. With these, constants `C`, `E`, and `F` (which appeared in sec. 5.2.1 as C, G, and H, respectively) can be set (see lst. 7).

Listing 7: C, e, and F coefficients

```
                    ! calculate the amount of water vapor at adorption saturation
                    nsat(ig, ik) = adswater_sat * Kd(ig, ik) / Ka(ig, ik)

                    ! calculate C, E, and F coefficients for later calculations
                    C(ig, ik) = interlayer_dz(ig, ik) / ptimestep
                    E(ig, ik) = Ka(ig, ik) / (1.D0 + ptimestep * Kd(ig, ik))
                    F(ig, ik) = Kd(ig, ik) / (1.D0 + ptimestep * Kd(ig, ik))

                    ! save the old water concentration
                    znsoilprev(ig, ik) = znsoil(ig, ik)
                    znsoilprev2(ig, ik) = znsoil(ig, ik)  ! needed in "special case" loop
                        because adswprev can be changed before this loop
                    adswprev(ig, ik) = adswater(ig, ik)  !!!! Besoin de adswprev2 ???
                    iceprev(ig, ik) = ice(ig, ik)  ! needed in "special case" loop for the
                         same reason
```

In lst. 7, values of the previous timestep are stored to treat the “special case”.

6.2.2.4 The main loop

This is arguably the most complex loop of the subroutine `soilwater()`.

In this loop, the fluxes—defined as positive pointing outwards—named `zdqsdifrego` (subsurface flux) and `flux` (flux at the interfaces) are initially reset to 0. Then, all the `ngrid` columns that are not in the polar caps have the variables `ads_water_saturation_flag_1` and `ads_water_saturation_flag_2` set to `.false.` for their `nsoil` cells, and the variable `ice_index_flag` set to `.true.` or `.false.` whether these contain ice or not.

Afterwards, coefficients `A` and `B` (they appear in the general formula in sec. 5.2.2.2) are computed, along with the saturation pressure `P_sat_soil` and the gas number density `nsatsoil`.

For the surface layer (cell number 0) coefficients `delta0`, `alpha0`, and `beta0` (see eq. 25) are computed from variables $a, b\dots$ named `pa`, `pb`... [respectively], which are passed by the caller (see sec. 6.1.6) `vdifc()` (from `vdifc_mod.F90` or `vdifc.F90`). With the `condensation_flag` set to `.true.` and the `sublimation_flag` set to `.false.`, a `do while()` loop governed by `sublimation_flag` is carried out. Therefore, *while water ice transitions to vapor*, the following is carried out:

1. The coefficients of the first layer $\Delta^t_{\frac{1}{2}}$, α^t_1, and β^t_1 are computed taking into account the cases with (`exchange(ig).eqv. .true.`) and without (`exchange(ig).eqv. .false.`) mass exchange with the atmosphere.

2. If ice is present at the top layer because `ice_index_flag(1).eqv. .true.` the values are set to $\alpha_1^t = 0$ and $\beta_1^t = n^t_{\text{sat},-\frac{1}{2}}$ (sec. 5.2.2.1)
3. A `do ik = 2, nsoil - 1` loop from the second to the penultimate cell is carried out to compute $\Delta^t_{k-\frac{1}{2}}$, α_k^t and β_k^t. Similarly, if ice is present because of `ice_index_flag(ik).eqv. .true.`, then $\alpha_k^t = 0$ and $\beta_k^t = n^t_{\text{sat},k-\frac{1}{2}}$
4. A preliminary computation of `pqsoil` , `ztot1` (total content of water) is carried out... For the bottom layer, if there is ice, $n_N^t = n_{\text{sat},N}$, i.e. `znsoil(ig, nsoil)= nsatsoil(ig, nsoil)`, otherwise eq. 27 is used. The adsorbed water in the bottom layer `adswater_temp(ig, nsoil)` is computed from eq. 18, and with a `do ik = nsoil-1, 1, -1` reverse loop, $n^t_{2,k+\frac{1}{2}}$ (`znsoil(ig, ik)`) and $n^t_{1,k+\frac{1}{2}}$ (`adswater_temp(ig, ik)`) are found for all layers using eq. 19.
5. Checking if there is saturation in a layer can be achieved by looping again over all soil levels. This is checked by comparing the adsorbed water to n_{sat} (the saturation level). If the level becomes saturated—assuming there was no saturation previously (`ads_water_saturation_flag_2(ik).eqv. .false.`)—because the adsorbed water is greater than the saturation level (`adswater_temp(ig, ik).ge.adswater_sat`), the adsorbed water is changed to the max saturation value `adswater(ig, ik)= adswater_sat` and both `ads_water_saturation_flag_1` and `ads_water_saturation_flag_2` are set to true. Coefficients `A` and `B` are changed too. If there is no saturation, the temporary value is assumed as definitive (`adswater(ig, ik)= adswater_temp(ig, ik)`).
6. Loop all over `nsoil` levels to check if any level has become newly saturated, and if any `ads_water_saturation_flag_1(ik).eqv. .true.` is found, then set `saturation_flag = .true.` and reset `ads_water_saturation_flag_1(ik)= .false.`. **Note that at a timestep t, `ads_water_saturation_flag_1` is used to track whether cells transition from a non saturated state to a saturated state during this timestep, while `ads_water_saturation_flag_2` tracks whether the level was saturated in the previous timestep.**

Afterwards, the flux at the top layer is calculated. In lst. 8, `pqsurf(ig, igcm_h2o_ice)` is the quantity of water ice on the surface of a column `ig` in kg/m$^3_{\text{regolith}}$, and if it is greater than 0—meaning there is ice—then the flux is computed from eq. 26. If there is exchange with the atmosphere, the flux $\frac{F}{\Delta t}$ is computed using eq. 25 in eq. 5.2.1.1.1.

Listing 8: C, E, and F coefficients

```
                        ! calulate the flux variable

                        ! calculate the flux in the top layer
                        if (exchange(ig)) then  ! if there is exchange
                              ! calculate the flux taking diffusion to the atmosphere into
                                  account
                              flux(ig, 0) = porosity_ice_free(ig, 1) * pb(ig, 1) / ptimestep &
                                    * ( znsoil(ig, 1) / rho(ig) - beta0(ig) - alpha0(ig) *
                                        znsoil(ig, 1) / rho(ig) )
                        elseif (pqsurf(ig, igcm_h2o_ice).gt.0.) then
                              ! assume that the surface is covered by waterice (if it is
                                  co2ice it should not call this subroutine at all)
                              flux(ig, 0) = D_mid(ig, 1) / midlayer_dz(ig, 0) * ( znsoil(ig,
                                  1) - P_sat_surf(ig) * rhosurf(ig) )
                        endif

                        ! check if the ground would take up water from the surface but there
                            is none
                        if ((.not.exchange(ig)).and.(pqsurf(ig, igcm_h2o_ice).eq.0.).and.(flux
                            (ig, 0).lt.0.)) then
                              flux(ig, 0) = 0.
```

```
                    endif

                    ! calculate the flux in all other layers
                    do ik = 1, nsoil - 1
                          flux(ig, ik) = D_inter(ig, ik) * ( znsoil(ig, ik + 1) - znsoil(
                              ig, ik) ) / midlayer_dz(ig, ik)
                    enddo
```

Condensation and sublimation are then taken care of by means of the continuity equation (conservation of mass). The formulae used is explained in sec. 5.2.2.1.

A special case appears at the end of the loop and arises where there is not enough ice on the surface to supply the subsurface for the whole timestep. The whole loop is carried out again taking special considerations.

6.2.2.5 Final loops

With the values computed different operations are performed:

- If there is too much ice, diffusion is not possible (see variable `choke_fraction`), blocking the flux. If there is an upwards flux coming from below, `close_top` is set to `.true` (the reciprocal is true for `close_bottom`). Therefore, these variables are set to `.true.` or `.false.` according to the conditions.
- The total mass is computed.
- Using `WRITEDIAGFI()`, the values are stored in a file.

6.2.3 Testing the subsurface model—noGCM simulations

According to Weinmann (n.d.), the `soilwater()` subroutine can be tested using noGCM, which is a version of the Martian model that runs the subsurface decoupled from the atmosphere and without most of the atmospheric calculations. In other wors, noGCM reads the boundary conditions from a file and basically runs the *only* subsurface model for as long as desired, usually until a steady state is achieved. Therefore, noGCM may not only be used to test whether `soilwater()` leads to the expected result, but also to create the initial conditions for the full-blown GCM[20].

Running noGCM requires that the aforementioned reference state be stored in a file. This file shall contain all the variables that are passed to/required by `soilwater()` so as to run. Therefore, a LMDZ simulation needs to be carried out with `callsoilwater=.false.`. To create a *forcing file* that a given time span (i.e. a month, a year…) one may use the tool `concat` on the output files of the reference simulations, and place this new file named **`forcings.nc`** in the noGCM working directory.

6.2.3.1 Compiling noGCM

noGCM may be compiled using `./makelmdz_fcm -arch gfortran -d 64x48x4 -p mars nogcm`.

Atmospheric subroutines such as `vdifc()` still need to be executed, but because `noGCM` is not aimed at simulating the atmosphere, the number of atmospheric layers should be as small as possible (i.e. as low as 2 or 3 for `gfortran`, and 4 for `ifort`) in favour of speed.

6.2.3.2 The starting file

One may use the tool `newstart` to accomodate `start.nc` and `startfi.nc` files to the low z-resolution of the atmosphere. Files could also be created by hand.

[20] Simple hand-made subsurface initial conditions for the full-blown LMDZ are discouraged, as innacurate initialisation of the model has teh potential to disrupt the water cycle. Results obtained by running a subsurface model with a reference atmospheric state as boundary conditions may be more suitable for atmospheric simulations of the atmosphere.

6.2.3.3 Running noGCM The noGCM is run like the normal LMDZ GCM.

7 Bibliography

Anderson, Duwayne M., Maurice J. Schwarz, and Allen R. Tice. 1978. "Water Vapor Adsorption by Sodium Montmorillonite at -5C." *Icarus* 34 (3): 638–44. https://doi.org/10.1016/0019-1035(78)90051-9.

Brunauer, Stephen, Lola S. Deming, W. Edwards Deming, and Edward Teller. 1940. "On a Theory of the van Der Waals Adsorption of Gases." *Journal of the American Chemical Society* 62 (7): 1723–32. https://doi.org/10.1021/ja01864a025.

Brunauer, Stephen, P. H. Emmett, and Edward Teller. 1938. "Adsorption of Gases in Multimolecular Layers." *Journal of the American Chemical Society* 60 (2): 309–19. https://doi.org/10.1021/ja01269a023.

CHEVRIER, V, D OSTROWSKI, and D SEARS. 2008. "Experimental Study of the Sublimation of Ice Through an Unconsolidated Clay Layer: Implications for the Stability of Ice on Mars and the Possible Diurnal Variations in Atmospheric Water." *Icarus* 196 (2): 459–76. https://doi.org/10.1016/j.icarus.2008.03.009.

Clotet, Anna. 2003. "Química Física Avançada, Catàlisi Heterogènia." 2003. http://www.quimica.urv.cat/~w3qf/DOCENCIA/QFA/DPDF/wqfa6c.pdf.

Fairbrother, David HOward. n.d. "Derivation of the Langmuir and Bet Isotherms." https://pages.jh.edu/~chem/fairbr/OLDS/derive.html.

Fanale, F. P., and W. A. Cannon. 1974. "Exchange of Adsorbed H2o and CO2between the Regolith and Atmosphere of Mars Caused by Changes in Surface Insolation." *Journal of Geophysical Research* 79 (24): 3397–3402. https://doi.org/10.1029/jc079i024p03397.

FANALE, F. P., and W. A. CANNON. 1971. "Adsorption on the Martian Regolith." *Nature* 230 (5295): 502–4. https://doi.org/10.1038/230502a0.

Flores, Romeo M. 2014. "Coalification, Gasification, and Gas Storage." In *Coal and Coalbed Gas*, 167–233. Elsevier. https://doi.org/10.1016/b978-0-12-396972-9.00004-5.

Jänchen, Jochen, David L. Bish, Diedrich T. F. Möhlmann, and Helmut Stach. 2006. "Investigation of the Water Sorption Properties of Mars-Relevant Micro- and Mesoporous Minerals." *Icarus* 180 (2): 353–58. https://doi.org/10.1016/j.icarus.2005.10.010.

Jänchen, Jochen, Richard V. Morris, David L. Bish, Mareike Janssen, and Udo Hellwig. 2009. "The H2o and CO2 Adsorption Properties of Phyllosilicate-Poor Palagonitic Dust and Smectites Under Martian Environmental Conditions." *Icarus* 200 (2): 463–67. https://doi.org/10.1016/j.icarus.2008.12.006.

Langmuir, Irving. 1918. "THE ADSORPTION OF GASES ON PLANE SURFACES OF GLASS, MICA AND PLATINUM." *Journal of the American Chemical Society* 40 (9): 1361–1403. https://doi.org/10.1021/ja02242a004.

———. 1932. "VAPOR PRESSURES, EVAPORATION, CONDENSATION AND ADSORPTION." *Journal of the American Chemical Society* 54 (7): 2798–2832. https://doi.org/10.1021/ja01346a022.

Millour, Ehouarn, and François Forget. 2018. *User Manual for the Lmd Martian Atmosphericgeneral Circulation Model*.

Murphy, D. M., and T. Koop. 2005. "Review of the Vapour Pressures of Ice and Supercooled Water for Atmospheric Applications." *Quarterly Journal of the Royal Meteorological Society* 131 (608): 1539–65. https://doi.org/10.1256/qj.04.94.

Pommerol, Antoine, Bernard Schmitt, Pierre Beck, and Olivier Brissaud. 2009. "Water Sorption on Martian Regolith Analogs: Thermodynamics and Near-Infrared Reflectance Spectroscopy." *Icarus* 204 (1): 114–36. https://doi.org/10.1016/j.icarus.2009.06.013.

Schorghofer, Norbert. 2005. "Stability and Exchange of Subsurface Ice on Mars." *Journal of Geophysical Research* 110 (E5). https://doi.org/10.1029/2004je002350.

Skamarock, William C. 2020. "MPAS Atmosphere." 2020. https://mpas-dev.github.io/atmosphere/atmosphere.html.

Titus, T. N., and G. E. Cushing. 2012. "Thermal Diffusivity Experiment at the Grand Falls Dune Field." In *Third International Planetary Dunes Workshop: Remote Sensing and Data Analysis of Planetary Dunes*, 1673:95–96.

Université Marie Curie-Skłodowska. 2013. "Université Marie Curie-Skłodowska." 2013. http://www.zzm.umcs.lublin.pl/Wyklad/FGF-Ang/4A.F.G.F.Ads.S-G.%20Interface.pdf.

Weinmann, Julian. 2019. "Influence of the Martian Regolithon the Atmospheric Methane Andwater Vapour Cycle." Master's thesis, Luleå tekniska universitet, IRAP.

———. n.d. "Handbook for noGCM Simulations."

Weinmann, Julian, and Pierre-Yves Meslin. 2019. "Exploring the Impact of the Regolith on the Martian Water Cycle with a Global Climate Model."

Zent, Aaron P., Fraser P. Fanale, James R. Salvail, and Susan E. Postawko. 1986. "Distribution and State of H2o in the High-Latitude Shallow Subsurface of Mars." *Icarus* 67 (1): 19–36. https://doi.org/10.1016/0019-1035(86)90171-5.

Zent, Aaron P., Robert M. Haberle, Howard C. Houben, and Bruce M. Jakosky. 1993. "A Coupled Subsurface-Boundary Layer Model of Water on Mars." *Journal of Geophysical Research: Planets* 98 (E2): 3319–37. https://doi.org/10.1029/92JE02805.

Zent, A. P., and R. C. Quinn. 1997. "Measurement of H2o Adsorption Under Mars-Like Conditions: Effects of Adsorbent Heterogeneity." *Journal of Geophysical Research: Planets* 102 (E4): 9085–95. https://doi.org/10.1029/96je03420.

8 Log

Work has progressed as shown below:

8.1 Main document

- sec. 1 from 2020-02-26 to 2020-03-03
- sec. 2 from 2020-03-02 to 2020-03-03
  - sec. 2.1 from 2020-03-02 to 2020-03-03
  - sec. 2.2 on 2020-03-16
  - sec. 2.3 on 2020-03-16
  - sec. 2.4 on 2020-03-16
  - sec. 2.5 on 2020-03-16
    * sec. 2.5.1 on 2020-03-16

    - sec. 2.5.2 on 2020-03-16

- sec. 3 from 2020-03-04 to 2020-03-13, and from 2020-04-21 to 2020-04-22
  - sec. 3.1 from 2020-03-04 to 2020-03-11, and on 2020-04-29
  - sec. 3.2 from 2020-03-11 to 2020-03-13, and from 2020-04-21 to 2020-04-22, and from 2020-04-27 to 2020-04-29, and from 2020-05-01 to 2020-05-15
    - sec. **??** on 2020-04-27, and on 2020-04-29
- sec. 4 from 2020-03-16 to 2020-03-20, and 2020-04-08
  - sec. 4.1 on 2020-03-16
  - sec. 4.2 from 2020-03-16 to 2020-03-20 and 2020-04-08
    - sec. 4.2.1 on 2020-03-20
    - sec. 4.2.2 on 2020-03-20 and 2020-04-08
      - sec. 4.2.2.1 on 2020-03-20 and 2020-04-08
      - sec. 4.2.2.2 on 2020-03-20 and 2020-04-08
- sec. 5 from 2020-03-26 to . . .
  - sec. 5.1 from 2020-03-26 to . . .
    - sec. 5.1.1 from 2020-03-26 to 2020-03-30
    - sec. 5.1.2 from 2020-03-26 to 2020-03-30
  - sec. 5.2 from 2020-03-31 to . . .
    - sec. 5.2.1 from 2020-03-31 to 2020-04-02
    - sec. 5.2.2 on 2020-04-02
  - sec. 5.3 from 2020-04-14, and from 2020-04-22 to 2020-04-24
- sec. 6 from 2020-03-20 to 2020-04-16
  - sec. 6.1 from 2020-03-20 to 2020-04-16
    - sec. 6.1.1 on 2020-03-21
    - sec. 6.1.2 on 2020-03-21
    - sec. 6.1.3 on 2020-03-21
      - sec. 6.1.3.1 on 2020-03-21
      - sec. 6.1.3.2 on 2020-03-21
    - sec. 6.1.4 on 2020-03-21
    - sec. 6.1.5 on 2020-04-16
    - sec. 6.1.6 on 2020-04-16
    - sec. 6.1.7 on 2020-04-16
  - sec. 6.2 on 2020-04-17, and on 2020-05-19, and on 2020-05-19
  - sec. 6.2.1 on 2020-05-20, and from 2020-05-25 to 2020-05-27
  - sec. 6.2.2 on 2020-04-17
    - sec. 6.2.3 on 2020-05-19

8.2 Appendices

- sec. 10.1 from 2020-03-04 to 2020-04-06

    - sec. 10.1.1 on 2020-03-04
    - sec. 10.1.2 on 2020-03-05
- sec. 10.2 on 2020-03-05
- sec. 10.3 from 2020-03-18 to 2020-04-07
    - sec. 10.3.1 from 2020-04-06 to 2020-04-07
    - sec. 10.3.2 from 2020-03-24 to 2020-03-25
    - sec. 10.3.3 on 2020-03-19
    - sec. 10.3.4 on 2020-03-19
    - sec. 10.3.5 from 2020-03-19 to 2020-03-20
        - sec. 10.3.5.1 from 2020-13-19 to 2020-03-20
            - sec. 10.3.5.1.1 on 2020-03-20
            - sec. 10.3.5.1.2 on 2020-03-20
    - sec. 10.3.6 on 2020-03-20
        - sec. 10.3.6.1 on 2020-03-20
            - sec. 10.3.6.1.1 on 2020-03-20
    - sec. 10.3.7 on 2020-04-06
        - sec. 10.3.7.1 on 2020-04-06
- sec. 10.4 from 2020-04-09 to 2020-04-12, and from 2020-05-28 to 2020-05-29
    - sec. 10.4.2 from 2020-04-10 to 2020-04-12
- sec. 10.5 on 2020-04-10
    - sec. 10.5.1 on 2020-04-10
- sec. 10.6 from 2020-05-01 to 2020-05-15 (referenced in sec. 3.2)

9 Compilation of this document

Document compiled on 2020/06/04 12:23:36.

