# GDD_Calculation_and_Drying_Algorithm

May 6, 2025

[1]:
```
from google.colab import output
output.enable_custom_widget_manager()
```

[2]:
```
!pip install ipywidgets plotly --quiet
```

```
                               1.6/1.6 MB
8.6 MB/s eta 0:00:00
```

Model 1: GDD Calculation

[4]:
```
import urllib.parse
import urllib.request
from io import BytesIO

import pandas as pd
import plotly.express as px
import ipywidgets as widgets
from ipywidgets import VBox, HBox, Layout, Button, Output
from IPython.display import display

import math

# =================================================================
# 1) Function to Fetch Weather Data from Visual Crossing
# =================================================================
def fetch_weather_data_from_api(api_key, city, state, start_date, end_date,
                                unit_group="us",␣
 ↪elements="datetime,tempmin,tempmax"):
    """
    Fetches weather data from Visual Crossing between start_date and end_date
    for a given city, state (e.g. city='Ithaca', state='NY').
    Returns a DataFrame with columns ['datetime','tempmin','tempmax',...].
    """
    location_string = f"{city}, {state}"
    encoded_location = urllib.parse.quote(location_string)
```

```
    base_url = "https://weather.visualcrossing.com/VisualCrossingWebServices/
↪rest/services/timeline"
    url = (
        f"{base_url}/{encoded_location}/{start_date}/{end_date}"
        f"?unitGroup={unit_group}&include=days&elements={elements}"
        f"&key={api_key}&contentType=csv"
    )

    try:
        response = urllib.request.urlopen(url)
        df_api = pd.read_csv(BytesIO(response.read()))
        return df_api
    except urllib.error.HTTPError as e:
        print("HTTP Error:", e.code, e.read().decode())
    except urllib.error.URLError as e:
        print("URL Error:", e.reason)

    return pd.DataFrame()  # Return empty if error


# =================================================================
# 2) GDD Calculation Functions
# =================================================================
def calc_average_gdd(t_min, t_max, T_base):
    return max(((t_max + t_min) / 2) - T_base, 0)

def calc_single_sine_gdd(t_min, t_max, T_base):
    T_mean = (t_max + t_min) / 2
    A = (t_max - t_min) / 2
    if t_max <= T_base:
        return 0.0
    if t_min >= T_base:
        return (T_mean - T_base)
    alpha = (T_base - T_mean) / A
    theta = math.acos(alpha)
    dd = ((T_mean - T_base)*(math.pi - 2*theta) + A*math.sin(2*theta)) / math.pi
    return dd

def calc_single_triangle_gdd(t_min, t_max, T_base):
    if t_max <= T_base:
        return 0.0
    if t_min >= T_base:
        return ((t_max + t_min)/2 - T_base)
    proportion_of_day = (t_max - T_base) / (t_max - t_min)
    avg_above = ((t_max + T_base) / 2) - T_base
    dd = proportion_of_day * avg_above
    return max(dd, 0)
```

```python
def calc_double_sine_gdd(t_min_today, t_max_today, t_min_tomorrow, T_base):
    seg1 = calc_single_sine_gdd(t_min_today, t_max_today, T_base) * 0.5
    seg2 = calc_single_sine_gdd(t_min_tomorrow, t_max_today, T_base) * 0.5
    return seg1 + seg2

def calc_double_triangle_gdd(t_min_today, t_max_today, t_min_tomorrow, T_base):
    seg1 = calc_single_triangle_gdd(t_min_today, t_max_today, T_base) * 0.5
    seg2 = calc_single_triangle_gdd(t_min_tomorrow, t_max_today, T_base) * 0.5
    return seg1 + seg2

def calculate_daily_gdd(row, df,
                        method="average",
                        T_base=50.0,
                        use_modified=False,
                        T_lower=50.0,
                        T_upper=86.0):
