The African School of Fundamental Physics and Applications

Integrating Scientific Computing into Math and Science Classes

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 04
Differential Equations

Brookhaven National Laboratory

Session **04** – Topics

- Use **SciPy** to simulate numerical solutions to ordinary differential equations (**ODE**s) representing physical laws
- Model the radioactive decay of **Fluorine-18**
- Model a **damped oscillator** to appreciate the impact of **critical damping**
- Simulate the **Dynamical Kinematics** of a **Model Rocket** through lift-off, where the system mass *decreases* as solid motor fuel is burnt to produce thrust
- Model the **electrostatic field** around the **charged parallel plates** inside a capacitor where the surrounding *walls* are <u>conductors</u> (fixed potential) or <u>insulators</u> (fixed charge)

Solve ODEs with SciPy

- We often must numerically solve systems of **differential equations** so we can simulate dynamic models
- The Python package **SciPy** contains several ready-to-use numerical methods to estimate the solution to a variety of differential equations
  - Ordinary and Partial Differential Equations (ODE/PDE)
  - Linear and Non-Linear Differential Equations
  - **Initial Value Problem (IVP)** and **Boundary Value Problems (BVP)**
  - Both Individual and Systems of <u>Linked</u> Differential Equations

Scientific Computing with Python

https://scipy.org

SciPy

Fundamental algorithms for scientific computing in Python

GET STARTED

FUNDAMENTAL ALGORITHMS

SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems.

BROADLY APPLICABLE

The algorithms and data structures provided by SciPy are broadly applicable across domains.

FOUNDATIONAL

Extends NumPy providing additional tools for array computing and provides specialized data structures, such as sparse matrices and k-dimensional trees.

PERFORMANT

SciPy wraps highly-optimized implementations written in low-level languages like Fortran, C, and C++. Enjoy the flexibility of Python with the speed of compiled code.

EASY TO USE

SciPy's high level syntax makes it accessible and productive for programmers from any background or experience level.

OPEN SOURCE

Distributed under a liberal BSD license, SciPy is developed and maintained publicly on GitHub by a vibrant, responsive, and diverse community.

Modelling Nuclear Decay

$N(t) \equiv$ number of nuclei at time t

$\tau \equiv$ mean lifetime (half life)

$$\frac{dN}{dt} = \boxed{-\frac{N(t)}{\tau}}$$

$$\frac{dN}{dt} = \frac{N(t+\Delta t) - N(t)}{\Delta t}$$

Fermat's Difference Quotient

$$\boxed{-\frac{N(t)}{\tau}} = \frac{N(t+\Delta t) - N(t)}{\Delta t}$$

$$N(t+\Delta t) = N(t) - \frac{N(t)}{\tau}\Delta t$$

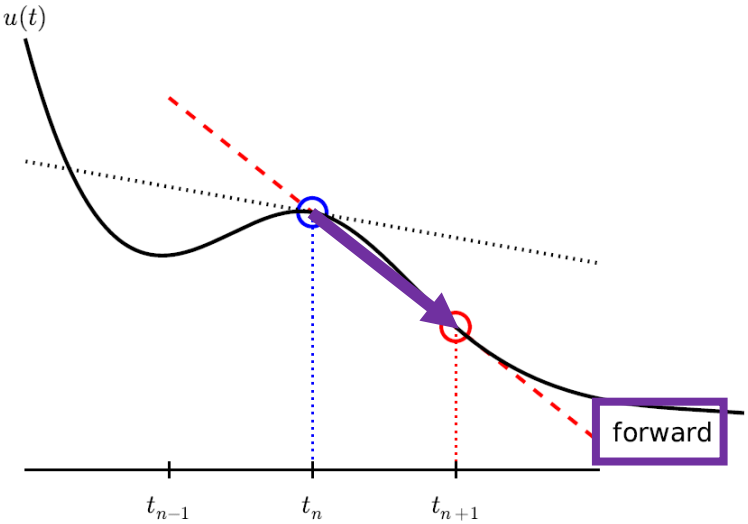
This is Euler's Method

Modelling Nuclear Decay

$N(t) \equiv$ number of nuclei at time t

$\tau \equiv$ mean lifetime (half life)