To compile the document (remember to update all included files, e.g. update and convert Lyx files to Markdown, as explained in sec. 9.2), use `panrun` and specify the format with the `-t` flag:

```
jupyter-nbconvert \
  --to markdown \
  --output-dir="." \
  ./work/fitting/fitting.ipynb \
jupyter-nbconvert \
  --to markdown \
  --output-dir="." \
  ./work/fitting/additional_data.ipynb \
&& ./panrun notes.md -t latex -o notes.pdf # or -t html -o notes.pdf
```

Alternatively, to compile a simple PDF use the following command[21]:

[21] Maybe set `listings: false` in the YAML metadata.

```
pandoc \
  --defaults='./headers/defaults.yaml' \
  --metadata-file='./headers/filters.yaml' \
  --metadata-file='./headers/latex.yaml' \
  -s notes.md -o notes.pdf
```

Note that the template `eisvogel.tex` has been slightly modified:

- The maximum figure size is now 8 cm-by-15 cm. The change has been done in line 371, where `\setkeys{Gin}{width=\maxwidth,height=\maxheight,keepaspectratio}` was changed to `\setkeys{Gin}{width=10cm,height=15cm,keepaspectratio}`.
- Line 286, `pdfcreator={LaTeX via pandoc with the Eisvogel template}}` was changed to `pdfcreator={Markdown to PDF via pandoc}}`.
- All the occurrences of `\small` were changed to `\footnotesize` to decrease the font size in code/listing environments.

9.1 Pandoc filters

As for the pandoc filters used, a custom version of `pandoc-source-exec` has been used, along with a custom script named `myscript` and others scripts that can be installed from PyPI. The list can be found in the `defaults.yaml` file.

One may have to execute the following so as to have the filters available:

```
sudo pacman -S pandoc
sudo pacman -S ruby
sudo pacman -S pandoc-crossref
sudo pacman -S pandoc-citeproc
pip install pandoc-mustache
pip install pandoc-codeblock-include
pip install pandoc-include
```

For Ubuntu, the available `pandoc` version using `apt` is not the latest:

```
sudo apt install cabal-install
cabal update
cabal install pandoc pandoc-crossref pandoc-citeproc
pip3 install pandoc-mustache
pip3 install pandoc-codeblock-include
pip3 install pandoc-include
```

Also add `PATH=$PATH:$HOME/.cabal/bin` to `~/.bashrc`. If errors appear, ensure that e.g. `pandoc-crossref` was built for the specific `pandoc` version that has been installed.

9.2 From LyX documents to Markdown

Use the following sequence to convert a LyX document to a Markdown file with crossreference support via the Pandoc filter `pandoc-crossref`:

1. Export from LyX to LaTeX using `lyx` and `pdflatex`
2. Convert to Markdown using `pandoc`
3. Remove some escape characters, e.g. `\_` becomes `_`

4. Change `\label{tag}$$` to `$${#tag}`. The command `'s/\\label\{(eq:[^\}]*)\}\$\$/\$\$\{#\1\}/g'` for `sed` is based on the Regular Expression `\\label{eq:[^\}]*\}\$\$`, which selects `\label{eq:my_tag}$$` (wanted) but not `\label{eq:my_tag}\end{gathered}` (not wanted):
   1. `\\label{eq:` selects up to `\label{eq:}`
   2. `[^\}]*` selects until the first `}` is found
   3. `\$\$` selects the following `$$`

If grouped, the command for `file="./work/equations/subsurface_formulae"` (without the extension) is the following:

```
lyx --export pdflatex $file.lyx \
&& pandoc --wrap=none $file.tex -o $file.md \
&& sed -i -e 's/\\_/_/g; s/\\#/#/g; s/\\@/@/g; s/\\>/>/g' $file.md \
&& sed -i -E 's/\\label\{(eq:[^\}]*)\}\$\$/\$\$\{#\1\}/g' $file.md
```

10 Appendices

10.1 Derivation of the Langmuir equation

10.1.1 Thermodynamic derivation

This derivation has been extracted from Clotet (2003)[22].

If the following is considered:

$$\mathrm{A}_g + \star \underset{K_d}{\overset{K_a}{\rightleftharpoons}} \mathrm{A}_{\text{ads}}$$

If N is the number of active sites, $n_\star$ the free sites, and n_{ads} the number of active sites already occupied (number of adsorbed molecules), then it is safe to say that

$$N = n_\star + n_{\text{ads}}$$

Given the coverage $\theta = \frac{n_{\text{ads}}}{N}$, for the desorption and with $[\mathrm{A}_{\text{ads}}] = \frac{n_{\text{ads}}}{V} = \frac{\theta N}{V}$, the velocity of desorption can be found:

$$v_d = K_d\,[\mathrm{A}_{\text{ads}}] = K_d \frac{\theta N}{V}$$

Similarly, for the adsorption phase, with $1-\theta = \frac{n_\star}{N}$ and the ideal gas law $\frac{N_{A_g}}{V} = \frac{p_A}{RT}$ (p_A is the vapor pressure of A_g), the following is found:

$$v_a = K_a\,[\mathrm{A}_g]\,[\star] = K_a\,[\mathrm{A}_g]\,\frac{(1-\theta)\,N}{V} = K_a \frac{p_A}{RT}\frac{(1-\theta)\,N}{V}$$

There is equilibrium, then $v_a = v_d$, so $K_d \frac{\theta N}{V} = \underbrace{\frac{K_a}{RT}}_{K_a'} p_A \frac{(1-\theta)N}{V}$ and therefore:

[22] See Fairbrother (n.d.) for an easier derivation assuming equilibrium from the start.

$$\frac{\theta}{1-\theta} = \underbrace{\frac{K_a'}{K_d}}_{k_{\text{ads}}} p_A$$

NOTE: In a lot of bibliography, K_a includes $\frac{1}{RT}$. So K_a' is usually presented as K_a in some of documents consulted.

Finally, the Langmuir isotherm is found:

$$\theta = \frac{k_{\text{ads}} p_A}{1 + k_{\text{ads}} p_A} \tag{36}$$

It has to be said that in the subsurface model, the adsoprtion kinetics are handled by using a parameter k:

$$\begin{aligned} K_a &= \frac{k\left(s^{-1}\right) k_{\text{ads}}}{1 + \frac{k_{\text{ads}}}{\varepsilon_0}} \\ K_d &= \frac{k\left(s^{-1}\right)}{1 + \frac{k_{\text{ads}}}{\varepsilon_0}} \end{aligned}$$

This appears when introducing the characteristic time τ_{ads} to eq. 14. The following has to be satisfied:

$$\frac{\partial}{\partial t}\left(\varepsilon n_{\text{vap}} + n_{\text{ads}}\right) = 0 \implies \varepsilon n_{\text{vap}} + n_{\text{ads}} = C_\infty$$

In the equation above C_∞ is an integration constant. By reintroducing the above into eq. 14:

$$\begin{aligned} \frac{\partial}{\partial t}\left(n_{\text{ads}}\right) &= K_a n_{\text{vap}} - K_d n_{\text{ads}} \\ &= K_a \left(\frac{C_\infty - n_{\text{ads}}}{\varepsilon}\right) - K_d n_{\text{ads}} \\ &= \frac{K_a}{\varepsilon} C_\infty - \left(\frac{K_a}{\varepsilon} + K_d\right) n_{\text{ads}} \\ n_{\text{ads}} &= \frac{K_a}{\varepsilon} C_\infty \left(1 - e^{-\frac{t}{\frac{K_a}{\varepsilon} + K_d}}\right) \\ &= \frac{K_a}{\varepsilon} C_\infty \left(1 - e^{-\frac{t}{\tau_{\text{ads}}}}\right) \end{aligned}$$

And therefore:

$$\tau_{\text{ads}} = \frac{1}{\frac{K_a}{\varepsilon} + K_d} \tag{37}$$

10.1.2 Kinetic derivation

From Fairbrother (n.d.), the rate of adsorption will be proportional to the pressure of the gas and the number of vacant sites for adsorption. If the total number of sites on the surface is N, then the rate of change of the surface coverage due to adsorption is:

$$\frac{d\theta}{dt} = K_a p_A N (1-\theta)$$

The rate of change of the coverage due to the adsorbate leaving the surface (desorption) is proportional to the number of adsorbed species:

$$\frac{d\theta}{dt} = -K_d N \theta$$

In these equations, K_a and K_d are the rate constants for adsorption and desorption repectively, and p_A is the pressure of the adsorbate gas. At equilibrium, the coverage is independent of time and thus the adorption and desorption rates are equal. The solution to this condition gives us a relation for θ, eq. 36.

10.2 Derivation of the BET equation

From Fairbrother (n.d.), if $\theta_0, \theta_1, \ldots, \theta_n$ is the surface area covered by $0, 1, \ldots, n$ layers of adsorbed molecules, at equilibrium θ_0 must remain constant, i.e. the rate of evaporation from the first layer must be equal to the rate of condensation onto bare surface:

$$K_{-1}\theta_1 = K_1\theta_0 \tag{38}$$

Similarly, at equilibrium θ_1 must remain constant, i.e. the rate of condensation on the bare surface plus the rate of evaporation from the second layer must be equal to the rate of condensation on the 1st layer plus the rate of evaporation from the second layer:

$$K_1 p\theta_0 + K_{-2}\theta_2 = K_2 p\theta_1 + K_{-1}\theta_1 \tag{39}$$

Substituting eq. 39 into eq. 38 gives:

$$K_{-2}\theta_2 = K_2 p\theta_1$$

In general,

$$K_{-i}\theta_i = K_i p\theta_{i-1}$$

Given the total surface area of the catalyst $A = \sum_{i=0}^{\infty}\theta_i$ and the total volume of gas adsorbed on surface $v = v_0 \sum_{i=0}^{\infty} i\theta_i$, with v_0 being the volume of gas adsorbed on one square centimeter of surface when it is covered with a complete layer, the following is true:

$$\frac{v}{\underbrace{v_0 A}_{v_m}} = \frac{\sum_{i=0}^{\infty} i\theta_i}{\sum_{i=0}^{\infty}\theta_i} \tag{40}$$

where v_m is the volume of gas adsorbed when the entire surface is covered with a complete monolayer.

From eq. 38, $\theta_1 = \underbrace{\left(\frac{K_1}{K_{-1}}\right)}_{y} p\,\theta_0$, and also $\theta_2 = \left(\frac{K_2}{K_{-2}}\right) p\theta_1 = \underbrace{\frac{p}{g}}_{x}\,\theta_1$. Note that it is assumed that the properties of the 1st, 2nd,... layers are equivalent $\frac{K_{-2}}{K_2} = \ldots = \frac{K_{-i}}{K_i} = g$.

Therefore, $\theta_3 = x\theta_2 = x^2\theta_1$, and generally, $\theta_i = x\theta_{i-1} = x^{i-1}\theta_1 = x^{i-1}y\theta_0 = cx^i\theta_0$ with $c = \frac{y}{x}$.

Substituting the result into eq. 40:

$$\frac{v}{v_m} = \frac{\sum_{i=0}^{\infty} i\theta_i}{\sum_{i=0}^{\infty}\theta_i} = \frac{c\theta_0\sum_{i=0}^{\infty} ix_i}{(\theta_0+\theta_1+\theta_2+\ldots)} = \frac{c\theta_0\sum_{i=0}^{\infty} ix_i}{\theta_0\left(1+c\sum_{i=0}^{\infty} x_i\right)} = \frac{c\sum_{i=0}^{\infty} ix_i}{1+c\sum_{i=0}^{\infty} x_i}$$

With the sum of an infinite geometric progression $\sum_{i=1}^{\infty} x_i = \frac{x}{1-x}$ and the following development:

$$\begin{aligned}\sum_{i=1}^{\infty} x_i &= x^1 + 2x^2 + 3x^3 + \ldots \\ &= x\left(1 + 2x + 3x^2 + \ldots\right) \\ &= x\frac{d}{dx}\left(\sum_{i=1}^{\infty} x^i\right)\end{aligned}$$

eq. 41 is obtained:

$$\frac{v}{v_m} = \frac{cx}{1-x^2}\cdot\frac{1}{1+\frac{cx}{1-x}} = \frac{cx}{(1-x)^{\cancel{2}}\,\frac{1-x+cx}{\cancel{1-x}}} = \frac{cx}{(1-x)(1-x+cx)} \tag{41}$$

At saturation pressure of gas p_0, an infinite number of adsorbate layers must build up on the surface. From eq. 41, for this to be possible, $\frac{cx}{(1-x)(1-x+cx)}$ must be infinite. This means that at p_0, x must be equal to 1, and therefore $g = p_0$ and $x = \frac{p}{p_0}$, and substituting into eq. 41, the BET isotherm is obtained:

$$v = \frac{v_m cx}{(1-x)\left[1+(c-1)x\right]} \tag{42}$$

eq. 42 can be rearranged to give eq. 43, which is equivalent to eq. 2.

$$\frac{x}{v(1-x)} = \frac{1}{v_m c} + \frac{x(c-1)}{v_m c} \tag{43}$$

Here is another useful resource.

10.3 Installation of LMDZ

The following steps will install a recent LMDZ version. To install the old version which supports the soilwater subroutines (sec. 6.2), check sec. 6.2.1.

The steps required to install LMDZ are listed in Millour and Forget (2018). Herebelow is an adaptation for two usecases:

- the Hyperion2 machine

- the author's machine

An official guide is also found on the LMDZ site, where they also show how to solve some possible errors.

There is also a script to automatically install the model and its dependencies on Linux using Gfortran, as explained in sec. 10.3.1.

10.3.1 Quick install guide for Linux with gfortran

The following steps install LMDZ MARS on a Linux machine:

1. Create `lmd` in `~/Desktop`
2. `cd ~/Desktop/lmd`
3. `wget https://www.lmd.jussieu.fr/~lmdz/planets/install_lmdz_mars.bash`
4. Add `sed -i 's/-fdefault-real-8/-fdefault-real-8 -fdefault-double-8/g'arch-local.fcm` in line 102 of the downloaded file `install_lmdz_mars.bash`, just after `cp arch-gfortran.fcm arch-local.fcm`
5. `chmod +x install_lmdz_mars.bash`
6. `cd ; svn co http://forge.ipsl.jussieu.fr/fcm/svn/PATCHED/FCM_V1.2`
7. `export PATH=$PATH:$HOME/FCM_V1.2/bin`
8. `cd -` to return to `~/Desktop/lmd`
9. Run `./install_lmdz_mars.bash`
   - Ensure that `arch-local.fcm` contains `-fdefault-double-8`, as shown in lst. 13
10. An executable file named `gcm_64x48x29_phymars_seq.e` should have appeared in `~/Desktop/lmd/trunk/LMDZ.COMMON/bin/`
    - Other grids can be configured by compiling with different parameters: `./makelmdz_fcm -arch local -p mars -d 64x48x49 gcm`

Note that these steps will install out-of-date software, but ensures compatibility and a *trouble-free* experience.

10.3.1.1 Alternative The steps below may not work, and have to be revised

The following steps—taken from here—download and install the LMDZ model on a Linux machine:

1. Create `lmd` in `~/Desktop`
2. `cd ~/Desktop/lmd`
3. `wget https://www.lmd.jussieu.fr/~lmdz/pub/install_lmdz.sh`
4. `chmod +x install_lmdz.sh`
5. Run `./install_lmdz.sh`
6. `cd ~/Desktop/lmd/LMDZtrunk/modipsl/modeles/LMDZ`
7. To compile a model, run e.g. `./makelmdz_fcm -arch local -d 64x48x29 -p mars gcm` for a 64-by-48-by-29 grid layout for Mars
   - Architecture must be set to `local`, as explained in the section *Remarques en vrac* of the site.

10.3.2 Quick install guide for Hyperion2 at IRAP

While the list below installs the LMDZ model with the latest dependencies, the list in sec. 10.3.1 ensures a trouble-free installation.

The following procedure has been followed to install LMDZ on the Hyperion2 machine. Based on sections *Getting and Building NetCDF* and *Building the NetCDF-4.2 and later Fortran libraries* of the NetCDF documentation, and Millour and Forget (2018). **Go to sec. 10.3.3 and following sections for detailed installation steps for other machines**.