    idx = row.name
    t_max = row['tmax']
    t_min = row['tmin']
    if use_modified:
        t_max = min(t_max, T_upper)
        t_min = max(t_min, T_lower)
    if method == "average":
        return calc_average_gdd(t_min, t_max, T_base)
    elif method == "sine":
        return calc_single_sine_gdd(t_min, t_max, T_base)
    elif method == "triangle":
        return calc_single_triangle_gdd(t_min, t_max, T_base)
    elif method == "double_sine":
        if idx < len(df) - 1:
            t_min_next = df.loc[idx+1, 'tmin']
            if use_modified:
                t_min_next = max(t_min_next, T_lower)
            return calc_double_sine_gdd(t_min, t_max, t_min_next, T_base)
        else:
            return calc_single_sine_gdd(t_min, t_max, T_base)
    elif method == "double_triangle":
        if idx < len(df) - 1:
            t_min_next = df.loc[idx+1, 'tmin']
            if use_modified:
                t_min_next = max(t_min_next, T_lower)
            return calc_double_triangle_gdd(t_min, t_max, t_min_next, T_base)
        else:
            return calc_single_triangle_gdd(t_min, t_max, T_base)
    else:
        raise ValueError(f"Unknown method: {method}")
```

```python
def calculate_gdds_for_df(df,
                          start_date,
                          end_date,
                          method="average",
                          T_base=50.0,
                          use_modified=False,
                          T_lower=50.0,
                          T_upper=86.0):
    df = df.copy()
    df['date'] = pd.to_datetime(df['date'])
    df.sort_values('date', inplace=True)
    df['daily_gdd'] = df.apply(
        lambda row: calculate_daily_gdd(row, df,
                                        method=method,
                                        T_base=T_base,
                                        use_modified=use_modified,
                                        T_lower=T_lower,
                                        T_upper=T_upper),
        axis=1
    )
    start_date = pd.to_datetime(start_date)
    end_date = pd.to_datetime(end_date)
    df.loc[df['date'] < start_date, 'daily_gdd'] = 0
    df.loc[df['date'] > end_date, 'daily_gdd'] = 0
    df['cumulative_gdd'] = df['daily_gdd'].cumsum()
    return df

# =================================================================
# 3) Interactive UI
# =================================================================
def interactive_gdd_api_viewer(api_key):
    """
    Provides a UI with separate city/state inputs, and improved layout
    to avoid overlapping or scrunching of widgets.
    """

    city_widget = widgets.Text(
        value="Ithaca",
        description="City:",
        layout=Layout(width="200px")
    )
    state_widget = widgets.Text(
        value="NY",
        description="State:",
        layout=Layout(width="150px")
    )
```

```
    start_date_widget = widgets.Text(
        value="2025-01-01",
        description="Start:",
        layout=Layout(width="180px")
    )
    end_date_widget = widgets.Text(
        value="2025-03-31",
        description="End:",
        layout=Layout(width="180px")
    )
    method_widget = widgets.Dropdown(
        options=["average", "sine", "triangle", "double_sine",␣
↪"double_triangle"],
        value="average",
        description="Method:",
        layout=Layout(width="200px")
    )
    tbase_widget = widgets.FloatText(
        value=50.0,
        description="T_base:",
        layout=Layout(width="150px")
    )
    modified_widget = widgets.Checkbox(
        value=False,
        description="Use Modified?"