Leonhard Euler
(1707 – 1783)



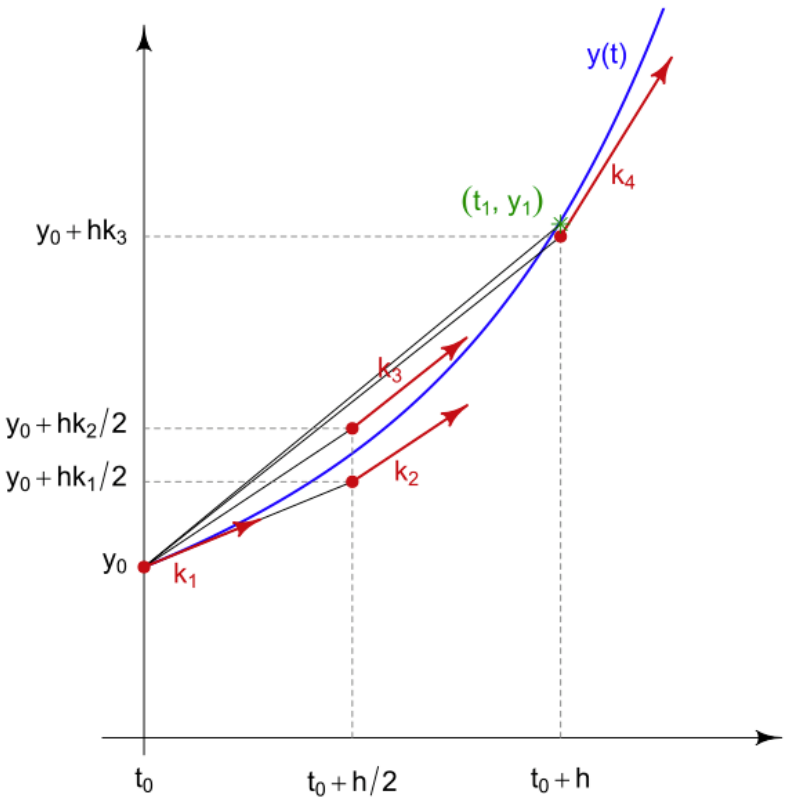
$$N(t + \Delta t) = N(t) - \frac{N(t)}{\tau}\Delta t$$

This is Euler's Method

Runge-Kutta Methods

- Around 1900, two German mathematicians, **Carl Runge** and **Wilhelm Kutta**, wanted to improve the accuracy of **Euler's Method**
- Following the same motivation underlying **Simpson's Rule** for numerical integration, they developed a method of interpolating a curve between the endpoints of a sampled interval
- Using a **weighted combination of tangent lines** sampled throughout an interval, a more accurate derivative can be calculated than using the *single* tangent line in **Euler's Method**

Runge-Kutta Methods



$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f\left(t_n + h, y_n + hk_3\right).
\end{aligned}$$

$$\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right), \\
t_{n+1} &= t_n + h
\end{aligned}$$

Scientific Computing with Python

Ordinary differential equations (**solve_ivp**)

Integrating a set of ordinary differential equations (ODEs) given initial conditions is another useful example. The function **solve_ivp** is available in SciPy for integrating a first-order vector differential equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right),$$

given initial conditions $\mathbf{y}\left(0\right) = y_0$, where $\mathbf{y}$ is a length N vector and $\mathbf{f}$ is a mapping from $\mathcal{R}^N$ to $\mathcal{R}^N$. A higher-order ordinary differential equation can always be reduced to a differential equation of this type by introducing intermediate derivatives into the $\mathbf{y}$ vector.

method : ***string or*** `OdeSolver`***, optional***

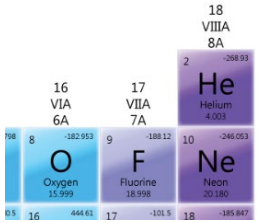
Integration method to use:

Runge-Kutta-Fehlberg (RKF45)