1. Connect to gateway machine: `ssh -X aprat-gasull@gw.irap.omp.eu`
2. Connect to Hyperion2: `ssh -X aprat-gasull@hyperion2.irap.omp.eu`
3. `cd Models`
4. In `~/Models`, retrieve the LMDZ GCM: `svn checkout http://svn.lmd.jussieu.fr/Planeto/trunk --depth empty`
5. `cd trunk`
6. In `~/Models/trunk`, update the directories: `svn update LMDZ.MARS LMDZ.COMMON UTIL`
7. Download the datadir directory as a compressed file `datadir.tar.gz` in `~/Models`: `wget https://www.lmd.jussieu.fr/~lmdz/planets/mars/datadir.tar.gz`
8. Extract the contents of the file `datadir.tar.gz`: `tar -xvzf datadir.tar.gz`
9. `cd ~`
10. `mkdir zlib`
11. `cd zlib`
12. `export ZDIR=$HOME/zlib`[23]
13. `wget https://www.zlib.net/zlib-1.2.11.tar.gz`
14. `tar -xvzf zlib-1.2.11.tar.gz`
15. `cd zlib-1.2.11`
16. `./configure --prefix=${ZDIR}`
17. `make install`
18. `cd ~`
19. `mkdir hdf5`
20. `cd hdf5`
21. `export H5DIR=$HOME/hdf5`
22. `wget https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.12/hdf5-1.12.0/src/hdf5-1.12.0.tar.gz`
23. `tar -xvzf hdf5-1.12.0.tar.gz`
24. `cd hdf5-1.12.0`
25. `./configure --with-zlib=${ZDIR} --prefix=${H5DIR} --enable-hl`
26. In `~/hdf5-1.12.0` check HDF5: `make check`[24]
27. In `~/hdf5-1.12.0` install HDF5: `make install`
28. `cd ..`
29. `export LD_LIBRARY_PATH=${H5DIR}/lib:${LD_LIBRARY_PATH}`
30. `cd ~`

[23] To make it easier and faster, one can avoid `export`-ing environment variables and therefore skip some of the steps above by copying the contents of the `~/.bash_profile` file presented in lst. 9.

[24] Failing at the time of writing. Stuck at `iopipe`, similar to this.

31. In ~ create the `netcdf` directory: `mkdir netcdf`
32. `cd netcdf`
33. `export NCDIR=$HOME/netcdf` (add to `vim .bash_profile`)
34. In `~/netcdf`, download NetCDF-C: `wget https://www.unidata.ucar.edu/downloads/netcdf/ftp/netcdf-c-4.7.3.tar.gz`
35. Extract the contents of the downloaded file: `tar -xvzf netcdf-c-4.7.3.tar.gz`
36. `cd netcdf-c-4.7.3/`
37. In `~/netcdf/netcdf-c-4.7.3` configure the build: `CPPFLAGS='-I${H5DIR}/include -I${ZDIR}/include'LDFLAGS='-L${H5DIR}/lib -L${ZDIR}/lib'./configure --prefix=${NCDIR} --disable-dap`[25]
38. In `~/netcdf/netcdf-c-4.7.3` check NetCDF: `make check`
39. In `~/netcdf/netcdf-c-4.7.3` install NetCDF: `make install`
40. `cd ..`
41. In `~/netcdf`, download NetCDF-Fortran: `wget get https://www.unidata.ucar.edu/downloads/netcdf/ftp/netcdf-fortran-4.5.2.tar.gz`
42. Extract the contents of the downloaded file: `tar -xvzf netcdf-fortran-4.5.2.tar.gz`
43. `cd netcdf-fortran-4.5.2.tar.gz`
44. `export LD_LIBRARY_PATH=${NCDIR}/lib:${LD_LIBRARY_PATH}`
45. `export NFDIR=$NCDIR`
46. In `~/netcdf/netcdf-fortran-4.5.2` configure the build: `./configure --prefix=${NCDIR}`
47. In `~/netcdf/netcdf-fortran-4.5.2` run `make check`[26]
48. In `~/netcdf/netcdf-fortran-4.5.2` run `make install`
49. Check NetCDF installation parameters to ensure C and Fortran versions are available: `~/netcdf/bin/nc-config --all`
50. `cd ~`
51. `cd Models/trunk/LMDZ.COMMON/`
52. In `~/Models/trunk/LMDZ.COMMON` download the patched[27] `fcm` tool: `svn checkout "http://forge.ipsl.jussieu.fr/fcm/svn/PATCHED/FCM_V1.3/"`[28]
53. In `~/Models/trunk/LMDZ.COMMON` change the name of the directory: `mv FCM_V1.3/ fcm`
54. `export PATH=$PATH:$HOME/Models/trunk/LMDZ.COMMON/fcm/bin`[29]
55. Add `ulimit -s unlimited` to `.bash_profile` using `vim ~/.bash_profile`
56. `cd ioipsl`
57. In `~/Models/trunk/LMDZ.COMMON/ioipsl` copy the IOIPSL install file: `cp install_ioipsl_gfortran-gcc.bash install_ioipsl_gfortran-gcc-mod.bash`[30]

[25] For some reason, the command that works cannot use environment variables `CPPFLAGS='-I/home/STAGIAIRES/aprat-gasull/hdf5/include -I${ZDIR}/include'LDFLAGS='-L/home/STAGIAIRES/aprat-gasull/hdf5/lib -L${ZDIR}/lib'./configure --prefix=${NCDIR} --disable-dap`. It may be due to shared-vs-static libraries.

[26] Failing at the time of writing. Stuck at `tst_f90` and `tst_fill_int64`.

[27] A patched version of FCM, as official releases required `Time/Piece.pm`. If `Time/Piece.pm` is available, the easiest way is to `wget https://github.com/metomi/fcm/archive/2019.09.0.tar.gz`, then `tar -xvzf 2019.09.0.tar.gz`, and finally `mv fcm-2019.09.0/ fcm`.

[28] In the online guide they recommend installing FCM version 1.2.

[29] After this step, executing `fcm` should not output `Can't locate Time/Piece.pm in @INC`. See the accompanying footnote related to the patched FCM version.

[30] An alternative is to copy the IOIPSL install file using `cp install_ioipsl_gnome.bash install_ioipsl_gnome-mod.bash` in `~/Models/trunk/LMDZ.COMMON/ioipsl`.

58. In ~/Models/trunk/LMDZ.COMMON/ioipsl edit the file: `vim install_ioipsl_gfortran-gcc-mod.bash` and change `netcdf_include=${NCDIR}/include` and `netcdf_lib=${NCDIR}/lib`[31]
59. `cd ../arch`
60. In ~/Models/trunk/LMDZ.COMMON/arch copy the FCM file: `cp arch-gfortran.fcm arch-MyArch.fcm`
61. In ~/Models/trunk/LMDZ.COMMON/arch run `vim arch-MyArch.fcm` and add `-fdefault-double-8` in `%BASE_FLAGS` as seen in lst. 12
62. In ~/Models/trunk/LMDZ.COMMON/arch copy the PATH file: `cp arch-gfortran.path arch-MyArch.path`
63. In ~/Models/trunk/LMDZ.COMMON/arch run `vim arch-MyArch.path` and add `-lnetcdff` in `NETCDF_LIB=` as seen in lst. 13
64. `cd ..`
65. `export NETCDF=$NCDIR`
66. In ~/Models/trunk/LMDZ.COMMON/ run `./makelmdz_fcm -arch MyArch -d 64x48x29 -p mars gcm`. If no problem has arisen, **go to the last step**. If there have been some problems, see below:
    1. In ~/Models/trunk/LMDZ.COMMON/ run `vim bld.cfg` and comment out the `muphy` and `bin` tags as shown in lst. 15 if the problem is caused by these `muphy` and `bin` tags
    2. In ~/Models/trunk/LMDZ.COMMON/ run again `./makelmdz_fcm -arch MyArch -d 64x48x29 -p mars gcm`
    3. `cd ~/Models/trunk/LMDZ.COMMON/libo/MyArch_64x48x29_phymars_seq.e/.config/bin/`
    4. In ~/Models/trunk/LMDZ.COMMON/libo/MyArch_64x48x29_phymars_seq.e/.config/bin/ a file named `gcm_64x48x29_phymars_seq.e` has been created
    5. `mkdir ~/Models/trunk/LMDZ.COMMON/bin`
    6. In ~/Models/trunk/LMDZ.COMMON/libo/MyArch_64x48x29_phymars_seq.e/.config/bin/ copy the compiled model to ~/Models/trunk/LMDZ.COMMON/bin: `cp gcm_64x48x29_phymars_seq.e ~/Models/trunk/LMDZ.COMMON/bin`
67. In ~/Models/trunk/LMDZ.COMMON/bin a file named `gcm_64x48x29_phymars_seq.e` should appear

After these steps, the LMDZ model should have been installed.

As a reference, in lst. 9 the contents of the ~/.bash_profile file are shown.

Listing 9: Bash profile file contents

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin:$HOME/Models/trunk/LMDZ.COMMON/fcm/bin

export PATH

export NCDIR=$HOME/netcdf
```

[31] If the alternative file `install_ioipsl_gnome-mod.bash` has been created, then in ~/Models/trunk/LMDZ.COMMON/ioipsl edit the file: `vim install_ioipsl_gnome-mod.bash` and add the contents in lst. 11. Remember to change the following `echo "#-Q- gnome NCDF_INC = ${NCDIR}/include">> AA_make.gdef` and `echo "#-Q- gnome NCDF_LIB = -L${NCDIR}/lib -lnetcdf -lnetcdff">> AA_make.gdef`.

```
export NFDIR=$NCDIR
export H5DIR=$HOME/hdf5
export ZDIR=$HOME/zlib
export LD_LIBRARY_PATH=${NCDIR}/lib:${H5DIR}/lib:${LD_LIBRARY_PATH}
export NETCDF=$NCDIR

ulimit -s unlimited
```

10.3.3 Obtaining LMDZ

In order to get the files that conform the LMDZ GCM the steps below need to be followed:

1. Create `lmd` in `~/Desktop`
2. `cd ~/Desktop/lmd`
3. `svn checkout http://svn.lmd.jussieu.fr/Planeto/trunk --depth empty` in `~/Desktop/lmd`
4. A new `trunk` directory will be created, `cd trunk`
5. `svn update LMDZ.MARS LMDZ.COMMON UTIL` and new files and folders will appear
6. `cd ..` to go to the `lmd` directory
7. Download `datadir.tar.gz` in `lmd`: `wget https://www.lmd.jussieu.fr/~lmdz/planets/mars/datadir.tar.gz`[32]
8. Extract the contents of the file `datadir.tar.gz`: `tar -xvzf datadir.tar.gz`
9. `mkdir starts`
10. `cd starts`
11. Download one of the start files: `wget https://www.lmd.jussieu.fr/~lmdz/planets/mars/starts/start_archive_64x48x49_MY24.nc`

10.3.4 Prerequisites

The following list contains all the prerequisites required to run LMDZ:

- GFortran: `sudo pacman -S gcc-fortran` to install
- NetCDF: `sudo pacman -S netcdf` to install
  - The Open MPI version can be installed using `sudo pacman -S netcdf-openmpi` (either the above or this verion should be installed)
  - NetCDF can be installed on Ubuntu with `sudo apt install netcdf-bin`
- NetCDF Fortran bindings: `sudo pacman -S netcdf-fortran`
  - NetCFD Fortran bindings for the Open MPI version can be installed with `sudo pacman -S netcdf-fortran-openmpi` if `netcdf-openmpi` is installed
- ParaView for visualizing NetCDF files: `sudo pacman -S paraview` to install
  - ParaView can be installed on Ubuntu with `sudo apt install paraview` or through their website
  - Alternative viewers: Ferret, Panoply, GrADS, and ncview
- KornShell: `sudo pacman -S ksh` to install
- The `fcm` utility.

[32]The files that will be downloaded are found here.

  - FCM can be installed on Ubuntu with `sudo apt install fcm`
  - FCM can be compiled on Linux from official FCM releases on GitHub
  - A patched version may be obtained by running `svn checkout "http://forge.ipsl.jussieu.fr/fcm/svn/PATCHED/FCM V1.2"`in `~/Desktop`.
  - Whichever version is chosen, a new `fcm` folder should be created in the system. For the case that this folder is `~/Desktop/lmd/trunk/LMDZ.COMMON/fcm`, the binaries in `~/Desktop/lmd/trunk/LMDZ.COMMON/fcm/bin` need to be made available throughout the system by means of `export PATH=$PATH:$HOME/Desktop/lmd/trunk/LMDZ.COMMON/fcm/bin`. Note that so as to make this change persist, one has to make the change in `~/.bash_profile`, `~/.profile`, or `~/.bashrc`.

- Run `ulimit -s unlimited` before running the GCM or add this command to `~/.bash_profile`, `~/.profile`, or `~/.bashrc`.

10.3.5 Installing the model

In Millour and Forget (2018), two installation methods are presented. In this guide the advised method (i.e. section *Settings in LMDZ.COMMON (advised)* in Millour and Forget (2018)) will be followed.

10.3.5.1 Settings in LMDZ.COMMON—IOIPSL installation The following differs a little bit from Millour and Forget (2018), as the steps listed there may not work for those who do not use `ifort` or prefer using `gfortran`. A new file needs to be created as explained in sec. 10.3.5.1.1, as the *easy* method shown in Millour and Forget (2018) does not work.

1. `cd ~/Desktop/lmd/trunk/LMDZ.COMMON/ioipsl`
2. Create and run `./install_ioipsl_gnome_gfortran.bash`, which is a modified version of `./install_ioipsl_gnome.bash` using `gfortran` instead of `ifort` available within the Intel Parallel Studio XE (see sec. 10.3.5.1.1).
3. After executing `./install_ioipsl_gnome_gfortran.bash` and some possible `Warnings`, `ioipsl` is successfully installed if the run ended with `OK: ioipsl library is in ...`, where `libioipsl.a` and companion files are located (in this case, `~/Desktop/lmd/trunk/LMDZ.COMMON/ioipsl/modipsl/lib/libioipsl.a`).

An *arguably esasier* alternative is to modify `~/lmd/trunk/LMDZ.COMMON/ioipsl/modipsl/util/AA_make.gdef` and run `./install_ioipsl_gfortran-gcc.bash`, as explained in sec. 10.3.5.1.2.

10.3.5.1.1 Modified version of `install_ioipsl_gnome.bash` for Manjaro Linux GNOME By using the parameters already presented in `~/Desktop/lmd/trunk/LMDZ.COMMON/ioipsl/modipsl/util/AA_make.gdef` (see lst. 10), the file `install_ioipsl_gnome.bash` in `~/lmd/trunk/LMDZ.COMMON/ioipsl` has to be changed to `install_ioipsl_gnome_gfortran.bash` with the contents shown in lst. 11.

Note that **it is CRITICAL** that in the line `echo "#-Q- gnome NCDF_INC = /usr/include">> AA_make.gdef` of `install_ioipsl_gnome.bash` (see lst. 11), the directory `/usr/include` contains `netcdf.mod`. Make sure that this file is in the path specified. You can see the files installed by any package using `pacman -Ql` **`package`**[33].

[33]Running `pacman -Ql netcdf-fortran-openmpi | grep netcdf.mod` returns `netcdf-fortran-openmpi /usr/include/netcdf.mod`, which means that `netdf.mod` is in `/usr/include`.

The reader may also change the line `Global definitions for Manjaro Linux GNOME (sapastre)for Arnau Prat Gasull (aeronau)` in lst. 11.

Each time `install_ioipsl_gnome_gfortran.bashr` is run, the parameters shown in lst. 11 are appended in

Listing 10: File taken as reference (AA-make.gdef)

```
#-Q- gfortran  #- Global definitions for gfortran, generic system
#-Q- gfortran  M_K = make
#-Q- gfortran  P_C = cpp
#-Q- gfortran  FCM_ARCH = gfortran
#-Q- gfortran  P_O = -fpreprocessed -P -C -traditional $(P_P)
#-Q- gfortran  F_C = gfortran -c -cpp
#-Q- gfortran  #-D- MD    F_D = -g -Wall -fbounds-check -pedantic -finit-real=nan
#-Q- gfortran  #-D- MN    F_D =
#-Q- gfortran  #-P- I4R4  F_P =
#-Q- gfortran  #-P- I4R8  F_P = -fdefault-real-8
#-Q- gfortran  #-P- I8R8  F_P = -fdefault-integer-8 -fdefault-real-8
#-Q- gfortran  w_w = -O3 -funroll-all-loops $(F_D) $(F_P) -I$(MODDIR)
#-Q- gfortran  F_O = $(w_w) -J$(MODDIR)
#-Q- gfortran  F_L = gfortran
#-Q- gfortran  M_M = 0
#-Q- gfortran  L_X = 0
#-Q- gfortran  L_O =
#-Q- gfortran  A_C = ar -rs
#-Q- gfortran  A_G = ar -x
#-Q- gfortran  C_C = cc -c
#-Q- gfortran  C_O =
#-Q- gfortran  C_L = cc
#-Q- gfortran  #-
#-Q- gfortran  NCDF_INC =
#-Q- gfortran  NCDF_LIB = -L -lnetcdf -lnetcdff
#-Q- gfortran  #-
```

Listing 11: New file file install-ioipsl-gnome-gfortran.bash derived from install-ioipsl-gnome.bash