    )
    tlower_widget = widgets.FloatText(
        value=50.0,
        description="T_lower:",
        layout=Layout(width="150px", display="none")
    )
    tupper_widget = widgets.FloatText(
        value=86.0,
        description="T_upper:",
        layout=Layout(width="150px", display="none")
    )
    update_button = widgets.Button(
        description="Get Data & Calculate",
        button_style="primary",
        layout=Layout(width="170px")
    )
    plot_output = Output()

    def on_modified_change(change):
        if modified_widget.value:
            tlower_widget.layout.display = "block"
            tupper_widget.layout.display = "block"
```

```python
        else:
            tlower_widget.layout.display = "none"
            tupper_widget.layout.display = "none"

    modified_widget.observe(on_modified_change, names="value")

    def on_update_button_click(b):
        with plot_output:
            plot_output.clear_output(wait=True)

            city_val = city_widget.value
            state_val = state_widget.value
            sdate = start_date_widget.value
            edate = end_date_widget.value

            df_api = fetch_weather_data_from_api(
                api_key=api_key,
                city=city_val,
                state=state_val,
                start_date=sdate,
                end_date=edate,
                unit_group="us",
                elements="datetime,tempmin,tempmax"
            )
            if df_api.empty:
                print("No data returned from API or an error occurred.")
                return

            df_renamed = df_api.copy()
            df_renamed.rename(columns={
                "datetime": "date",
                "tempmax": "tmax",
                "tempmin": "tmin"
            }, inplace=True)

            df_calc = calculate_gdds_for_df(
                df=df_renamed,
                start_date=sdate,
                end_date=edate,
                method=method_widget.value,
                T_base=tbase_widget.value,
                use_modified=modified_widget.value,
                T_lower=tlower_widget.value,
                T_upper=tupper_widget.value
            )

            location_str = f"{city_val}, {state_val}"
```

```python
            fig = px.line(
                df_calc,
                x="date",
                y="cumulative_gdd",
                title=f"Cumulative GDD ({method_widget.value}) for␣
↪{location_str}"
            )
            fig.update_layout(xaxis_title="Date", yaxis_title="Cumulative GDD")
            fig.show()

    update_button.on_click(on_update_button_click)

    # ---------------------
    # Lay out the widgets
    # ---------------------
    # row1: city, state
    row1 = HBox([city_widget, state_widget], layout=Layout(padding="5px"))
    # row2: start_date, end_date
    row2 = HBox([start_date_widget, end_date_widget],␣
↪layout=Layout(padding="5px"))
    # row3: method, T_base, Use Modified?
    row3 = HBox([method_widget, tbase_widget, modified_widget],␣
↪layout=Layout(padding="5px"))
    # row4: T_lower, T_upper, button
    row4 = HBox([tlower_widget, tupper_widget, update_button],␣
↪layout=Layout(padding="5px"))

    ui = VBox([row1, row2, row3, row4, plot_output],␣
↪layout=Layout(padding="10px"))

    # Initialize the check logic
    on_modified_change(None)

    display(ui)

# API Input
if __name__ == "__main__":
    API_KEY = "BPFV62G5X9SUQB8HLPPFHNYFV"
    interactive_gdd_api_viewer(API_KEY)
```

```
VBox(children=(HBox(children=(Text(value='Ithaca', description='City:',␣
↪layout=Layout(width='200px')), Text(va…
```