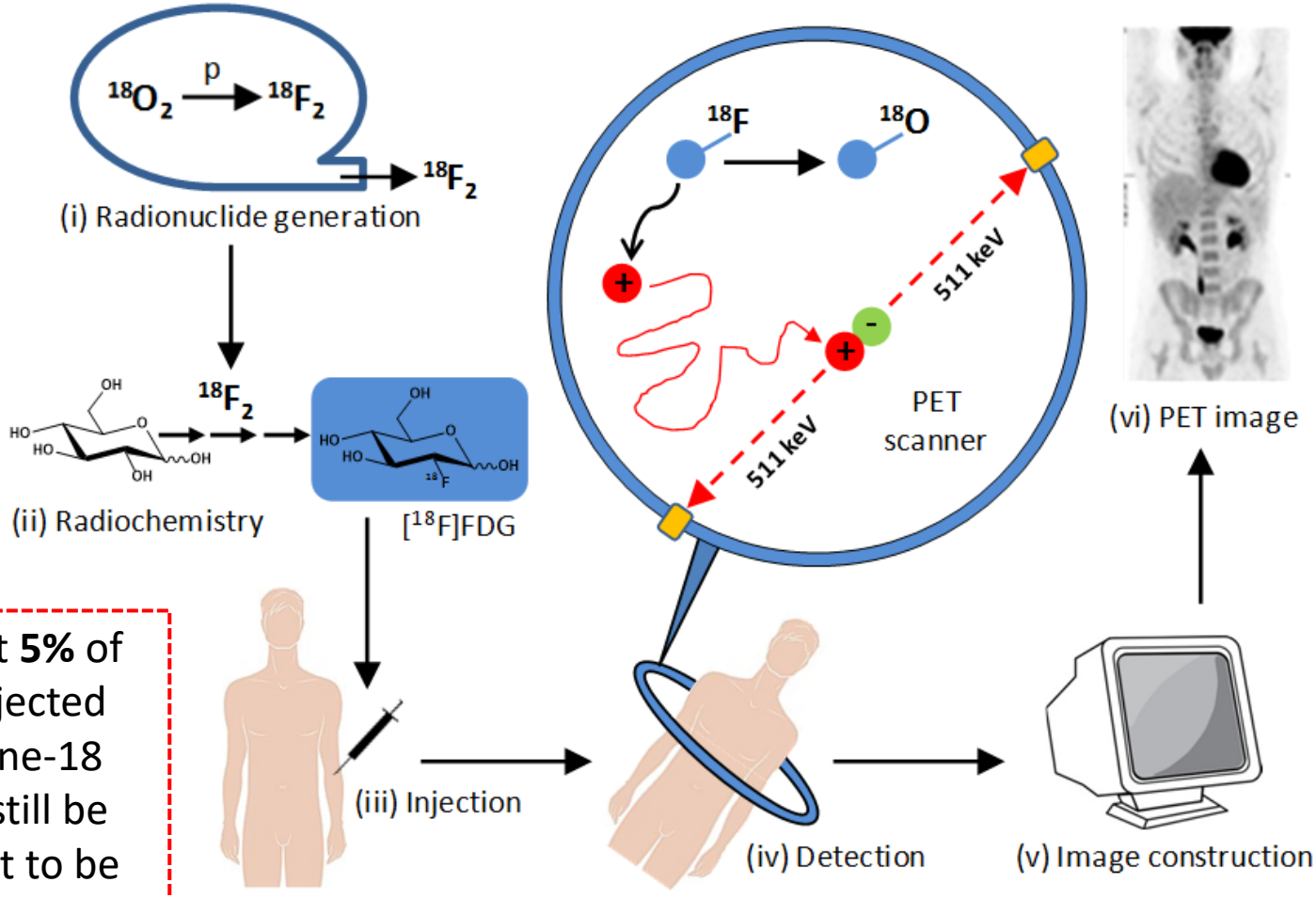
- 'RK45' (default): Explicit Runge-Kutta method of order 5(4) [1]. The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula (local extrapolation is done). A quartic interpolation polynomial is used for the dense output [2]. Can be applied in the complex domain.