```
# 2. Set correct settings:
# add a "gnome" configuration to AA_make.gdef
echo "#-Q- gnome  #- Global definitions for Manjaro Linux GNOME (sapastre) for Arnau
    Prat Gasull (aeronau)" >> AA_make.gdef
echo "#-Q- gnome  M_K = make" >> AA_make.gdef
echo "#-Q- gnome  P_C = cpp" >> AA_make.gdef
#-Q- gfortran  FCM_ARCH = gfortran
echo '#-Q- gnome  P_O = -fpreprocessed -P -C -traditional $(P_P)' >> AA_make.gdef
echo "#-Q- gnome  F_C = gfortran -c -cpp" >> AA_make.gdef
echo "#-Q- gnome  #-D- MD    F_D = -g -Wall -fbounds-check -pedantic -finit-real=nan" >>
     AA_make.gdef
echo "#-Q- gnome  #-D- MN    F_D =" >> AA_make.gdef
echo "#-Q- gnome  #-P- I4R4  F_P =" >> AA_make.gdef
echo "#-Q- gnome  #-P- I4R8  F_P = -fdefault-real-8" >> AA_make.gdef
echo "#-Q- gnome  #-P- I8R8  F_P = -fdefault-integer-8 -fdefault-real-8" >> AA_make.gdef
echo '#-Q- gnome  F_O = -O3 -funroll-all-loops $(F_D) $(F_P) -I$(MODDIR) -J$(MODDIR)' >>
     AA_make.gdef
echo "#-Q- gnome  F_L = gfortran" >> AA_make.gdef
echo "#-Q- gnome  M_M = 0" >> AA_make.gdef
echo "#-Q- gnome  L_X = 0" >> AA_make.gdef
echo "#-Q- gnome  L_O =" >> AA_make.gdef
echo "#-Q- gnome  A_C = ar -rs" >> AA_make.gdef
echo "#-Q- gnome  A_G = ar -x" >> AA_make.gdef
echo "#-Q- gnome  C_C = cc -c" >> AA_make.gdef
echo "#-Q- gnome  C_O =" >> AA_make.gdef
echo "#-Q- gnome  C_L = cc" >> AA_make.gdef
echo "#-Q- gnome  #-" >> AA_make.gdef
echo "#-Q- gnome  NCDF_INC = /usr/include" >> AA_make.gdef
echo "#-Q- gnome  NCDF_LIB = -L -lnetcdf -lnetcdff" >> AA_make.gdef
echo "#-Q- gnome  #-" >> AA_make.gdef
```

10.3.5.1.2 Modified version of AA_make.gdef for Manjaro Linux GNOME

In order to compile IPIPSL, one can also modify the line `#-Q- gfortran NCDF_INC =` (line 385[34]) of the `gfortran` section in `AA_make.gdef` (located in `~/Desktop/lmd/trunk/LMDZ.COMMON/ioipsl/modipsl/util`) to `#-Q- gfortran NCDF_INC = /usr/include`, where `/usr/include` contains `netdf.mod`. The line is also shown in lst. 10.

This makes running either `install_ioipsl_gfortran.bash` (if `export NETCDF=/usr` was previously run) or `install_ioipsl_gfortran-gcc.bash`—both located in `~/Desktop/lmd/trunk/LMDZ.COMMON/ioipsl`—work. This makes the compilation of IOIPSL arguably easier than creating the new file `install_ioipsl_gnome_gfortran.bash` (as explained in sec. 10.3.5.1.1), but requires modifying default files.

10.3.6 Compiling the model

The following steps are required to compile the software:

1. `cd ~/Desktop/lmd/trunk/LMDZ.COMMON`
2. Ensure that `fcm` is in `PATH` (see sec. 10.3.4)
3. In `~/Desktop/lmd/trunk/LMDZ.COMMON/arch` copy `arch-gfortran.path` and `arch-gfortran.fcm` to `arch-MyArch.path` and `arch-MyArch.fcm`, respectively
4. In `arch-MyArch.fcm`, add `-fdefault-double-8` in `%BASE_FLAGS`, as seen in lst. 12
    - If not added, errors such as `Error: Type mismatch in argument 'mem_mccn_co2'at (1); passed REAL(16)to REAL(8)` will arise
5. In `arch-MyArch.path`, add `-lnetcdff` in `NETCDF_LIB=`, as seen in lst. 13
    - If not added, errors such as `/usr/bin/ld: dynetat0.f90:(.text+0x120b): undefined reference to '__netcdf_MOD_nf90_inq_varid'` will appear
    - In `arch-MyArch.path`, a reference to the XIOS library appears. It will only be used if IOIPSL is not installed, so there is no need to compile it if the installation steps presented in sec. 10.3.5.1 have been followed
6. Ensure `NETCDF` is the desired system/environment variable that points to the directory that includes the folders `/include` and `/lib` where NetCDF files are stored (i.e. run `export NETCDF=/usr` if not done as part of sec. 10.3.5.1.2)
7. Run `./makelmdz_fcm -arch MyArch -d 64x48x29 -p mars gcm` for a 64-by-48-by-29 grid layout for Mars.
    - If some errors such as the ones shown in lst. 14 arise from file `~/Desktop/lmd/trunk/LMDZ.COMMON/bld.cfg` proceed to sec. 10.3.6.1.1.
8. In `~/Desktop/lmd/trunk/LMDZ.COMMON/bin` an executable called `gcm_64x48x29_phymars_seq.e` has been created.

[34] At the time of writing, line number 385 `#-Q- gfortran NCDF_INC =` in `AA_make.gdef` is to be modified.

Listing 12: Contents of arch-MyArch.fcm

```
%COMPILER            gfortran
%LINK                gfortran
%AR                  ar
%MAKE                make
%FPP_FLAGS           -P -traditional
%CPP_FLAGS           -P
%CPP_DEF             LAPACK
%FPP_DEF             NC_DOUBLE
%BASE_FFLAGS         -c -fdefault-real-8 -fdefault-double-8 -ffree-line-length-none -fno
    -align-commons
%PROD_FFLAGS         -O3
%DEV_FFLAGS          -O
%DEBUG_FFLAGS        -ffpe-trap=invalid,zero,overflow -fbounds-check -g3 -O0 -fstack-
    protector-all
%C_COMPILER          gcc
%C_OPTIM             -O3
%MPI_FFLAGS
%OMP_FFLAGS
%BASE_LD
%MPI_LD
%OMP_LD
```

Listing 13: Contents of arch-MyArch.path

```
ROOT=$PWD

NETCDF_LIBDIR="-L$NETCDF/lib"
NETCDF_LIB="-lnetcdff -lnetcdf"
NETCDF_INCDIR="-I$NETCDF/include"

IOIPSL_INCDIR="-I$ROOT/ioipsl/modipsl/lib"
IOIPSL_LIBDIR="-L$ROOT/ioipsl/modipsl/lib"
IOIPSL_LIB="-lioipsl"

XIOS_INCDIR="-I$ROOT/../XIOS/inc"
XIOS_LIBDIR="-L$ROOT/../XIOS/lib"
XIOS_LIB="-lxios -lstdc++"
```

10.3.6.1 Compilation problems

10.3.6.1.1 Errors in the `bld.cfg` FCM configuration file—file not in the `bin` subdirectory

This is because FCM 1.3 was installed, while FCM 1.2 should be installed, as explained in sec. 10.3.2.

In `bld.cfg` one can choose which modules to install. In lst. 14, one can see that an error is due to `muphy`, the Titan microphysic package. In Martian simulations, files related to Titan are not used, so line 47 in `bld.cfg` can be commented out, as shown in lst. 15. The `bin` tag can also be commented out. Once this is done, try running again the command (step 7 in sec. 10.3.6): `./makelmdz_fcm -arch MyArch -d 64x48x29 -p mars gcm`. An executable file called `gcm_64x48x29_phymars_seq.e` is found in `~/Desktop/lmd/trunk/LMDZ.COMMON/libo/MyArch_64x48x29_phymars_seq.e/.config/bin`. One can artificially create the `bin` directory `mkdir ~/Desktop/lmd/trunk/LMDZ.COMMON/bin` and from the directory`~/Models/trunk/LMDZ.COMMON/libo/MyArch_64x48x29_phymars_seq.e/.config/bin/`, it is easy to copy the compiled model to `~/Models/trunk/LMDZ.COMMON/bin` with `cp gcm_64x48x29_phymars_seq.e ~/Models/trunk/LMDZ.COMMON/bin`.

Listing 14: Possible compilation error due to bld.cfg

```
ERROR: /home/aeronau/Desktop/lmd/trunk/LMDZ.COMMON/bld.cfg: LINE 47:
       bld::tool::fflags::muphy: invalid sub-package in declaration.
ERROR: /home/aeronau/Desktop/lmd/trunk/LMDZ.COMMON/bld.cfg: LINE 59:
       dir::bin: label not recognised.
```

Listing 15: Commented out lines in bld.cfg

```
# IMPORTANT: muphytitan package hates floating point promotion !
# This is unfortunately specific to Titan microphysic package !!
# bld::tool::fflags::muphy  -c %INCDIR %COMPIL_FFLAGS %PARA_FFLAGS

...

# dir::bin             %ROOT_PATH/bin
```

10.3.7 Compiling other companion tools

10.3.7.1 Compiling the `newstart` tool

To compile the `newstart` tool follow the steps below:

1. `cd ~/Desktop/lmd/trunk/LMDZ.COMMON`
2. `./makelmdz_fcm -arch MyArch -d 64x48x29 -p mars newstart`
3. The file `newstart_64x48x29_phymars_seq.e` may have been created in `~/Desktop/lmd/trunk/LMDZ.COMMON/bin`
   - If there is no new file in `~/Desktop/lmd/trunk/LMDZ.COMMON/bin`, go to the `LMDZ.COMMON` directory via `cd ~/Desktop/lmd/trunk/LMDZ.COMMON` and copy the file directly `cp libo/MyArch_64x48x29_phymars_seq.e/.config/bin/newstart_64x48x29_phymars_seq.e bin`, just like it had been performed in sec. 10.3.6 for the GCM `gcm_64x48x29_phymars_seq.e`

10.4 Programming in Fortran

The following may be used:

- To run Fortran in Jupyter, follow this example.
- To execute a Markdown code block written in Fortran (e.g. lst. 16), use `pandoc-source-exec`.

10.4.1 Simple examples

Mandatory *hello world* example in Fortran shown in lst. 16:

Listing 16: Executing Fortran code in a Markdown code block

```fortran
program hello
  print *, "Hello World!"
end program hello
```

Output:

```
INFO: Contents not changed since 2020-04-11 10:36:05.000000000 +0200
 Hello World!
```

The code in lst. 17 sums $a + a + a + \ldots$ without surpassing 100. The original file required the input of the user.

Listing 17: A summation in Fortran

```
! sum.f90
! Performs summations using in a loop using EXIT statement
! Saves input information and the summation in a data file

program summation
implicit none
integer :: sum, a

! print*, "This program performs summations"
print*, "This program performs a summation a + a + ... <= 100"
open(unit=10, file="./work/fortran/tutorial/SumData.DAT")

sum = 0
a = 5

do
 ! print*, "Add:"
 ! read*, a
 ! if (a == 0) then
 if (sum >= 100) then
  exit
 else
  sum = sum + a
 end if
 print*, "Curr_sum  =", sum
 write(10,*) sum
end do

print*, "Summation =", sum
write(10,*) "Summation =", sum
close(10)

end
```

Output:

```
INFO: Contents not changed since 2020-04-11 10:36:06.000000000 +0200
 This program performs a summation a + a + ... <= 100
 Curr_sum  =           5
 Curr_sum  =          10
 Curr_sum  =          15
 Curr_sum  =          20
 Curr_sum  =          25
 Curr_sum  =          30
 Curr_sum  =          35
 Curr_sum  =          40
 Curr_sum  =          45
 Curr_sum  =          50
 Curr_sum  =          55
 Curr_sum  =          60
 Curr_sum  =          65
 Curr_sum  =          70
 Curr_sum   =          75
 Curr_sum   =          80
 Curr_sum   =          85
 Curr_sum   =          90
 Curr_sum   =          95
 Curr_sum   =         100
 Summation  =         100
```

10.4.2 Fortran quick reference

Extracted/copied from the following resources:

- **Fortran 90/95 reference**
- fortran_quick_reference_cheat_crib_sheet
- Fortran-Cheat-Sheet
- The GNU Fortran Compiler
- The Intel Fortran Compiler

FORTRAN 77 and below are case sensitive, but Fortran 90 and above are NOT case sensitive. The following sectoins may mix upper and lower case letters.

10.4.2.1 Terminology

Statement An instruction which is either executabe or nonexecutable.
Construct A sequence of statements ending with a construct terminal statement.
Function A procedure that returns the value of a single variable.
Procedure Either a function or subroutine. Intrinsic procedure, external procedure, module procedure, internal procedure, dummy procedure or statement function.
Subroutine A procedure that is invoked by a CALL statement or defined assignment statement. It can return more than one argument.

10.4.2.2 Special characters

' (Apostrophe) Editing, declaring a string
" (Quotation Marks) Declaring a string
*** (Backslash and asterisk)** Comment lines.
: (Colon) Editing.
:: (Double Colon) Separator.
! (Exclamation) inline comment.
; (Semicolon) Separates Statement on single source line. Except when it is in a character context.
& (Ampersand) Line continuation character.

10.4.2.3 Data types and conversions Variables are declared with their type and a comma separated list of variable names. Options may also be included.

```
type:: names
type, options:: names
```

`type` is one of the following:

Logical Boolean taking values `.true.` or `.false.`, like C's `bool`.

`logical`

Conversion with `logical(l [, kind])`.

Integer Usually 32 or 64 bit, like C's `int`.

`integer`

Conversion with `int(a [, kind])`. Some use `integer*n` for a n-byte integer, but it is discouraged. Fortran 2008 adds `int8`, `int16`...

Real Usually 32 bit, like C's `float`.

`real`

Conversion with `real(a [, kind])`.

Float Conversion with `float(arg)`, a specific type of `real()` of kind 4.

Double Usually 64 bit, like C's `double`.

`double precision`

Conversion with `dble(a)`.

Character Strings.

`character(len=10):: ch` (if `len` omitted, it equals to 1).

Conversion with `char(i [, kind])`.

Complex For complex numbers.

`complex`

Conversion with `cmplx(x [, y [, kind]])`.

User defined types Like C's `structs`.

`type(name)`

The `kind` type parameters are defined here. Note that it may be important to pass `-fdefault-integer-8` and `-fdefault-real-8` to the compiler, as seen in sec. 10.3.1, sec. 10.3.2, sec. 10.3.5.1.1, and sec. 10.3.6.

Variables can be initialized when declared, e.g `integer:: a=0, b=5`, *but this implicitly adds save, making them static*[35]*!*.

For local variables, `options` is a comma separated list of:

`parameter` Constants, like C's `const`.

`allocatable` An array that will be dynamically allocated.

`dimension(...)` Declare shape of array.

`save` Retain value between calls.

For subroutine arguments, `options` may also include `intent` and `optional`.

Also see sec. 10.5.1.

10.4.2.4 Arrays

The following declares a vector `v` of length 4, a matrix `A` with 4 rows and 6 columns (arrays can have up to 7 dimensions), and vectors x and y of length 50.

[35] That is, they are initialized only the first time a function is called, while subsequent calls retain the value from the previous call..

```
real:: v(4), A(4,6)
real, dimension(50):: x, y
```

One can also declare the lower and upper bound (impossible in C) using `real:: x(-50:-1), y(-25,24)`. For unknown lengths use `:` as shown herebelow. It is used for allocatable arrays, and for *assumed-shape array* arguments in procedures:

```
real:: arg(:)
real, allocatable:: x(:), B(:,:)
```

10.4.2.4.1 Array initialization Arrays are initialized with the pair `(/ ... /)` as in `x = (/ 1, 2, 3.5, 4.2 /)`, but Fortran 2003 also accepts brackets `[ ... ]`. For multi-dimensional matrices, `reshape(...)` is used (values are taken *column-wise*, so each row below becomes a column):

```
A = reshape( (/ &
  1, 2, 3,      &
  4, 5, 6       &
/), (/ 3, 2 /)
```

The implied do syntax creates a list of values. Implied do loops can be nested. (Note the i and j are just dummy loop counters, and not the array indices.) An implied do loop can be one element in a list.

	Implied do notation	Equivalent to

Simple	`x=(/ (2*i, i=1,4)/)`	`do i = 1,4`
		`  x(i) = 2*i`
		`end do`

Nested	`x=(/ ((i*j, i=1,4), j=1,6)/)`	`k = 1`
		`do j = 1,6`
		`  do i = 1,4`
		`    x(k) = i*j`
		`    k = k+1`
		`  end do`
		`end do`

Item in list	`x=(/ 0, 0, (i, i=3,5), 0, 0 /)`	`x = (/ 0, 0, 3, 4, 5, 0, 0 /)`

10.4.2.4.2 Array operations Arrays with the same shape (i.e. same number ofe dimensins, and same length in each dimension) are called *conformable*. A scalar is conformable with any array. Many operators and intrinsic functions may be applied element-wise to conformable arrays. Some examples:

	Matrix notation	Equivalent to
Element-wise multiplication	`z = x*y`	`do i = 1,4` `z(i) = x(i)*y(i)` `end do`
Initialization	`A = 0.`	`do i = 1,4` `do j = 1,6` `A(i,j) = 0.` `end do` `end do`
Element-wise function	`z = sin(x)`	`do i = 1,4` `z(i) = sin( x(i))` `end do`
Element-wise function	`z = max( 100., x, y )`	`do i = 1,4` `z(i) = max( 100., x(i), y(i))` `end do`

10.4.2.5 Strings Strings are fixed length (text that is shorter is padded on right with spaces) and are enclosed in `' .... '` or `" ... "`. To include an apostrophe in the middle of a word, the string has to be wrapped in `"` (e.g. `"foo'bar"`) or add two apostrophes (e.g. `'foo''bar'`). For `"` the case is reciprocal.

Example of passing array of strings:

```
character(len=*), dimension(:):: strings
character(len=len(strings)):: str
```

Frequently, strings have to be trimmed to remove padding when printing them (e.g. `print '(a,a,a)', " Error opening file '", trim(filename), "'"`).

10.4.2.6 Derived types `type` is used to create data structures (like C `structs`) and must be declared in modules to be used by multiple subroutines. They are initialized much like C, and structures can be nested. Individual members are accessed with a `%` percent, as they would be with a `.` period in C. Example:

```
type person
    character(len=20):: first, last
    integer::           birthyear
    character(len=1)::  gender
end type

type employee
    type(person)::      person
    integer::           hire_date
    character(len=20):: department
end type
```

```
type(person):: jack
type(employee):: jill
jack = person( "Jack", "Smith", 1984, "M" )
jill = employee( person( "Jill", "Smith", 1984, "F" ), 2003, "sales" )

print *, jack%first, jack%last
print *, jill%person%first, jill%person%last, jill%department
```

10.4.2.7 Operators From highest to lowest precedence. Groups of equal precedence are shown by color and dividing lines; within a group, precedence is left-to-right. Parenthesis obviously supersede order of operations.

Operator	Description	Example
`**`	exponent	`a**b`
`*`	multiply	`a*b`
`/`	divide	`a/b`
`+`	add	`a + b`
`-`	subtract	`a - b`
`//`	string concatenation	`"foo"//"bar"`
< or `.lt.`	less than	`a < b`
<= or `.le.`	less than or equals	`a <= b`
> or `.gt.`	greater than	`a > b`
>= or `.ge.`	greater than or equals	`a >= b`
== or `.eq.`	equals	`a == b`
/= or `.ne.`	not equals †	`a /= b`
`.not.`	logical not (unary)	`.not. a`
`.and.`	logical and	`a .and. b`
`.or.`	logical or	`a .or. b`
`.eqv.`	logical = equals	`a .eqv. b`
`.neqv.`	xor, logical not equals	`a .neqv. b`

In C `true == true`, but in Fortran, `.true. .eqv. .true.`, i.e. arithmetic operators do not operate on logical expressions (though == could be overloaded for logicals). The reciprocal—logical operators operate only on logical expressions—holds also true.

10.4.2.8 Conditionals Fortran has single line if statements:

```
if ( logical_expr ) statement
```

and block if-else statements:

```
if ( logical_expr ) then
  ...
else if ( logical_expr ) then
  ...
else
  ...
end if
```

Here, `logical_expr` must have a logical value. Numeric and pointer expressions are not allowed.

There is a C's **switch**-like statement in Fortran, named `select`:

```
select case( hour )
  case( 1:8 )
    activity = 'sleep'
  case( 9:11, 13, 14 )
    activity = 'class'
  case( 12, 17 )
    activity = 'meal'
  case default
    activity = 'copious free time'
end select
```

10.4.2.9 Loops

Consider the following:

```
do variable = first, last[, increment]
  ...
end do
```

This is equivalent to `for( variable = first; variable <= last; variable += increment ){ ... }` for an increment > 0. The counter (i.e. `increment`) is not needed, but `exit` must be specified.

To immediately leave a loop `exit` must be used.`cycle` skips the rest of the current iteration.

There is a do-while construct, like C's while loop:

```
do while( logical_expr )
  ...
end do
```

which is equivalent to:

```
do
  if ( .not. logical_expr ) break
    ...
end do
```

10.4.2.10 Procedures

Fortran has 2 types of procedures: functions, which return a value (and usually should not modify their arguments), and subroutines, which presumably modify their arguments or have other side effects. A subroutine is like a void function in C.

10.4.2.10.1 Subroutine syntax

In:

```
subroutine name( args )
  ...argument declarations...
  ...local variable declarations...
  ...statements...
end [subroutine [name]]
```

`return` exits the function or subroutine immediately, like C's `return`. However, **it does not take the return value**, unlike C's `return`.

10.4.2.10.2 Function syntax

Consider:

```
type function name( args )
    ...argument declarations...
    ...local variable declarations...
    ...statements...
    name = ...
end [function [name]]
```

To specify the return value assign to the function's name, as if it were a variable (i.e. add `name = ...` in the body). Fortran can handle returning arrays and composite types with no difficulty.

10.4.2.10.3 Declaring arguments

Variables may declare the intent [in their name], i.e. whether it is input, output, or both input & output.

Declaration	Description
`intent(in)`	variable may not be changed, like C's const
`intent(out)`	variable may not be referenced before being set, no equivalent in C
`intent(inout)`	no constraints

The reason for all the intents is that **Fortran passes everything by reference**, so it would be very easy to accidentally modify some scalar variable.

Arguments may also be optional, using the `optional` keyword. To test whether an optional argument was passed in, use the `present(argument)` function. No default value is assigned.

10.4.2.10.4 Passing array arguments

To pass arrays into a procedure a colon `:` is used in place of the unknown lengths of each dimension (these become assumed-shape arrays). Assumed-shape arrays **absolutely require** the procedure to be in a module, which generates an implicit interface, or to have an explicit interface. Without an interface, it will compile but fail at runtime, since the caller doesn't know to use assumed-shape semantics.

The size function queries for the total number of elements in an array. The `lbound` and `ubound` functions query for the lower and upper bounds for a particular dimension.

```
real:: x(:)
real:: A(:,:)

m = ubound(A,1)
n = ubound(A,2)

n = size(array)
n = lbound(array, dimension)
n = ubound(array, dimension)
```

10.4.2.10.5 Returning array arguments To return an array, the return type, including its size, must be declared along with the other arguments. It cannot be put before the function keyword. For instance:

```
function matrix_multiply( A, B )
  real:: A(:,:), B(:,:)
  real:: matrix_multiply( ubound(A,1), ubound(B,2) )

  matrix_multiply = matmul( A, B )
end function matrix_multiply
```

10.4.2.11 Modules Variables declared in the module are global to the entire module. Procedures and variables can be declared `public` or `private`. The `implicit none` statement is used to inhibit a very old feature of Fortran that by default treats all variables that start with the letters `i`, `j`, `k`, `l`, `m`, and `n` as integers and all other variables as real arguments. **`implicit none` should always be used**. It prevents potential confusion in variable types, and makes detection of typographic errors easier.

```
module name
  implicit none
  save
  [public | private]
  ...module variable declarations...
  ...interfaces declarations...
contains
  ...procedure definitions...
end [module [name]]
```

If there are no procedures, then `contains` is not required.

Another program or procedure uses the variables and procedures in a module with the `use` module command. By default, all of the module's public variables and procedures are imported into the current namespace. Variables and procedures to be imported can be selected using the `only` syntax, and locally renamed using => syntax, similar to Python's `import bar.b as c` (e.g. `use bar, only: a, b => c`).

```
module bar
  implicit none
  integer:: a, b

  procedure baz( b )
    integer:: b
    print *, b
  end procedure baz
end module bar

procedure foo()
  use bar
  a = 1
  b = 5
  baz( 5 )
end procedure foo

procedure foo2()
  use bar, only: a, b => c
  a = 2
  c = 3
end procedure foo2
```

10.4.2.12 Interfaces (prototypes) Without a name, the `interface ... end interface` lists procedures, in much the way a C header file lists function prototypes. Interfaces should be written for all procedures defined in the same file as your main program. But because modules automatically create interfaces, it may be better to write everything inside modules.

```
interface

  real function foo( arg1, arg2 )
    implicit none
    integer:: arg1
    real:: arg2
  end function

  subroutine bar( arg1 )
    implicit none
    integer:: arg1
  end subroutine

end interface
```

10.4.2.13 Interfaces (overloading generic procedures) Generic (or overloaded) procedures are also defined using interface. The generic name for the procedure is to be defined before the interfaces for specific procedures. To use a specitic procedure defined in a module, use `module procedure name`.

```
module norm_m
  use complex_m, only: norm_complex

  interface norm
    interface
      real function matrix_norm( A )
        real:: A(:,:)
      end subroutine
    end interface

    module procedure norm_complex, norm_real
  end interface norm

contains

  real function norm_real( x )
    real:: x(:)
    ...code...
  end subroutine
end module norm_m


module complex_m
contains
  real function norm_complex( x )
    complex:: x(:)
    ...code...
  end subroutine
end module norm_complex_m
```

The procedures must be differentiated by their argument types, both by position and by name. So the following won't work, because `norm(A=X, kind='2')` matches both `foo1` and `foo2`.

```
interface norm
  interface
    subroutine foo1( A, kind )
      real:: A(:,:)
      character(len=1):: kind
    end subroutine
  end interface
  interface
    subroutine foo2( kind, A )
      real:: A(:,:)
      character(len=1):: kind
    end subroutine
  end interface
end interface
```

Note that the different procedures (`foo1` and `foo2`) are mapped to a common name `norm`, but in C++ the procedures would have the same name with different arguments.

10.4.2.14 Input and Ouput

10.4.2.14.1 List directed input With `read *, variables`, the data is delimited into fields by comma, whitespace, / slash, or newline. Strings are delimited by either '…' single or "…" double quotes, or are a single "word". Two consecutive commas are a null value—the variable remains unchanged. Slash terminates input—all remaining variables are unchanged.

10.4.2.14.2 User-formatted input Input is achieved with:

```
read '(formats)', variables
```

or

```
read label, variables
label  format (formats)
```

User-formatted input uses fixed width fields, specified in formats, which is a comma separated list of the format codes below.

Format	Description
`Iw`	Read `w` characters as an integer
`Fw.d`	Read `w` characters as a real. If it doesn't contain a decimal point, assume `d` decimal places.
`Ew.d`	For input, same as `Fw.d`
`Aw`	Read `w` characters as string. If `w > len(variable)` then it truncates the left side of text. (This is because formatted output is right aligned, so the leading spaces should be lopped off.)
`A`	Read `w=len(variable)` characters as a string.
`Lw`	Read `w` characters as logical. Text starting with `T` or `.T` is true, text starting with `F` or `.F` is false.
`wX`	Ignore `w` characters.
`Tc`	Move ("tab") to absolute position `c`
`TLw`	Move left `w` characters

Format	Description
`TRw`	Move right `w` characters
`/`	Ignore rest of line (record) and read next line (record)

10.4.2.14.3 List directed output List directed output uses default formats to format each expression. The format, particularly field widths for different types, is compiler dependent.

```
print *, expressions
```

10.4.2.14.4 User formatted output Consider:

```
print '(formats)', expressions
```

or

```
print label, expressions
label format (formats)
```

User-formatted output uses fixed width fields, specified in formats, which is a comma separated list of the format codes below. All fields are right aligned. Unlike C's printf, if a value doesn't fit in the field, it is printed as `****` asterisks, instead of expanding the width.

Format	Description
`Iw`	Print integer in `w` characters
`Fw.d`	Print real in `w` characters with `d` decimal places
`Ew.d`	Print real in exponent format, in `w` characters with `d` decimal places in mantissa and 4 characters for exponent
`Aw`	Print string in `w` characters
`A`	Print string in `len(variable)` characters
`Lw`	Print `w-1` blanks, then `T` or `F` for logical
`wX`	Print `w` blanks
`Tc`	Move ("tab") to absolute position `c`
`TLw`	Move left `w` characters
`TRw`	Move right `w` characters
`/`	Start new line

An array name, or array slice, refers to the whole array or slice. So this reads 5 reals into the first row of a matrix, then prints out the first row:

```
real:: A(5,5)
read  *, A(1,:)
print *, A(1,:)
```

10.4.2.14.5 File input/output

Consider the following functions:

```
open ( unit=n, file=filename, status=file_status, iostat=variable )
read ( unit=n, fmt=format,    iostat=variable ) variables
write( unit=n, fmt=format,    iostat=variable ) expressions
close( unit=n )
```

where

status	Description
old	Must already exist
new	Must not exist (like O_EXCL?)
replace	Delete if exists (like O_TRUNC)
scratch	Unnamed temporary, delete when program exits
unknown	Compiler dependent

n is unit number, or * for default input/output unit. format is '(formats)' or label, or * for list-directed input/output. iostat saves error code to the given variable.

10.4.2.14.6 Binary unformatted input/output

OmitS the format specifier. Writes data in a binary format. *Must be read back with exactly the same types of variables, on the same computer architecture* (e.g. with the same endianness and word size.) Example:

```
write( unit=9 ) x, y, z
read( unit=9 ) a, b, c
```

10.4.2.15 Intrinsic functions

Intrinsic functions are listed here.

10.5 More on Fortran

10.5.1 Type declaration statements

COMMON Common storage area for variables that are in several program units.

DATA Puts initial values into variables.

NON_OVERRIDABLE Declares a bound procedure cannot be overridden in a subclass of this class.

`PROCEDURE,NONOVERRIDABLE::pr`.

PARAMETER Defines a named constant.

`REAL,PARAMETER::PI=3.141593`.

ALLOCATABLE Declares an array is allocatable (dyanmic array).

`REAL,ALLOCATABLE,DIMENSION(:)::a = -1`. Use with `allocate()` and `deallocate()`.

DIMENSION Declares the rank and and shape of an array, the dimensions (start and end index) of an array.

`REAL,DIMENSION(-7:10,3:10)::matrix = -1.`

EXTERNAL Declares that a name is a function external to a program unit.

`REAL,EXTERNAL::fun1.`

INTENT Specifies the intended use of a dummy argument.

`REAL,INTENT(IN)::ndim.`

INTRINSIC Declares that a name is a specific intrinsic function

`REAL,INTRINSIC::sin`

NOPASS Declares a bound procedure cannot be overridden in a subclass of this class.

`PROCEDURE,NOPASS::add`

OPTIONAL Declares that a dummey argument is optional.

`REAL,OPTIONAL,INTENT(IN)::maxval`

POINTER Declares that a variable is a pointer.

`INTEGER,POINTER::ptr`

PRIVATE Declares that an object is private to a module.

`REAL,PRIVATE::internal_data`

PROTECTED Declares that an object in a module is protected, meaning that it can be used but not modified outside the module in which it is defined.

`REAL,PROTECTED::x`

PUBLIC Declares that an object is public to a module.

`REAL,PUBLIC::cir=2.54`

SAVE Declares that the value of a variable must be preserved across different calls to the same subroutine/-function.

`REAL,SAVE::sum SAVE`

TARGET Specifies that an object can become the target of a pointer (it can be pointed to).

`REAL,TARGET::val1`

VOLATILE Declares that a value of a variable might be changed at any time by some source external to the program.

`REAL,VOLATILE::vol1`

10.5.2 Others

- `1.3D3` is a double precision number, while `1.3E3` is a single precision
- `DEXP` is `EXP` but for `REAL(8)`

10.6 Importing additional materials

P.-Y. Meslin suggested superimposing data from Fanale and Cannon (1974), Anderson, Schwarz, and Tice (1978), Jänchen et al. (2006), CHEVRIER, OSTROWSKI, and SEARS (2008), and Jänchen et al. (2009) on the plots of Pommerol et al. (2009).

The WebPlotDigitizer (in GitHub) allows one to upload a picture of a plot and extract the actual values of the data presented in a CSV file.

The imports are defined below:

```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.colors as mcolors
```

```
# Set matplotlib style
plt.style.use("ggplot")
```

```
def rotate(li, x):
    return li[-x % len(li):] + li[:-x % len(li)]
```

A regolith plotting function is defined:

```
def plotRegolith(regolith, xtag, ytag, title=None, **kwargs):
    f, ax = plt.subplots()
    for label, isotherm in regolith.items():
        ax.plot(isotherm[xtag], isotherm[ytag], label=label, **kwargs)

    ax.legend(loc="best")
    ax.set_xlabel(xtag)
    ax.set_ylabel(ytag)
    ax.set_title(title)

    return ax
```

10.6.1 Data from FANALE and CANNON (1971)

Table 1 in FANALE and CANNON (1971) presents values for the Vacaville basalt at 29 °C and $SSA = 5.8\,\mathrm{m}^2/\mathrm{g}$.

```
vcvlle_1 = dict()

Fanale1971_SSA = 5.8e3 # [m^2/kg]

Fanale1971_Ts = [302]

colnames = ["x", "mg_h2o_ads_over_g_rock", "micromol_h2o_ads_over_g_rock_SSA", "
    monolayers_ads"]

vcvlle_1["ads_302"]= pd.read_csv("./data/Fanale1971/h2o_ads_302.csv", header=None, names
    =colnames).to_dict('list')

Fanale1971_materials = [vcvlle_1]

for Fanale1971_material in Fanale1971_materials:
    for Fanale1971_iso in Fanale1971_material.values(): # Get lists
        Fanale1971_iso["g_h2o_ads_over_g_rock"] = np.multiply(Fanale1971_iso["
           mg_h2o_ads_over_g_rock"], 1e-3)
        Fanale1971_iso["y"] = np.divide(Fanale1971_iso["g_h2o_ads_over_g_rock"],
           Fanale1971_SSA) # [mass ratio normalized by SSA]
```

```
plotRegolith(vcvlle_1, "x", "mg_h2o_ads_over_g_rock", title="Vacaville basalt")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fafda6d84f0>
```

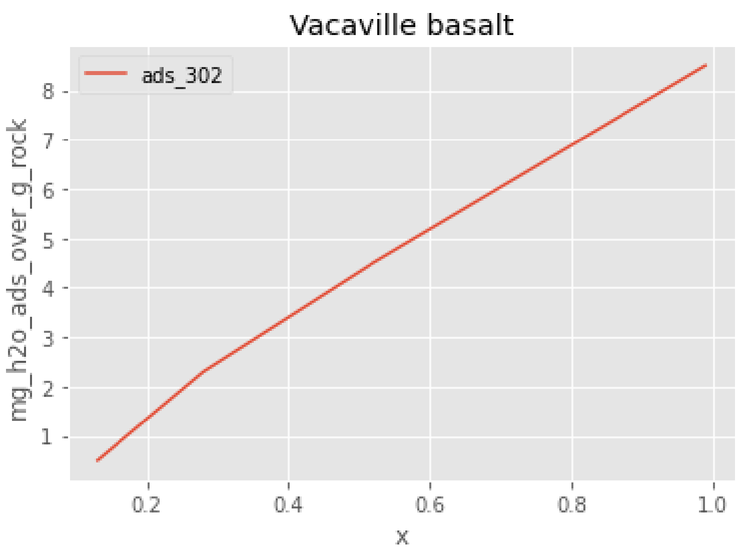


Figure 43: png

10.6.2 Data from Fanale and Cannon (1974)

Figure 1 in Fanale and Cannon (1974) shows water adsorption isotherms at -20, 0, and 25 °C for Vacaville basalt, with a BET $SSA = 9.8\,\mathrm{m}^2/\mathrm{g}$. As seen in the plot, isotherms at 0 and 25 °C follow—approximately—the same curve.

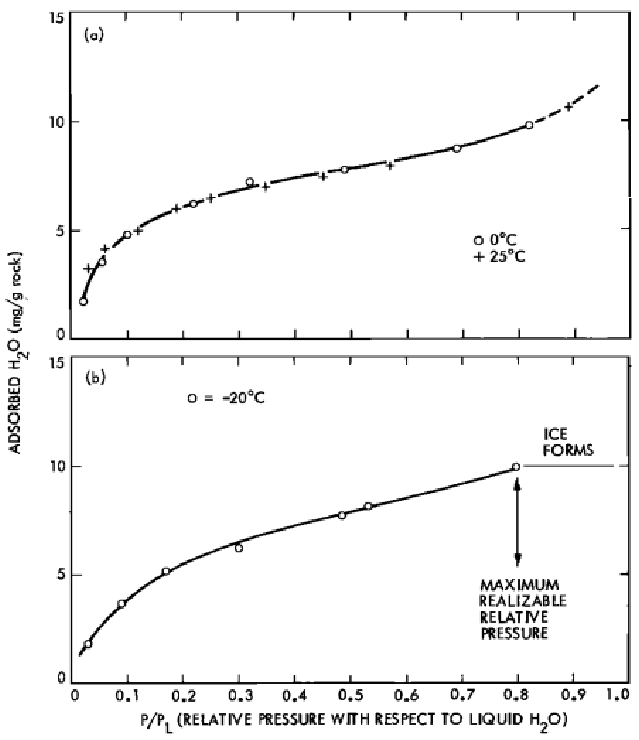


Fig. 1. H_2O adsorption isotherms at (*a*) 0° and 25°C and (*b*) −20°C. The mass of adsorbed water (milligrams) per gram of rock mass is plotted versus temperature. Note that all isotherms are almost superimposable and that the main limitation on adsorption in systems below the triple point is the distance along the isotherm one can proceed before the maximum realizable relative pressure is achieved. The curve in *b* extends to a P/P_0 of 1.0 when the system is in equilibrium with ice. But this corresponds to a relative pressure with respect to supercooled water (P/P_L) of only 0.8. This is termed the maximum realizable relative pressure. Beyond this point the only result of adding H_2O molecules to the system in both theory and laboratory practice is to make ice; no increase in the amount of adsorbed water occurs. It should be noted that if the adsorbed layer is sufficiently saline, ice formation might not occur until far below 0°C, and the adsorption curve might continue farther to the right than shown in *b*.

Figure 44: Istoherms from Fanale and Cannon (1974)