Model 2: Drying calculation Algorithm

[5]:
```python
import urllib.parse
import urllib.request
```

```python
from io import BytesIO

import numpy as np
import pandas as pd
import plotly.express as px
import ipywidgets as widgets
from ipywidgets import VBox, HBox, Layout, Button, Output
from IPython.display import display

from datetime import datetime
from dateutil.relativedelta import relativedelta

# ============================================================
# 1) Function to Fetch Weather Data from Visual Crossing
# ============================================================
def fetch_drying_weather_data_from_api(api_key, city, state, start_date,␣
↪end_date,
                                unit_group="us"):
    location_string = f"{city}, {state}"
    encoded_location = urllib.parse.quote(location_string)

    base_url = "https://weather.visualcrossing.com/VisualCrossingWebServices/
↪rest/services/timeline"
    daily_url = (
        f"{base_url}/{encoded_location}/{start_date}/{end_date}"
        f"?
↪unitGroup={unit_group}&include=days&elements=datetime,temp,dew,soilmoisturevol01"
        f"&key={api_key}&contentType=csv"
    )

    try:
        response = urllib.request.urlopen(daily_url)
        df_days = pd.read_csv(BytesIO(response.read()))
    except urllib.error.HTTPError as e:
        print("HTTP Error:", e.code, e.read().decode())
    except urllib.error.URLError as e:
        print("URL Error:", e.reason)

    hourly_url = (
        f"{base_url}/{encoded_location}/{start_date}/{end_date}"
        f"?
↪unitGroup={unit_group}&include=hours&elements=datetime,solarradiation"
        f"&key={api_key}&contentType=csv"
    )

    try:
        response = urllib.request.urlopen(hourly_url)
```

```
        df_hours = pd.read_csv(BytesIO(response.read()))
    except urllib.error.HTTPError as e:
        print("HTTP Error:", e.code, e.read().decode())
    except urllib.error.URLError as e:
        print("URL Error:", e.reason)

    try:
        return df_days, df_hours
    except:
        return pd.DataFrame()  # Return empty if error

# ============================================================
# 2) Interactive UI to Get Weather Data
# ============================================================
def interactive_weather_data_viewer(api_key):
    city_widget = widgets.Text(value="San Joaquin Valley", description="City:",␣
 ↪layout=Layout(width="200px"))
    state_widget = widgets.Text(value="CA", description="State:",␣
 ↪layout=Layout(width="150px"))
    start_date_widget = widgets.Text(value="2024-10-25", description="Harvest␣
 ↪Date:", layout=Layout(width="180px"))
    #moisture_widget = widgets.FloatText(value=.80, description="Starting␣
 ↪Moisture:", layout=Layout(width="180px"))
    #end_date_widget = widgets.Text(value="2025-06-31", description="End:",␣
 ↪layout=Layout(width="180px"))
    update_button = widgets.Button(description="Estimate Drying",␣
 ↪button_style="primary", layout=Layout(width="170px"))
    plot_output = Output()

    def on_update_button_click(b):
        with plot_output:
            plot_output.clear_output(wait=True)
            city = city_widget.value
            state = state_widget.value
            sdate = start_date_widget.value

            sdate_dt = datetime.strptime(sdate, "%Y-%m-%d")
            edate_dt = sdate_dt + relativedelta(months=2)
            edate = edate_dt.strftime("%Y-%m-%d")

            d_df, h_df = fetch_drying_weather_data_from_api(
                api_key=api_key,
                city=city,
                state=state,
                start_date=sdate,
                end_date=edate
```

```
        )

        if d_df.empty or h_df.empty:
            print("No data returned from API or an error occurred.")
            return

        df = merge_dfs(daily_df=d_df, hourly_df=h_df)
        df = predict_moisture_content(df, sdate)

        df.rename(columns={"datetime": "date"}, inplace=True)
        df["date"] = pd.to_datetime(df["date"])
        location_str = f"{city}, {state}"

        # Quick plot of max temp as an example
        fig = px.line(df, x="date", y="predicted_moisture",␣
↪title=f"Expected Alfalfa Moisture Content for {location_str} - Harvest Date:␣
↪{sdate}")
        fig.update_layout(xaxis_title="Date", yaxis_title="Moisture Content␣
↪(%)")
        fig.show()

    update_button.on_click(on_update_button_click)

    # Layout
    row1 = HBox([start_date_widget, city_widget, state_widget],␣
↪layout=Layout(padding="5px"))
    #row2 = HBox([start_date_widget, end_date_widget],␣
↪layout=Layout(padding="5px"))