First, convert this ODE into standard form by setting $\mathbf{y} = \left[\frac{dw}{dz}, w\right]$ and $t = z$. Thus, the differential equation becomes

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} ty_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \mathbf{y}.$$



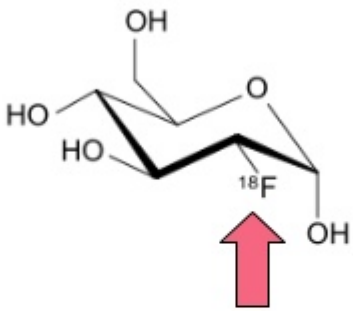
Fluorine-18

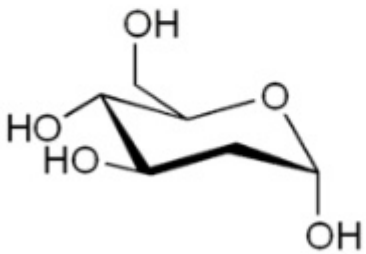


At least **5%** of the injected Flourine-18 must still be present to be detected

Fluorine-18

Example: FDG





2-Deoxy-D-Glucose (2DG)

- Fluorodeoxyglucose is a radiopharmaceutical is a glucose analog with the radioactive isotope Fluorine-18 in place of OH
- ^{18}F has a half life of 110 minutes
- FDG is taken up by high glucose using cells such as brain, kidney, and cancer cells.
- Once absorbed, it undergoes a biochemical reaction whose products cannot be further metabolized, and are retained in cells.
- After decay, the ^{18}F atom becomes a harmless non-radioactive heavy oxygen $^{18}O^-$ that joins up with a hydrogen atom, and forms glucose phosphate that is eliminated via carbon dioxide and water

Run fluroine18_decay.ipynb – **Cells 1...2**

Import needed packages/modules

```
[1]  # Cell 1
     from pathlib import Path

     import matplotlib.pyplot as plt
     import numpy as np
     from scipy.integrate import solve_ivp

     import pandas as pd
```

①

Define the model function containing the differential equation

$$\frac{dN}{dt} = -\frac{N(t)}{\tau}$$

②

```
[2]  # Cell 2
     def model(time, state_vector, tau):
         # Unpack current state vector
         n = state_vector
         # Express differential equation
         d_n = -n / tau
         return d_n
```

③ ④ ⑤ ⑥

Run fluroine18_decay.ipynb – **Cell 3**

Set simulation parameters and initial conditions

1. We will simulate the decay over a 12 hour duration
2. The half-life of F^{18} is 110 minutes ①
3. We will start at 100% concentration of nuclei

```
[3]  # Cell 3
     final_time = 12 # hours
     half_life = 110 / 60 # hours
     initial_concentration = 100 # molecules/L
```

②

Run fluroine18_decay.ipynb – **Cell 4**

Use scipy's `solve_ivp()` to numerically estimate the ODE using the RKF45 Method ①

1. We will limit the solver to a maximum time step of 0.01 hour
2. The actual time values used will be returned by the solver
3. The solver will also return the nuclei concentration at each time value

```
# Cell 4
sol = solve_ivp(                                  ②
    model,                                        ③
    (0, final_time),  # tuple of time span        ④
    [initial_concentration],  # initial state vector  ⑤
    max_step=0.01,                                ⑥
    args=(half_life,),  # tuple of constants used in ODE  ⑦
)

# Retrieve results of the solution
time_steps = sol.t                                ⑧
nuclei, = sol.y

# Display the first 10 time and concentration values
pd.DataFrame({
    'Time (hrs)': time_steps[:10],                ⑨
    '% Concentration': nuclei[:10]
})
```

Run fluroine18_decay.ipynb – **Cell 4**

Use scipy's `solve_ivp()` to numerically estimate the ODE using the RKF45 Method ①