```python
vcvlle = dict()

Fanale1974_SSA = 9.8e3 # [m^2/kg]

Fanale1974_Ts = [253, 273, 298]

colnames = ["x", "mg_h2o_ads_over_g_rock"]

vcvlle["ads_253"]= pd.read_csv("./data/Fanale1974/h2o_ads_-20.csv", header=None, names=
    colnames).to_dict('list')
vcvlle["ads_273_and_298"]= pd.read_csv("./data/Fanale1974/h2o_ads_0_and_25.csv", header=
    None, names=colnames).to_dict('list')

Fanale1974_materials = [vcvlle]

for Fanale1974_material in Fanale1974_materials:
    for Fanale1974_iso in Fanale1974_material.values(): # Get lists
        Fanale1974_iso["g_h2o_ads_over_g_rock"] = np.multiply(Fanale1974_iso["
            mg_h2o_ads_over_g_rock"], 1e-3)
        Fanale1974_iso["y"] = np.divide(Fanale1974_iso["g_h2o_ads_over_g_rock"],
            Fanale1974_SSA) # [mass ratio normalized by SSA]
```

```
plotRegolith(vcvlle, "x", "mg_h2o_ads_over_g_rock", title="Vacaville basalt")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fafda6d8fd0>
```

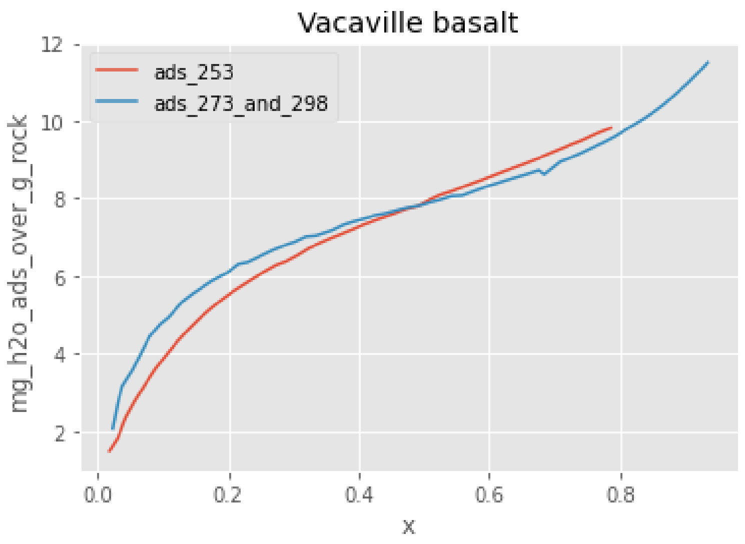


Figure 45: png

10.6.3 Data from Anderson, Schwarz, and Tice (1978)

Table I in Anderson, Schwarz, and Tice (1978) list all the values used in the plots. These correspond to adsorption-desorption data for sodium montmorillonite at $-5\,^{\circ}$C. Each of the isotherms was stored in different sheets of an ODS spreadsheet file. Each of the sheets was converted to CSV files using the custom `ods2csv` script, which calls the `ssconvert` tool from gnumeric.

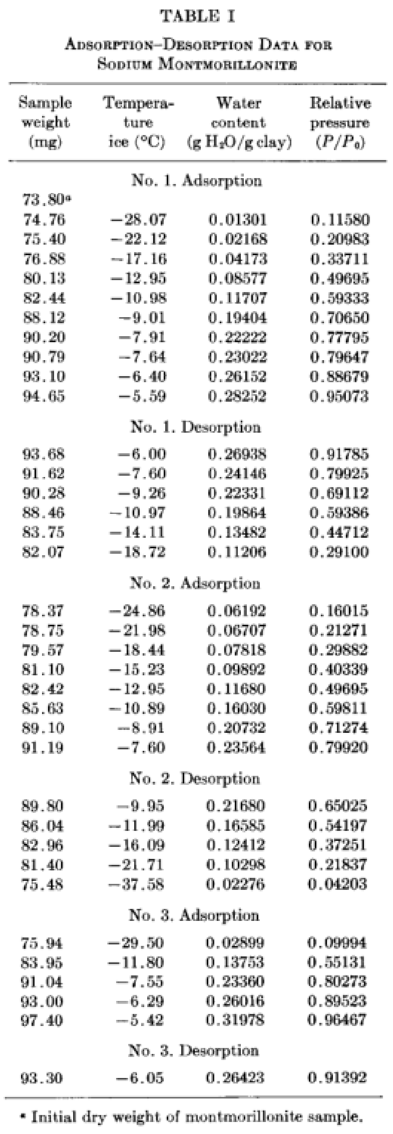


TABLE I

ADSORPTION–DESORPTION DATA FOR SODIUM MONTMORILLONITE

Sample weight (mg)	Temperature ice (°C)	Water content (g H_2O/g clay)	Relative pressure (P/P_0)
No. 1. Adsorption			
73.80[a]			
74.76	−28.07	0.01301	0.11580
75.40	−22.12	0.02168	0.20983
76.88	−17.16	0.04173	0.33711
80.13	−12.95	0.08577	0.49695
82.44	−10.98	0.11707	0.59333
88.12	−9.01	0.19404	0.70650
90.20	−7.91	0.22222	0.77795
90.79	−7.64	0.23022	0.79647
93.10	−6.40	0.26152	0.88679
94.65	−5.59	0.28252	0.95073
No. 1. Desorption			
93.68	−6.00	0.26938	0.91785
91.62	−7.60	0.24146	0.79925
90.28	−9.26	0.22331	0.69112
88.46	−10.97	0.19864	0.59386
83.75	−14.11	0.13482	0.44712
82.07	−18.72	0.11206	0.29100
No. 2. Adsorption			
78.37	−24.86	0.06192	0.16015
78.75	−21.98	0.06707	0.21271
79.57	−18.44	0.07818	0.29882
81.10	−15.23	0.09892	0.40339
82.42	−12.95	0.11680	0.49695
85.63	−10.89	0.16030	0.59811
89.10	−8.91	0.20732	0.71274
91.19	−7.60	0.23564	0.79920
No. 2. Desorption			
89.80	−9.95	0.21680	0.65025
86.04	−11.99	0.16585	0.54197
82.96	−16.09	0.12412	0.37251
81.40	−21.71	0.10298	0.21837
75.48	−37.58	0.02276	0.04203
No. 3. Adsorption			
75.94	−29.50	0.02899	0.09994
83.95	−11.80	0.13753	0.55131
91.04	−7.55	0.23360	0.80273
93.00	−6.29	0.26016	0.89523
97.40	−5.42	0.31978	0.96467
No. 3. Desorption			
93.30	−6.05	0.26423	0.91392

[a] Initial dry weight of montmorillonite sample.

Figure 46: Istoherms from Anderson, Schwarz, and Tice (1978)

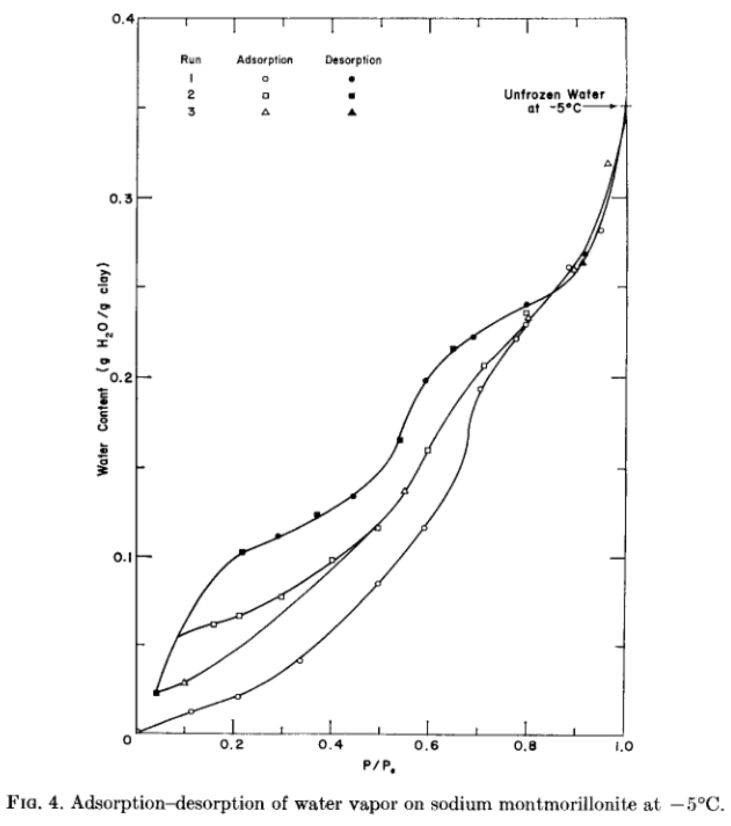


Figure 47: Istoherms from Anderson, Schwarz, and Tice (1978)

```
mmllote_1 = dict()

Anderson1978_SSA = 1060e3 # [m^2/kg]

Anderson1978_Ts = [268]

for i in range(6): # Files are stored as h2o_ads_des_-5.csv.$num, with num going from 0
    to 5 (6 files)
    mmllote_1[i]= pd.read_csv("./data/Anderson1978/h2o_ads_des_-5.csv."+str(i)).to_dict(
        'list')

# Optional rename keys
mmllote_1["ads1_268"] = mmllote_1.pop(0)
mmllote_1["des1_268"] = mmllote_1.pop(1)
mmllote_1["ads2_268"] = mmllote_1.pop(2)
mmllote_1["des2_268"] = mmllote_1.pop(3)
mmllote_1["ads3_268"] = mmllote_1.pop(4)
mmllote_1["des3_268"] = mmllote_1.pop(5)

Anderson1978_materials = [mmllote_1]

for Anderson1978_material in Anderson1978_materials:
    for Anderson1978_iso in Anderson1978_material.values(): # Get lists
        Anderson1978_iso["y"] = np.divide(Anderson1978_iso["h2o_g_over_clay_g"],
            Anderson1978_SSA) # [mass ratio normalized by SSA]
```

```
plotRegolith(mmllote_1, "x", "h2o_g_over_clay_g", title="Sodium Montmorillonite")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fafd7ee8c10>
```

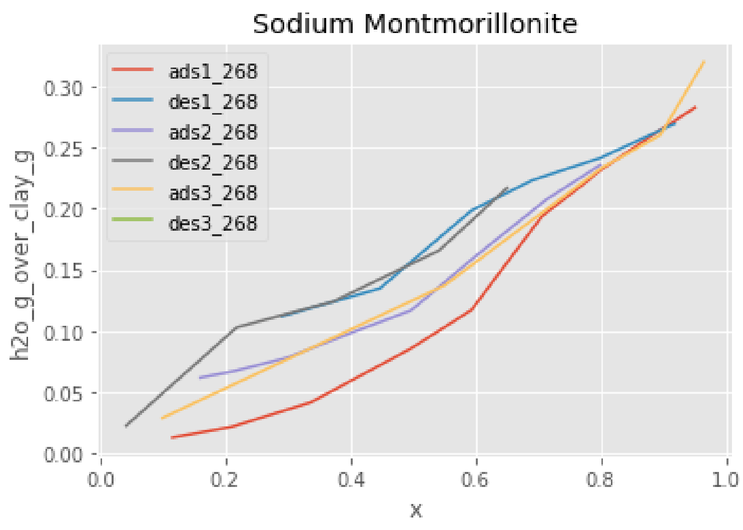


Figure 48: png

10.6.4 Data from Jänchen et al. (2006)

In Jänchen et al. (2006) isotherms at different temperatures are presented for different materials:

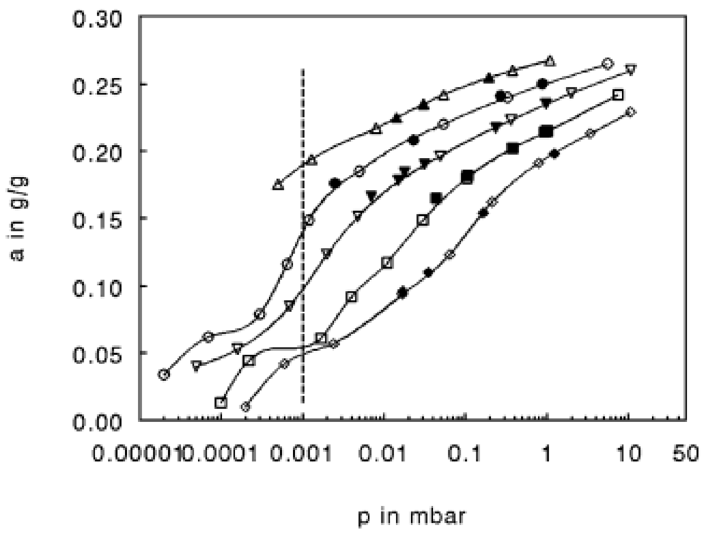


Fig. 1. Water adsorption isotherms for chabazite at 257, 275, 293, 313 and 333 K; filled symbols denote desorption, the dashed line indicates the relative water pressure on the martian surface.

Figure 49: Figure 1 from Jänchen et al. (2006)

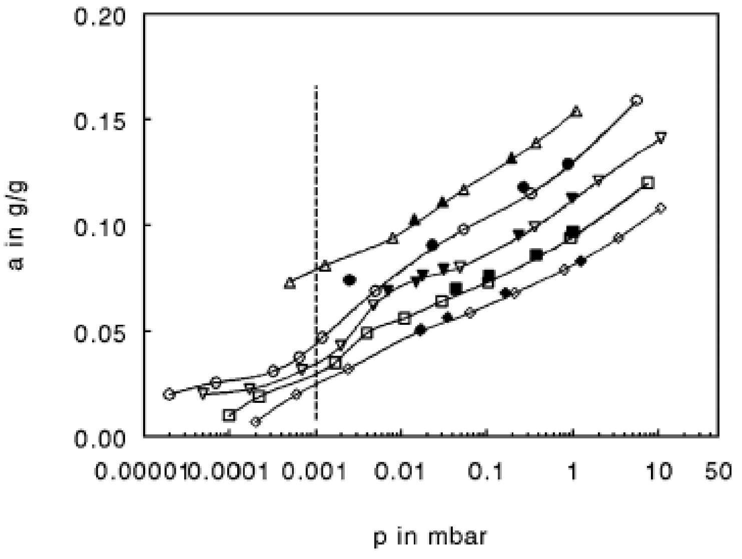


Fig. 2. Water adsorption isotherms for clinoptilolite at 257, 275, 293, 313 and 333 K; filled symbols denote desorption, the dashed line indicates the relative water pressure on the martian surface.

Figure 50: Figure 2 from Jänchen et al. (2006)

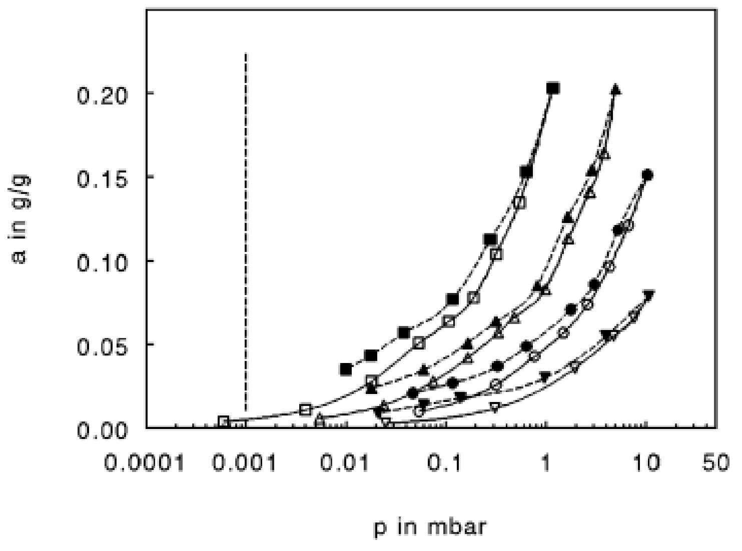


Fig. 3. Water adsorption isotherms for montmorillonite at 257, 275, 293 and 313 K; filled symbols denote desorption, the dashed line indicates the relative water pressure on the martian surface.

Figure 51: Figure 3 from Jänchen et al. (2006)

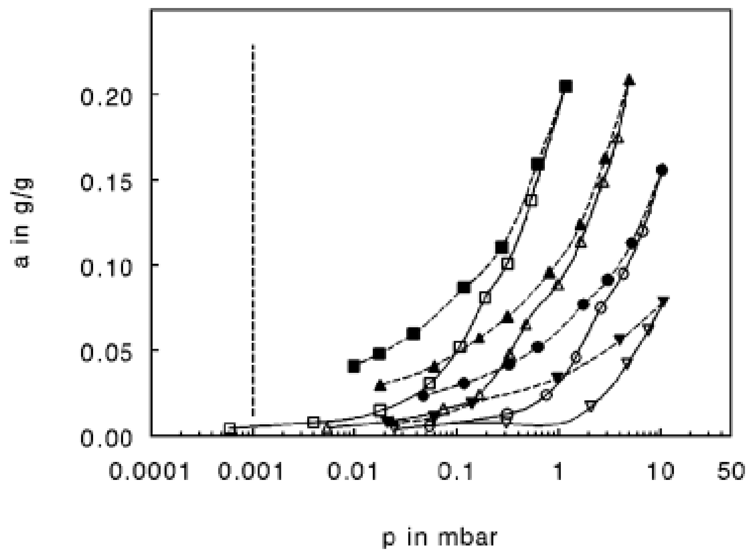


Fig. 4. Water adsorption isotherms for nontronite at 257, 275, 293 and 313 K; filled symbols denote desorption, the dashed line indicates the relative water pressure on the martian surface.

Figure 52: Figure 4 from Jänchen et al. (2006)

```
mmllote = dict()