    row3 = HBox([update_button], layout=Layout(padding="5px"))
    ui = VBox([row1, row3, plot_output], layout=Layout(padding="10px"))

    display(ui)

# ==========================================================
# 3) Drying calculation Algorithm
# ==========================================================

def merge_dfs(daily_df, hourly_df):
    '''
    Extracts the peak solar radiation for each day from the
    hourly df and adds it to the daily df.
    '''
    d_df = daily_df
    h_df = hourly_df

    # Step 1: Convert datetime columns to datetime objects
    h_df['datetime'] = pd.to_datetime(h_df['datetime'])
```

```python
    d_df['datetime'] = pd.to_datetime(d_df['datetime'])

    # Step 2: Extract date from hourly timestamps
    h_df['date'] = h_df['datetime'].dt.date

    # Step 3: Group by date and get max solar radiation
    daily_peaks = h_df.groupby('date')['solarradiation'].max().reset_index()
    daily_peaks.rename(columns={'solarradiation': 'peak_solarradiation'},␣
↪inplace=True)

    # Step 4: Merge with daily dataframe
    # Convert daily_df datetime to just date for merging
    d_df['date'] = d_df['datetime'].dt.date
    merged_df = pd.merge(d_df, daily_peaks, on='date', how='left')

    # (Optional) Drop the 'date' column if not needed
    merged_df.drop(columns='date', inplace=True)

    return merged_df

def calculate_vapor_pressure_deficit(df):
    '''Takes weather df (with temp and dew in °F) and adds vapor pressure␣
↪deficit column in kPa'''

    # Convert temperature and dew point from Fahrenheit to Celsius
    df['temp_C'] = (df['temp'] - 32) / 1.8
    df['dew_C'] = (df['dew'] - 32) / 1.8

    # Calculate saturation and actual vapor pressure
    df['saturation_vapor_pressure'] = 0.6108 * np.exp((17.27 * df['temp_C']) /␣
↪(df['temp_C'] + 237.3))
    df['actual_vapor_pressure'] = 0.6108 * np.exp((17.27 * df['dew_C']) /␣
↪(df['dew_C'] + 237.3))

    # Calculate VPD
    df['vapor_pressure_deficit'] = df['saturation_vapor_pressure'] -␣
↪df['actual_vapor_pressure']

    return df

def swath_density_conversion(plants_per_sqft, g_per_plant=25):
    '''convert swath density from plants/ft^2 to g/m^2'''
    plants_per_sqm = plants_per_sqft * 10.764
    return plants_per_sqm * g_per_plant  # returns g/m²
```

```python
def predict_moisture_content(df, startdate, swath_density=25,␣
↪starting_moisture=0.80, application_rate=0):
    """
    Simulate daily drying and return DataFrame with moisture content␣
↪predictions.
    """
    # Ensure datetime and sort
    df['datetime'] = pd.to_datetime(df['datetime'])
    df = df[df['datetime'] >= pd.to_datetime(startdate)].copy()
    df.sort_values('datetime', inplace=True)

    # Calculate VPD
    df = calculate_vapor_pressure_deficit(df)

    # Initialize columns
    moisture_contents = [starting_moisture]
    drying_rates = []
    current_moisture = starting_moisture

    for idx, row in df.iterrows():
        day_number = len(moisture_contents) - 1
        DAY = 1 if day_number == 0 else 0

        SI = row['peak_solarradiation']
        VPD = row['vapor_pressure_deficit']
        SM = 100 * row['soilmoisturevol01']
        SD = 450 #swath_density_conversion(swath_density)
        AR = 0 #application_rate

        k = calculate_drying_rate_constant(SI, VPD, DAY, SM, SD, AR)

        # Update moisture content
        current_moisture *= np.exp(-k)
        moisture_contents.append(current_moisture)
        drying_rates.append(k)

        # Stop if moisture is below 0.5
        if current_moisture <= 0.08:
            break


    # Create result DataFrame
    result_df = df.iloc[:len(moisture_contents)-1].copy()
    result_df['drying_rates'] = drying_rates
    result_df['predicted_moisture'] = moisture_contents[:-1]

    result_df = result_df.dropna(subset=['predicted_moisture'])
```

```
    return result_df

def calculate_drying_rate_constant(SI, VPD, DAY, SM, SD, AR=0):
  '''
  SI = solar insolation, W/m^2
  VPD = vapor pressure deficit, kPA
  DAY = 1 for first day, 0 otherwise
  SM = soil moisture content, % dry basis
  SD = swath density, g/m^2
  AR = application rate of chemical solution, g_solution/g_dry-matter
  '''

  drying_rate = ((SI * (1. + 9.03*AR)) + (43.8 * VPD)) / ((61.4 * SM) + SD * (1.
↪82 - 0.83 * DAY) * ((1.68 + 24.8 * AR)) + 2767)

  return drying_rate

# API Key
if __name__ == "__main__":
    API_KEY = "BPFV62G5X9SUQB8HLPPFHNYFV"
    interactive_weather_data_viewer(API_KEY)
```

```
VBox(children=(HBox(children=(Text(value='2024-10-25', description='Harvest Date:
↪', layout=Layout(width='180px…
```