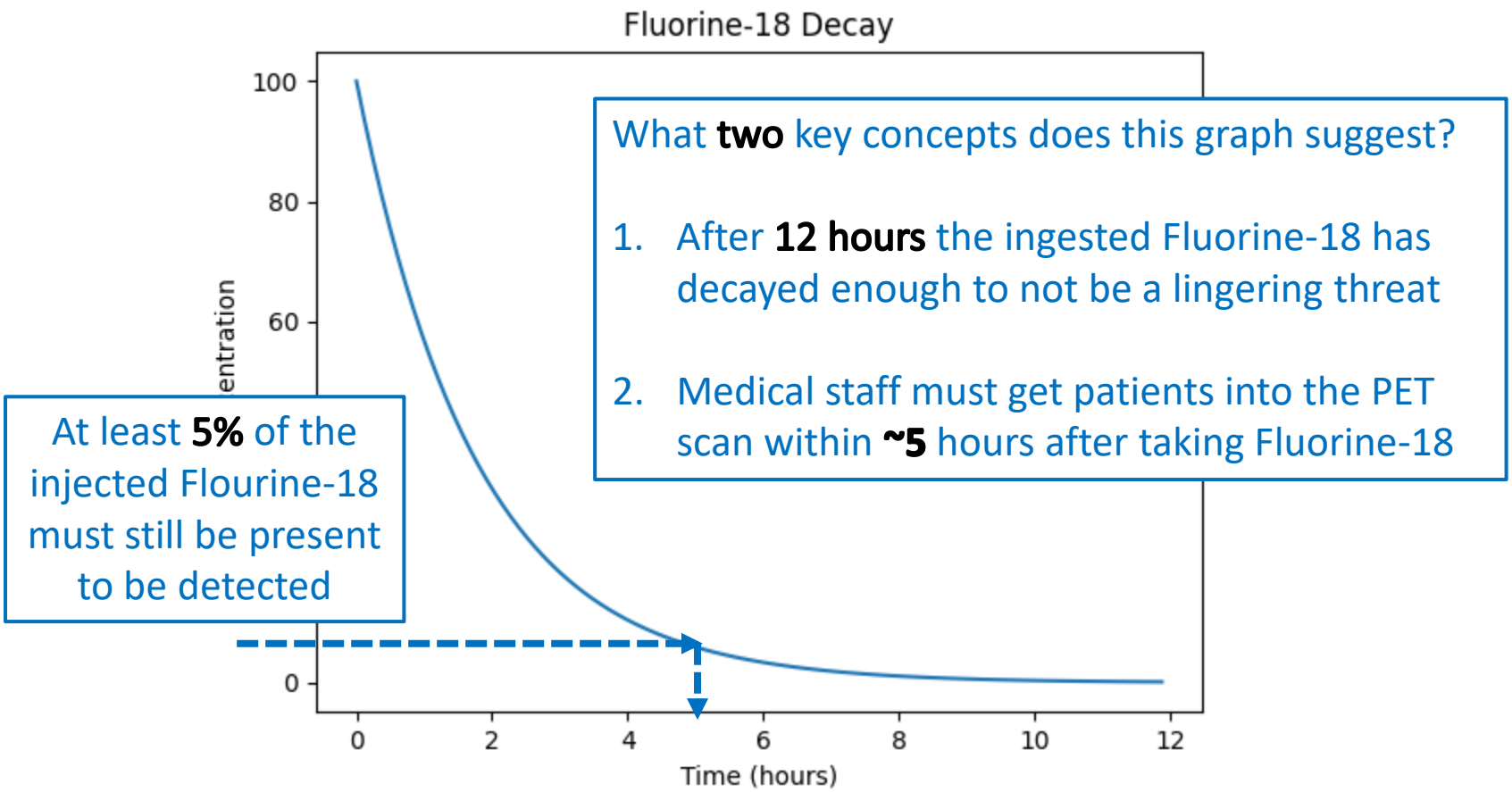
1. We will limit the solver to a maximum time step of 0.01 hour
2. The actual time values used will be returned by the solver
3. The solver will also return the nuclei concentration at each time value

```
# Cell 4
sol = solve_ivp(
    model,
    (0, final_time),  # tuple of time span
    [initial_concentration],  # initial state vector
    max_step=0.01,
    args=(half_life,),  # tuple of constants used in ODE
)

# Retrieve results of the solution
time_steps = sol.t
nuclei, = sol.y

# Display the first 10 time and concentration values
pd.DataFrame({
    'Time (hrs)': time_steps[:10],
    '% Concentration': nuclei[:10]
})
```