Janchen2009_dens_H2O_ice = 920. # [kg/m^3] TODO It actually depends on temperature
t_monolayer = 3e-10

# Dubinin defines the specific volume W as W = a/dens_H2O_ice, where a is theadsorbed
    amount (kg/kg rego)
# SSA = a/dens_H2O_ice with t_monolayer = 3e-10

Janchen2006_SSA_chbzite = ( 0.270 / Janchen2009_dens_H2O_ice ) / t_monolayer
Janchen2006_SSA_clolite = ( 0.090 / Janchen2009_dens_H2O_ice ) / t_monolayer
Janchen2006_SSA_mmllote_2 = 74e3 # ( 0.230 / Janchen2009_dens_H2O_ice ) / t_monolayer
Janchen2006_SSA_nntrite = 66e3 # ( 0.173 / Janchen2009_dens_H2O_ice ) / t_monolayer

Janchen2006_Ts_1_2 = [257, 275, 293, 313, 333] # Temps for which isotherms are available
Janchen2006_Ts_3_4 = [257, 275, 293, 313] # Temps for which isotherms are available
Janchen2006_Ts = sorted(list(set(Janchen2006_Ts_3_4).union(Janchen2006_Ts_1_2)))
# print(Janchen2006_Ts)

colnames = ["p_mbar", "g_h2o_ads_over_g_rock"]

# Figure 1
chbzite = dict([("ads_"+str(Janchen2006_T),pd.read_csv("./data/Janchen2006/h2o_ads_fig1_
    "+str(Janchen2006_T)+".csv", header=None, names=colnames).to_dict('list')) for
    Janchen2006_T in Janchen2006_Ts_1_2])

# Figure 2
clolite = dict([("ads_"+str(Janchen2006_T),pd.read_csv("./data/Janchen2006/h2o_ads_fig2_
    "+str(Janchen2006_T)+".csv", header=None, names=colnames).to_dict('list')) for
    Janchen2006_T in Janchen2006_Ts_1_2])

# Figure 3
mmllote_2 = dict([("ads_"+str(Janchen2006_T),pd.read_csv("./data/Janchen2006/
    h2o_ads_fig3_"+str(Janchen2006_T)+".csv", header=None, names=colnames).to_dict('list
    ')) for Janchen2006_T in Janchen2006_Ts_3_4])

# Figure 4
nntrite = dict([("ads_"+str(Janchen2006_T),pd.read_csv("./data/Janchen2006/h2o_ads_fig4_
    "+str(Janchen2006_T)+".csv", header=None, names=colnames).to_dict('list')) for
    Janchen2006_T in Janchen2006_Ts_3_4])

for chbzite_iso in chbzite.values():
    chbzite_iso["y"] = np.divide(chbzite_iso["g_h2o_ads_over_g_rock"],
        Janchen2006_SSA_chbzite) # [mass ratio normalized by SSA]

for clolite_iso in clolite.values():
    clolite_iso["y"] = np.divide(clolite_iso["g_h2o_ads_over_g_rock"],
        Janchen2006_SSA_clolite) # [mass ratio normalized by SSA]

for mmllote_2_iso in mmllote_2.values():
    mmllote_2_iso["y"] = np.divide(mmllote_2_iso["g_h2o_ads_over_g_rock"],
        Janchen2006_SSA_mmllote_2) # [mass ratio normalized by SSA]

for nntrite_iso in nntrite.values():
    nntrite_iso["y"] = np.divide(nntrite_iso["g_h2o_ads_over_g_rock"],
        Janchen2006_SSA_nntrite) # [mass ratio normalized by SSA]

Janchen2006_materials = [chbzite, clolite, mmllote_2, nntrite]

for Janchen2006_material in Janchen2006_materials:
    for Janchen2006_iso in Janchen2006_material.values(): # Get lists
        Janchen2006_iso["p"] = np.multiply(Janchen2006_iso["p_mbar"], 100) # [Pa]
```

```
plotRegolith(chbzite, "p_mbar", "g_h2o_ads_over_g_rock", title="Chabazite").semilogx()
```

```
[]
```

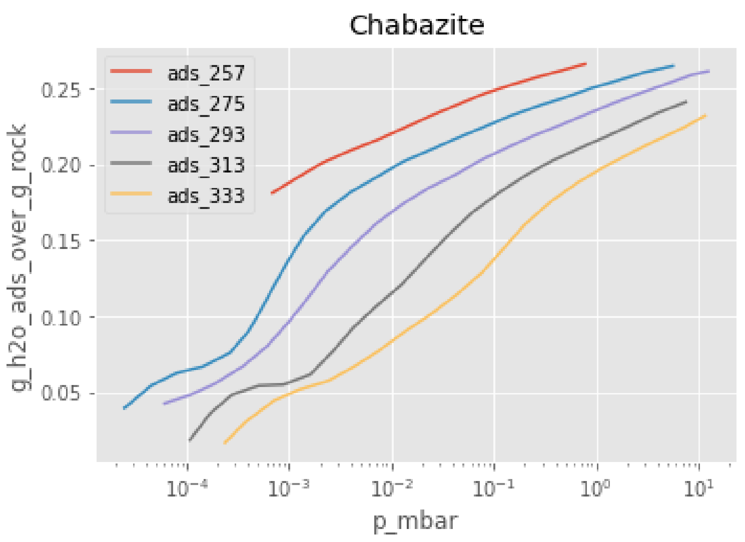


Figure 53: png

```
plotRegolith(clolite, "p_mbar", "g_h2o_ads_over_g_rock", title="Clinoptilolite").
    semilogx()
```

```
[]
```

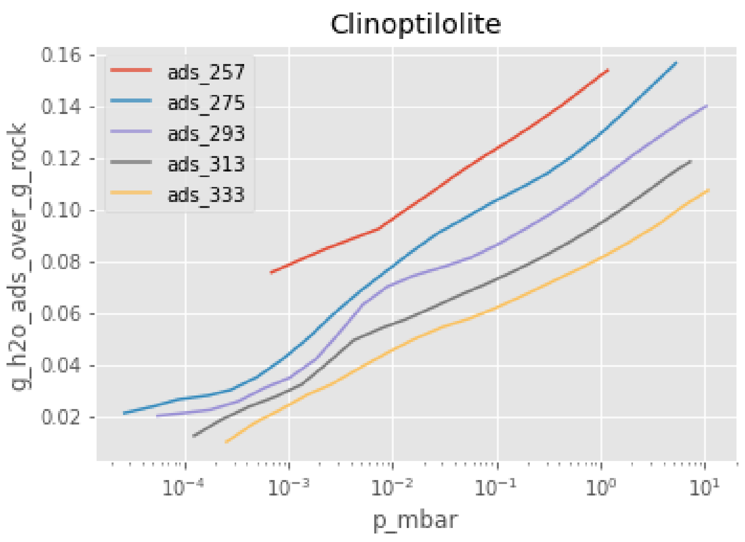


Figure 54: png

```
plotRegolith(mmllote_2, "p_mbar", "g_h2o_ads_over_g_rock", title="Montmorillonite").
    semilogx()
```

```
[]
```

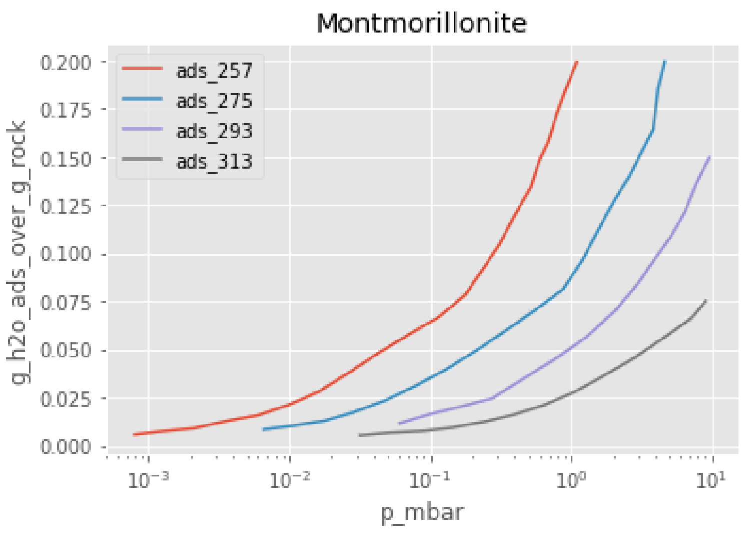


Figure 55: png

```
plotRegolith(nntrite, "p_mbar", "g_h2o_ads_over_g_rock", title="Nontronite").semilogx()
```

```
[]
```

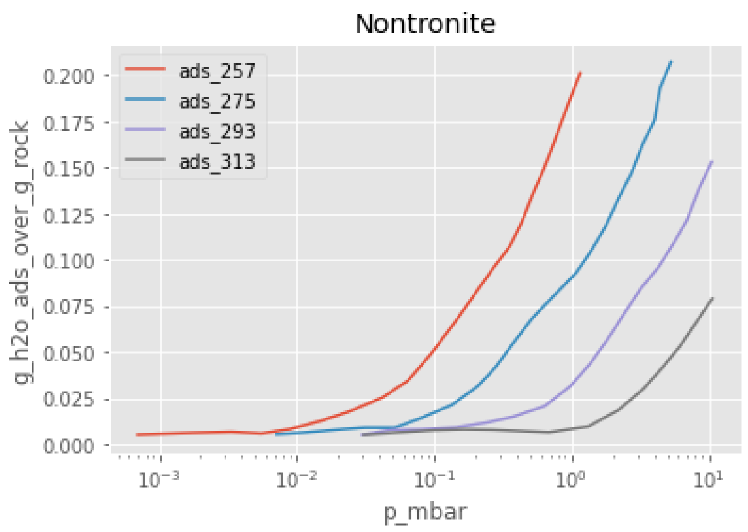


Figure 56: png

10.6.5 Data from CHEVRIER, OSTROWSKI, and SEARS (2008)

In CHEVRIER, OSTROWSKI, and SEARS (2008) the authors' results are compared to those from Zent and Quinn (1997) and more:

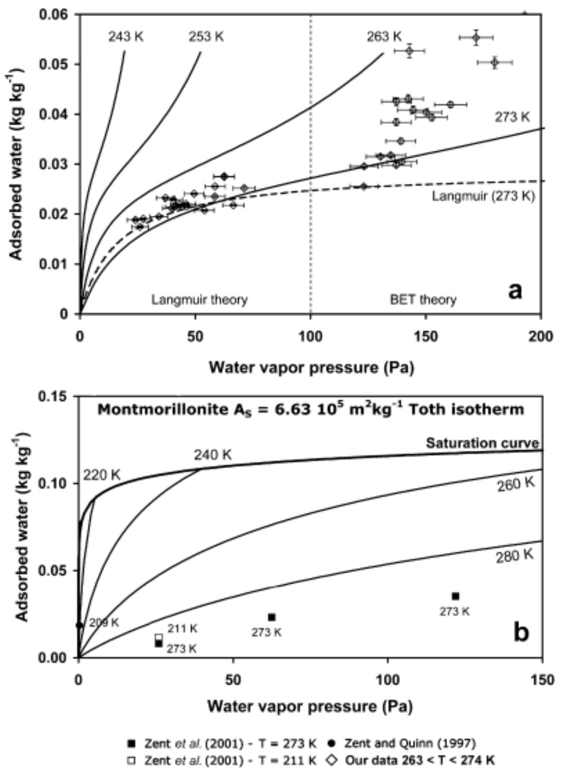


Fig. 10. Isotherms for water adsorbing onto clay particles derived from the present experiments compared with experimental data from the present and earlier studies. (a) Isotherms determined from our data, using both the Langmuir theory at 273 K (dashed line) and the BET theory at various temperatures (solid lines). The limit of application of the Langmuir theory (100 Pa) is indicated as a vertical dashed line. The diamonds represent the present data with their experimental uncertainties. (b) Isotherms determined by Zent and Quinn (1997), using the Toth theory and the specific surface area $A_S = 6.63 \times 10^5$ m^2 kg^{-1}. Are also presented the data used by Zent and Quinn (1997) for the determination of the isotherm, and later data obtained by Zent et al. (2001). The thick black line labeled "saturation curve" corresponds to the formation of the liquid phase. Although the amount of adsorbed water is very similar between both sets of experiments, the surface area of Zent's clays is about 6 to 7 times larger than ours. This suggests differences in grain size and/or effect of CO_2 adsorption in our experiments.

Figure 57: Figure 6 from CHEVRIER, OSTROWSKI, and SEARS (2008)

```
mmllote = dict()

Chevrier2008_SSA_chevrier = 9e4 # Alternative: 1.1e5 [m^2/kg]
Chevrier2008_SSA_zent = 6.63e5 # [m^2/kg]

Chevrier2008_Ts = [243, 253, 263, 273] # Temps for which isotherms are available
Zent1997_Ts = [220, 240, 260, 280] # Temps for which isotherms are available

colnames = ["p_Pa", "kg_h2o_ads_over_kg"]

clay_chevrier = dict([("ads_"+str(Chevrier2008_T),pd.read_csv("./data/Chevrier2008/
    h2o_ads_chevrier_"+str(Chevrier2008_T)+".csv", header=None, names=colnames).to_dict(
    'list')) for Chevrier2008_T in Chevrier2008_Ts])
clay_zent = dict([("ads_"+str(Zent1997_T),pd.read_csv("./data/Chevrier2008/h2o_ads_zent_
    "+str(Zent1997_T)+".csv", header=None, names=colnames).to_dict('list')) for
    Zent1997_T in Zent1997_Ts])

for clay_chevrier_iso in clay_chevrier.values():
    clay_chevrier_iso["y"] = np.divide(clay_chevrier_iso["kg_h2o_ads_over_kg"],
        Chevrier2008_SSA_chevrier) # [mass ratio normalized by SSA]

for clay_zent_iso in clay_zent.values():
    clay_zent_iso["y"] = np.divide(clay_zent_iso["kg_h2o_ads_over_kg"],
        Chevrier2008_SSA_zent) # [mass ratio normalized by SSA]

Chevrier2008_materials = [clay_chevrier, clay_zent]

for Chevrier2008_material in Chevrier2008_materials:
    for Chevrier2008_iso in Chevrier2008_material.values(): # Get lists
        Chevrier2008_iso["p"] = Chevrier2008_iso["p_Pa"] # [Pa]
```

```
plotRegolith(clay_chevrier, "p_Pa", "kg_h2o_ads_over_kg",title="Chevrier2008")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fafd7ad1fd0>
```

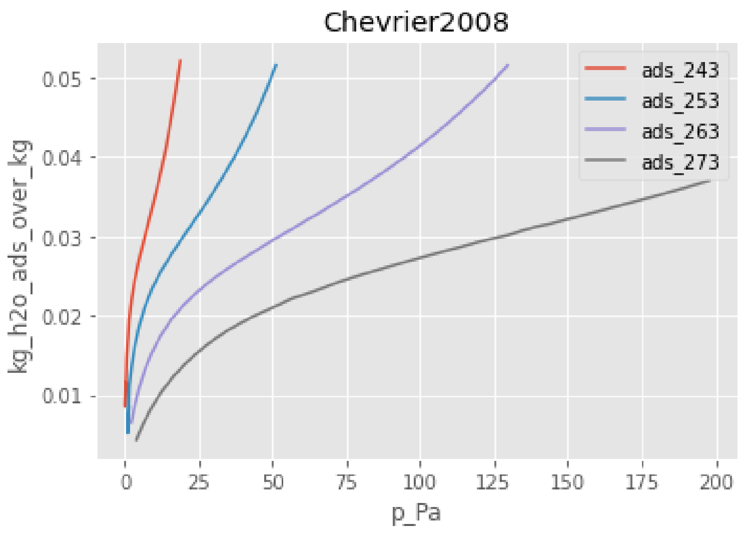


Figure 58: png

```
plotRegolith(clay_zent, "p_Pa", "kg_h2o_ads_over_kg", title="Zent1997 and more")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fafd7b00cd0>
```

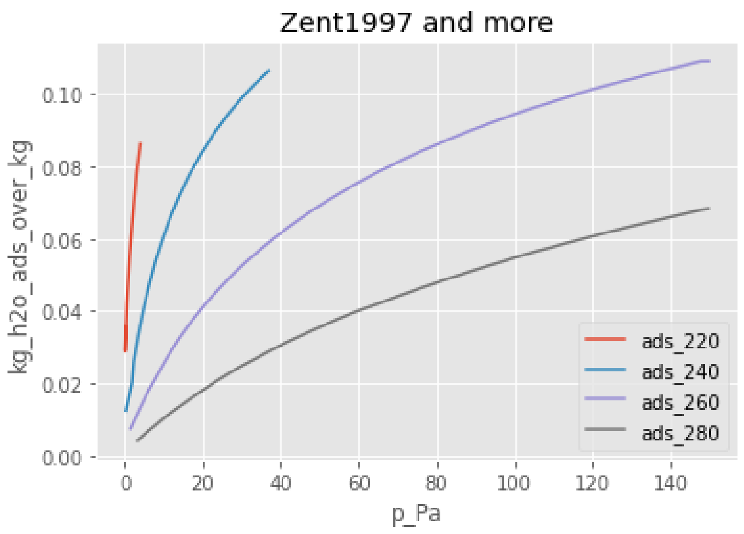


Figure 59: png

10.6.6 Data from Jänchen et al. (2009)

In Jänchen et al. (2009) isotherms at different temperatures are presented for HWMK919, as well as for HWMK919 and NG1 at 293 K:

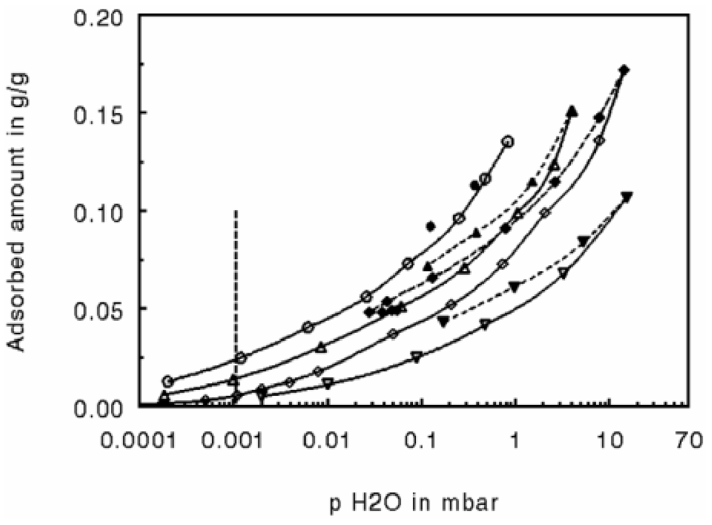


Fig. 5. H_2O adsorption (open symbols) and desorption (closed symbols) isotherms for HWMK919 <5 μm at 255 (circles), 273 (triangles), 293 (diamonds), and 313 K (inverted triangles), adsorbed amount, *a*, in g H_2O/g HWMK919, the dashed line indicates the H_2O vapor pressure on the martian surface.

Figure 60: Figure 5 from Jänchen et al. (2009)

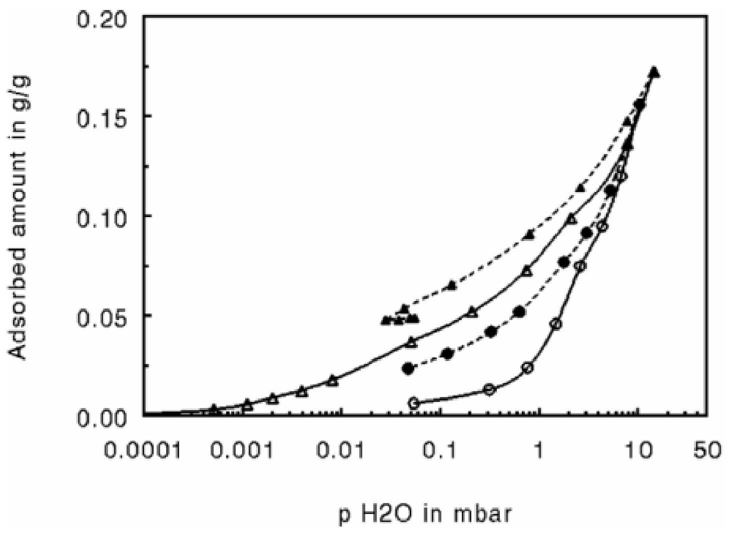


Fig. 6. H_2O adsorption/desorption isotherms for <5 μm HWMK919 (triangles) and NG-1 (circles) at 293 K after degassing at 383 K in high vacuum, filled symbols denote desorption, adsorbed amount, *a*, in g H_2O/g mineral.

Figure 61: Figure 6 from Jänchen et al. (2009)