## 0.1 Scratch work and testing

[6]:
```
d_df, h_df = fetch_drying_weather_data_from_api("BPFV62G5X9SUQB8HLPPFHNYFV",␣
↪"San Joaquin Valley", "CA", "2024-06-01", "2024-08-01",unit_group="us")
merged_df = merge_dfs(d_df, h_df)
df2 = predict_moisture_content(merged_df, "2024-06-01")
df2
```

[6]:

| | datetime | temp | dew | soilmoisturevol01 | peak_solarradiation | temp_C | \ |
|---|---|---|---|---|---|---|---|
| 0 | 2024-06-01 | 63.4 | 56.0 | 0.153 | 413.0 | 17.444444 | |
| 1 | 2024-06-02 | 63.9 | 55.8 | 0.150 | 589.0 | 17.722222 | |
| 2 | 2024-06-03 | 65.2 | 55.7 | 0.148 | 805.0 | 18.444444 | |
| 3 | 2024-06-04 | 65.9 | 56.8 | 0.148 | 850.0 | 18.833333 | |
| 4 | 2024-06-05 | 64.8 | 58.2 | 0.148 | 517.0 | 18.222222 | |
| 5 | 2024-06-06 | 64.8 | 59.5 | 0.145 | 381.0 | 18.222222 | |
| 6 | 2024-06-07 | 64.7 | 58.9 | 0.143 | 335.0 | 18.166667 | |
| 7 | 2024-06-08 | 65.5 | 57.5 | 0.144 | 619.0 | 18.611111 | |
| 8 | 2024-06-09 | 65.3 | 56.9 | 0.143 | 896.0 | 18.500000 | |
| 9 | 2024-06-10 | 67.0 | 57.7 | 0.141 | 816.0 | 19.444444 | |
| 10 | 2024-06-11 | 65.9 | 59.3 | 0.139 | 304.0 | 18.833333 | |
| 11 | 2024-06-12 | 66.2 | 58.7 | 0.140 | 907.0 | 19.000000 | |

```
12 2024-06-13  66.1  57.1            0.139            879.0  18.944444
13 2024-06-14  67.1  57.7            0.137            894.0  19.500000
14 2024-06-15  66.9  59.7            0.136            869.0  19.388889
15 2024-06-16  67.4  58.2            0.136            864.0  19.666667
16 2024-06-17  65.7  55.7            0.136            819.0  18.722222

        dew_C  saturation_vapor_pressure  actual_vapor_pressure  \
0   13.333333                   1.992984               1.530743
1   13.222222                   2.028215               1.519682
2   13.166667                   2.122378               1.514178
3   13.777778                   2.174649               1.575697
4   14.555556                   2.093005               1.657158
5   15.277778                   2.093005               1.736084
6   14.944444                   2.085718               1.699256
7   14.166667                   2.144644               1.615977
8   13.833333                   2.129777               1.581397
9   14.277778                   2.259066               1.627650
10  15.166667                   2.174649               1.723731
11  14.833333                   2.197393               1.687134
12  13.944444                   2.189789               1.592851
13  14.277778                   2.266880               1.627650
14  15.388889                   2.251275               1.748514
15  14.555556                   2.290466               1.657158
16  13.166667                   2.159601               1.514178

    vapor_pressure_deficit  drying_rates  predicted_moisture
0                 0.462241      0.097252            0.800000
1                 0.508532      0.120712            0.725862
2                 0.608200      0.164628            0.643323
3                 0.598952      0.173455            0.545673
4                 0.435847      0.106122            0.458777
5                 0.356921      0.078803            0.412585
6                 0.386462      0.070092            0.381320
7                 0.528667      0.127739            0.355508
8                 0.548381      0.183236            0.312877
9                 0.631415      0.168439            0.260492
10                0.450918      0.064797            0.220111
11                0.510260      0.185776            0.206301
12                0.596938      0.181160            0.171325
13                0.639230      0.184988            0.142936
14                0.502760      0.178993            0.118796
15                0.633308      0.179137            0.099327
16                0.645422      0.170204            0.083037
```

[7]:

```python
def swath_density_conversion(plants_per_sqft, g_per_plant=5):
    '''convert swath density from plants/ft^2 to g/m^2'''
    plants_per_sqm = plants_per_sqft * 10.764
```

```
    return plants_per_sqm * g_per_plant  # returns g/m²

out = swath_density_conversion(25)
print(out)
```

```
1345.4999999999998
```

[8]:
```
DAY = 1
SI = 700 #row['peak_solarradiation']
VPD = 2.5 #row['vapor_pressure_deficit']
SM = 17 #row['soilmoisturevol01']
SD = 450 #swath_density
AR = 0 #application_rate

drying_rate = ((SI * (1. + 9.03*AR)) + (43.8 * VPD)) / ((61.4 * SM) + SD * (1.
↪82 - 0.83 * DAY) * ((1.68 + 24.8 * AR)) + 2767)

print(drying_rate)
```

```
0.17755152174485223
```