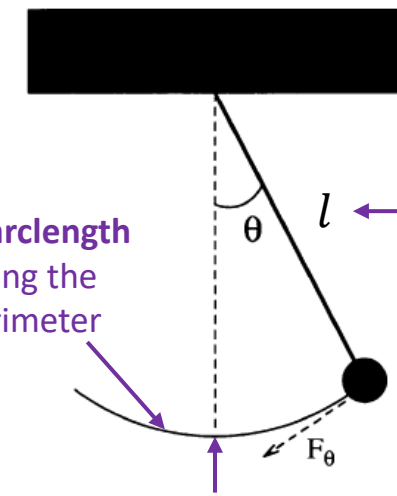
② ③ ④ ⑥ ⑧ ⑨

	Time (hrs)	% Concentration
0	0.00	100.000000
1	0.01	99.456030
2	0.02	98.915020
3	0.03	98.376952
4	0.04	97.841811
5	0.05	97.309582
6	0.06	96.780247
7	0.07	96.253792
8	0.08	95.730200
9	0.09	95.209457

Run fluroine18_decay.ipynb – **Cell 5**



Modelling a Simple Pendulum



s = **arclength** along the perimeter

l = **length** of the rod (constant)

$s = 0$ when $\theta = 0$

$$s = l\theta$$

$$F = ma$$

Take 2nd derivative of both sides with respect to time

$$F = m\boxed{\frac{d^2s}{dt^2}}$$

$$\boxed{\frac{d^2s}{dt^2}} = l\frac{d^2\theta}{dt^2}$$

$$\boxed{F_\theta} = ml\frac{d^2\theta}{dt^2}$$

$$\boxed{F_\theta} = -mg\sin\theta$$

Gravity is a restoring force

$$\cancel{m}l\frac{d^2\theta}{dt^2} = -\cancel{m}g\sin\theta$$

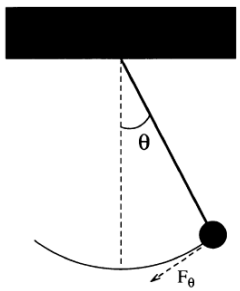
$$\frac{d^2\theta}{dt^2} = -\boxed{\frac{g}{l}}\sin\theta$$

$\boxed{\frac{g}{l} = \text{Phase Constant}}$

$$\boxed{\begin{aligned}\frac{d\omega}{dt} &= -\frac{g}{l}\sin\theta \\ \frac{d\theta}{dt} &= \omega\end{aligned}}$$

We can break the 2nd order ODE into <u>two</u> *linked* first-order ODEs and use **RKF45** on each

Damped Harmonic Oscillator



$$\frac{d^2\theta}{dt^2} = -\boxed{\frac{g}{l}}\sin\theta$$

$\frac{g}{l}$ = The **phase constant**

We can introduce a new **resistive force term** into the equation of motion for a pendulum

$$\frac{d^2\theta}{dt^2} = -q\frac{d\theta}{dt} - \frac{g}{l}\sin\theta$$

q = The **damping constant**

$$\boxed{\begin{aligned} \frac{d\omega}{dt} &= -q\omega - \frac{g}{l}\sin\theta \\ \frac{d\theta}{dt} &= \omega \end{aligned}}$$

Damped Harmonic Oscillator

- **Underdamped** – The system swings back and forth (oscillates) over its equilibrium point, but ultimately comes to rest

Example: $q = 1$

- **Overdamped** – The system *never* oscillates, but it takes a while for it to return to its equilibrium point

Example: $q = \frac{(phase\ constant)^2}{2}$

- **Critically damped** – The system *never* oscillates <u>and</u> returns to its equilibrium point **in the least amount of time**

$$q = \frac{(phase\ constant)^2}{4}$$

Note: There is only <u>one</u> value for q where the system is *critically* damped

Run damped_pendulum.ipynb – **Cells 1...2**

Import needed packages/modules

```
[1]  # Cell 1
     from pathlib import Path

     import matplotlib.pyplot
     import numpy as np
     from matplotlib.ticker i
     from scipy.integrate impo

     import pandas as pd
```

①

Define the model function containing the differential equation:

$$\frac{d^2\theta}{dt^2} = -q\frac{d\theta}{dt} - \frac{g}{l}\sin\theta$$

②

Using linked first order differential equations:

1. $\frac{d\omega}{dt} = -q\omega - \frac{g}{l}\sin\theta$
2. $\frac{d\theta}{dt} = \omega$