```
mmllote = dict()

Janchen2009_SSA_HWMK919 = 203e3 # [m^2/kg]
Janchen2009_SSA_NG1 = 66e3 # [m^2/kg]

Janchen2009_Ts_5 = [255, 273, 293, 313] # Temps for which isotherms are available
```

```
Janchen2009_Ts_6 = [293]
Janchen2009_Ts = sorted(list(set(Janchen2009_Ts_5).union(Janchen2009_Ts_6)))
# print(Janchen2009_Ts)

colnames = ["p_mbar", "g_h2o_ads_over_g_rock"]

HWMK919_1 = dict([("ads_"+str(Janchen2009_T),pd.read_csv("./data/Janchen2009/
    h2o_ads_fig5_"+str(Janchen2009_T)+".csv", header=None, names=colnames).to_dict('list
    ')) for Janchen2009_T in Janchen2009_Ts_5])
HWMK919_2 = dict([("ads_"+str(Janchen2009_T)+"_2",pd.read_csv("./data/Janchen2009/
    h2o_ads_fig6_"+str(Janchen2009_T)+"_HWMK919.csv", header=None, names=colnames).
    to_dict('list')) for Janchen2009_T in Janchen2009_Ts_6])
HWMK919 = {**HWMK919_1, **HWMK919_2} # Merge dicts

NG1 = dict([("ads_"+str(Janchen2009_T),pd.read_csv("./data/Janchen2009/h2o_ads_fig6_"+
    str(Janchen2009_T)+"_NG1.csv", header=None, names=["p_mbar", "g_h2o_ads_over_g_rock"
    ]).to_dict('list')) for Janchen2009_T in Janchen2009_Ts_6])
NG1_HWMK919 = {**NG1, **HWMK919_2} # Merge dicts

for HWMK919_iso in HWMK919.values():
    HWMK919_iso["y"] = np.divide(HWMK919_iso["g_h2o_ads_over_g_rock"],
        Janchen2009_SSA_HWMK919) # [mass ratio normalized by SSA]

for NG1_iso in NG1.values():
    NG1_iso["y"] = np.divide(NG1_iso["g_h2o_ads_over_g_rock"], Janchen2009_SSA_NG1) # [
        mass ratio normalized by SSA]

Janchen2009_materials = [HWMK919, NG1]

for Janchen2009_material in Janchen2009_materials:
    for Janchen2009_iso in Janchen2009_material.values(): # Get lists
        Janchen2009_iso["p"] = np.multiply(Janchen2009_iso["p_mbar"], 100) # [Pa]
```

```
plotRegolith(HWMK919, "p_mbar", "g_h2o_ads_over_g_rock", title="HWMK919").semilogx()
```

```
[]
```

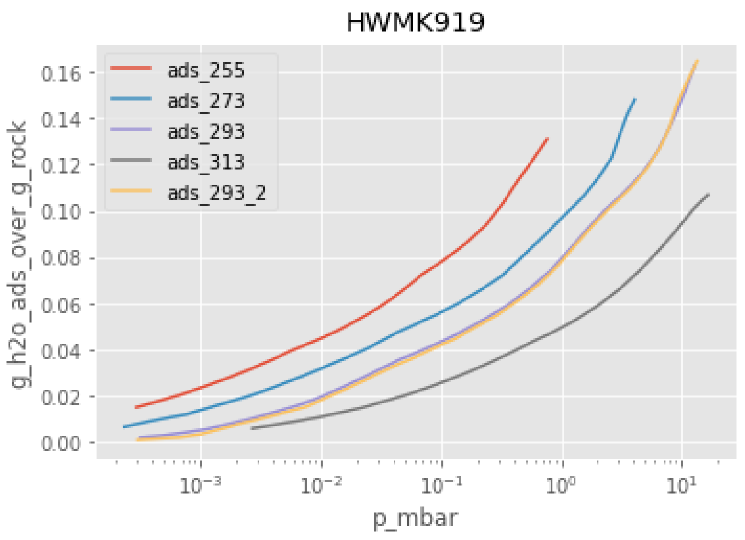


Figure 62: png

```
plotRegolith(NG1_HWMK919, "p_mbar", "g_h2o_ads_over_g_rock", title="HWMK919 and NG1").
    semilogx()
```

```
[]
```

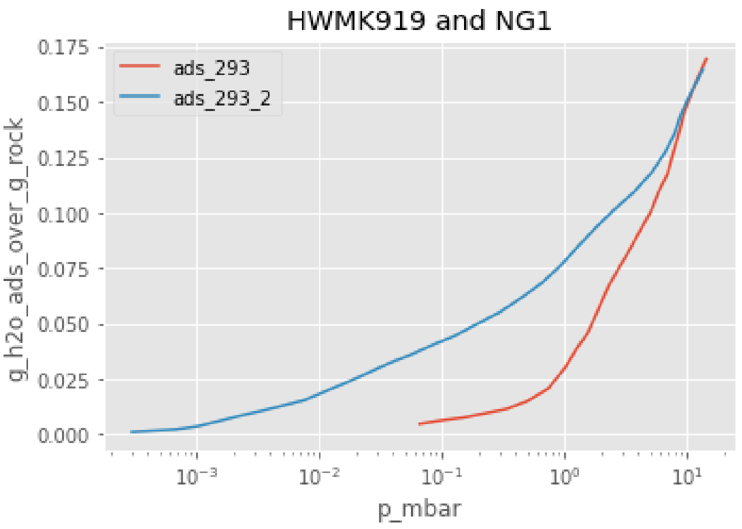


Figure 63: png

10.6.7 Uniformizing data

The pressure data from Jänchen et al. (2006), CHEVRIER, OSTROWSKI, and SEARS (2008), and@Jnchen2009, has to be divided by the water saturation pressure using the Clasius–Clapeyron relation so as to obtain the desired values.

```
Tref = 243. # [K] Isotherm data from Pommerol2009 found at 293 K
Psmbar = 0.37 # [mbar] Saturation pressure at 293K
Ps = Psmbar*100 # [Pa] Saturation pressure at 293
```

```
# Pss = Ps*np.exp((1/Tref - 1/Chevrier2008_Ts))
# print(Pss)
```

This tool may be used to compute saturation pressures.

One may also use the formula from CRC Handbook of Chemistry and Physics (1995–1996 edition) shown in CHEVRIER, OSTROWSKI, and SEARS (2008):

$$\log\left(\mathrm{p_{sat}\ bar}\right) = 7.551 - \frac{2666}{T_s}$$

The best fit comes from Murphy and Koop (2005):

```
def CRC_Psat(T): # [Pa] Returns Ps in Pascal from T in Kelvin
    # return np.power(10, 7.551-2666/T)*1e5 # 1e5 is the conversion factor from bar to
      Pa for function in CRC Handbook of Chemistry and Physics
    # return np.exp(28.9074 - 6143.7/T) # Murphy2005
    # return np.exp(54.842763 - 6763.22/T - 4.21*np.log(T) + 0.000367*T + np.tanh(0.0415
       (T - 218.8)) (53.878 - 1331.22/T + np.log(T)*(-9.44523) + 0.014025*T)) #
      Murphy2005
    return np.exp(9.550426 - 5723.265/T + 3.53068*np.log(T) - 0.00728332*T) # Murphy2005
```

```
Fanale1971_Pss = dict(zip(Fanale1971_Ts, [CRC_Psat(T) for T in Fanale1971_Ts]))
```

```
Fanale1974_Pss = dict(zip(Fanale1974_Ts, [CRC_Psat(T) for T in Fanale1974_Ts]))
```

```
Anderson1978_Pss = dict(zip(Anderson1978_Ts, [CRC_Psat(T) for T in Anderson1978_Ts]))
```

```
# Janchen2006_Pss = dict(zip(Janchen2006_Ts, [Ps, 692.579, 2307.98, 7299.81, 19732.6]))
    # TODO No saturation pressures under 273 were found
Janchen2006_Pss = dict(zip(Janchen2006_Ts, [CRC_Psat(T) for T in Janchen2006_Ts]))
# print(Janchen2006_Pss)
```

```
# Chevrier2008_Pss = dict(zip(Chevrier2008_Ts, [Ps, Ps, Ps, Ps])) # TODO No saturation
    pressures under 273 were found
Chevrier2008_Pss = dict(zip(Chevrier2008_Ts, [CRC_Psat(T) for T in Chevrier2008_Ts]))
# print(Chevrier2008_Pss)
```

```
# Zent1997_Pss = dict(zip(Zent1997_Ts, [Ps, Ps, Ps, 985.23])) # TODO No saturation
    pressures under 273 were found
Zent1997_Pss = dict(zip(Zent1997_Ts, [CRC_Psat(T) for T in Zent1997_Ts]))
# print(Zent1997_Pss)
```

```
# Janchen2009_Pss = dict(zip(Janchen2009_Ts, [Ps, Ps, 2307.98, 7299.81, 19732.6])) #
    TODO No saturation pressures under 273 were found
Janchen2009_Pss = dict(zip(Janchen2009_Ts, [CRC_Psat(T) for T in Janchen2009_Ts]))
# print(Janchen2009_Pss)
```

```
Pss = {**Fanale1971_Pss, **Fanale1974_Pss, **Anderson1978_Pss, **Janchen2006_Pss, **
    Chevrier2008_Pss, **Zent1997_Pss, **Janchen2009_Pss}
Pss = dict(sorted(Pss.items())) # Keys are temperatures in [K], values are saturation
    pressures in [Pa]
# print(Pss)
```

Compute the relative pressure **only** for those materials missing this value.

```
additional_materials = Janchen2006_materials + Chevrier2008_materials +
    Janchen2009_materials # [vcvlle, chbzite, clolite, mmllote, nntrite, clay_chevrier,
    clay_zent, HWMK919, NG1]
```

```
for additional_material in additional_materials:
    for add_iso in additional_material.keys():
        for Pss_temp in Pss.keys():
            if str(Pss_temp) in add_iso.split("_"): # Check if the isotherm temperature
                is in Pss
                # print("Current isotherm:", add_iso)
                additional_material[add_iso]["x"] = np.array(additional_material[add_iso
                    ]["p"]) / Pss[Pss_temp] # Relative pressure P/Ps
```

Sodium montmorillonite `mmllote` may now be created from `mmllote_1` and `mmllote_2`:

```
mmllote = {**mmllote_1, **mmllote_2} # Merge dicts
```

All the temperatures are listed here:

10.6.8 Final list of additional materials

The materials that are to be added are listed below:

```
additional_materials = {
    "Vacaville (Fanale1971)": vcvlle_1,
    "Vacaville (Fanale1974)": vcvlle,
    "Chabazite (Janchen2006)": chbzite,
    "Clinoptilolite (Janchen2006)": clolite,
    # "Montmorillonite (Anderson1978 and Janchen2006)": mmllote,
    "Montmorillonite (Anderson1978)": mmllote_1,
    "Montmorillonite (Janchen2006)": mmllote_2,
    "Nontronite (Janchen2006)": nntrite,
    "Clay (Chevrier2008)": clay_chevrier,
    "Clay (Zent1997)": clay_zent,
    "HWMK919 (Jänchen2009)": HWMK919,
    "NG1 (Jänchen2009)": NG1
}
```

These are plotted below:

```
from decimal import Decimal, ROUND_HALF_UP
```

```
n = len(additional_materials) + 1 # Number of plots=every material + all materials
cols = 3
rows = int(Decimal(n/cols).quantize(Decimal('1'), rounding=ROUND_HALF_UP)) # Python's
    round() implements IEEE standard

if cols*rows < n:
    rows=rows + 1

fadd, axadd = plt.subplots(rows, cols, figsize=(15,15), constrained_layout=True)
fadd.suptitle("Additional materials")

if cols*rows-n != 0:
    [fadd.delaxes(axadd[-i, -1]) for i in range(1,cols*rows-n+1)] # Delete last subplots
        if n is odd

colors = list(mcolors.TABLEAU_COLORS.keys()) # color list https://matplotlib.org/3.1.0/
    gallery/color/named_colors.html

i = 0

# First plot
ci = i%rows
cj = int(i/rows)

# New color cycler
# flat_list = [item for sublist in l for item in sublist] # https://stackoverflow.com/
    questions/952914/how-to-make-a-flat-list-out-of-list-of-lists
# l = [additional_material[1].keys() for additional_material in additional_materials.
    items()]
# NUM_COLORS = len([item for sublist in l for item in sublist]) # Number of colors
# cm = plt.get_cmap('tab20')
# axadd[ci, cj].set_prop_cycle('color', [cm(1.*i/NUM_COLORS) for i in range(NUM_COLORS)
    ])

add_patch = []

for curr_add_material, additional_material in additional_materials.items():

    color = colors[0]
    colors = rotate(colors,-1) # shift colors

    for labl, vals in additional_material.items():
        # axadd.plot(vals["x"], vals["y"], label=curr_add_material + " " + labl)
        axadd[ci, cj].plot(vals["x"], vals["y"], color=color)

    add_patch.append(mpatches.Patch(color=color, label=curr_add_material))

axadd[ci, cj].set_yscale('log')
axadd[ci, cj].set_ylim([1e-8, 3e-6])
axadd[ci, cj].set_title("All additional materials")
axadd[ci, cj].set_xlabel("x")
axadd[ci, cj].set_ylabel("wt_ads/SSA")
# axadd.legend(loc="best", bbox_to_anchor=(1, 1))
axadd[ci, cj].legend(handles=[i for i in add_patch], loc="best")
```

```
for curr_add_material, additional_material in additional_materials.items():

    i=i+1

    # Current plot
    ci = i%rows
    cj = int(i/rows)

    # print("i=" + str(i), "ci=" + str(ci), "cj=" + str(cj))

    for labl, vals in additional_material.items():
        for bg_curr_add_material, bg_additional_material in additional_materials.items()
            :
            for bg_labl, bg_vals in bg_additional_material.items():
                if (labl, vals) == (bg_labl, bg_vals):
                    axadd[ci, cj].plot(vals["x"], vals["y"], label=labl, zorder=2)
                else:
                    axadd[ci, cj].plot(bg_vals["x"], bg_vals["y"], color="lightgray",
                        zorder=1)

    axadd[ci, cj].set_yscale('log')
    axadd[ci, cj].set_title(curr_add_material)
    axadd[ci, cj].set_xlabel("P/Ps")
    axadd[ci, cj].set_ylabel("wt_ads/SSA")
    axadd[ci, cj].legend(loc="right")
```

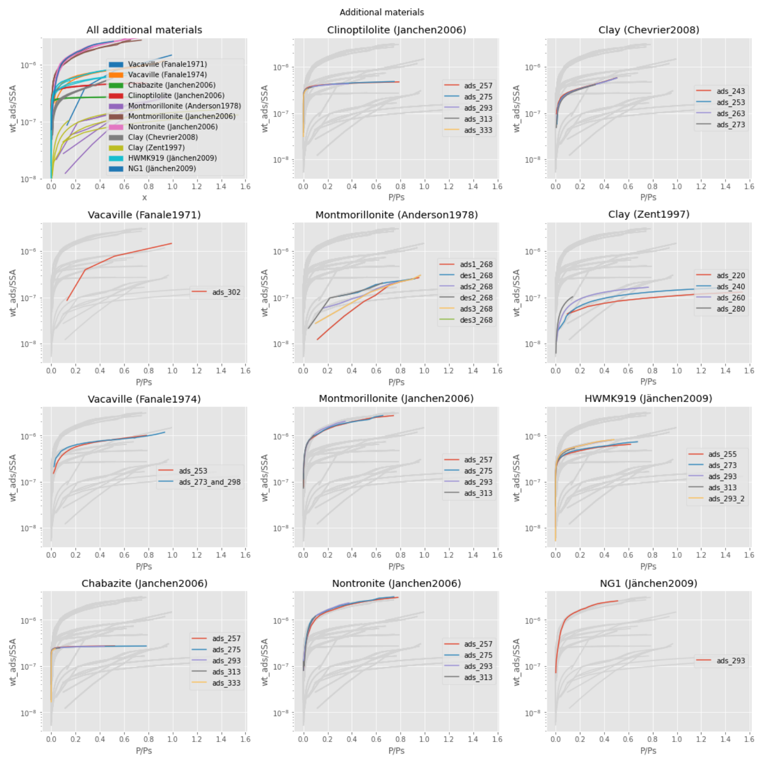


Figure 64: png