③

```
[2]  # Cell 2
     def model(time, state_vector, phase_constant, damping_constant):
         # Unpack dependent variables from current state vector
         omega, theta = state_vector
         # Express differential equation
         d_omega = -damping_constant * omega - phase_constant * np.sin(theta)
         d_theta = omega
         return d_omega, d_theta
```

④ ⑤ ⑥ ⑦

Run damped_pendulum.ipynb – **Cell 3**

Set simulation parameters and initial conditions ①

1. The pendulum maintains a constant length of 1.0 meter
2. The acceleration due to gravity $g = 9.81 \frac{m}{s^2}$
3. The simulation will last **15** seconds ②
4. The initial anglular displacement $\theta = 75^\circ$
5. The pendulum will be released from *rest* such that $\omega_0 = 0.0 \ \frac{rad}{s}$
6. The three **damping constants** will be set to their critical values ③

```
[3]  # Cell 3
     pendulum_length = 1.0  # meters
     phase_constant = 9.81 / pendulum_length
     time_final = 15.0 # seconds
     theta_initial = np.radians(75)  # 75 degrees
     omega_initial = 0.0 # rad/s

     # Set the damping constants
     underdamped_constant = 1.0
     overdamped_constant = pow(phase_constant, 2) / 2.0
     critically_damped_constant = pow(phase_constant, 2) / 4.0
```

④ ⑤ ⑥ ⑦

Run damped_pendulum.ipynb – **Cell 4**

Use scipy's `solve_ivp()` to numerically estimate the ODE using the RKF45 Method

1. We will limit the solver to a *maximum* time step of 0.01 second
2. The actual time values used will be returned by the solver
3. The solver will return the angular velocity (ω) at each time value
4. The solver will return the angular displacement (θ) at each time value

```
[4]  # Cell 4
     sol = solve_ivp(
         model,
         (0, time_final),
         [omega_initial, theta_initial],
         max_step=0.01,
         args=(phase_constant, underdamped_constant),
     )
     time_steps = sol.t
     theta_underdamped = sol.y[1]
```

①

```
sol = solve_ivp(
    model,
    (0, time_final),
    [omega_initial, theta_initial],
    max_step=0.01,
    args=(phase_constant, overdamped_constant),
)
theta_overdamped = sol.y[1]
```

②

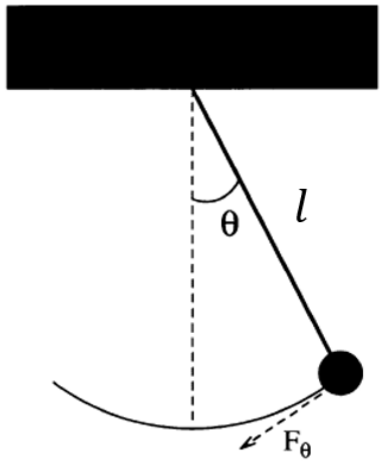
```
sol = solve_ivp(
    model,
    (0, time_final),
    [omega_initial, theta_initial],
    max_step=0.01,
    args=(phase_constant, critically_damped_constant),
)
theta_critically_damped = sol.y[1]
```

③

```
# Display the first 10 time and displacement values
pd.DataFrame({
    'Time (s)': time_steps[:10],
    'Under': theta_underdamped[:10],
    'Over': theta_overdamped[:10],
    'Crit': theta_critically_damped[:10],
})
```

④

Check damped_pendulum.ipynb – **Cell 4**



	Time (s)	Under	Over	Crit
0	0.000000	1.308997	1.308997	1.308997
1	0.000105	1.308997	1.308997	1.308997
2	0.001160	1.308991	1.308991	1.308991
3	0.011160	1.308409	1.308500	1.308456
4	0.021160	1.306891	1.307444	1.307194
5	0.031160	1.304445	1.306040	1.305360
6	0.041160	1.301082	1.304421	1.303077
7	0.051160	1.296812	1.302669	1.300441
8	0.061160	1.291644	1.300835	1.297529
9	0.071160	1.285590	1.298951	1.294400

$$\frac{d^2\theta}{dt^2} = -q\frac{d\theta}{dt} - \frac{g}{l}\sin\theta$$

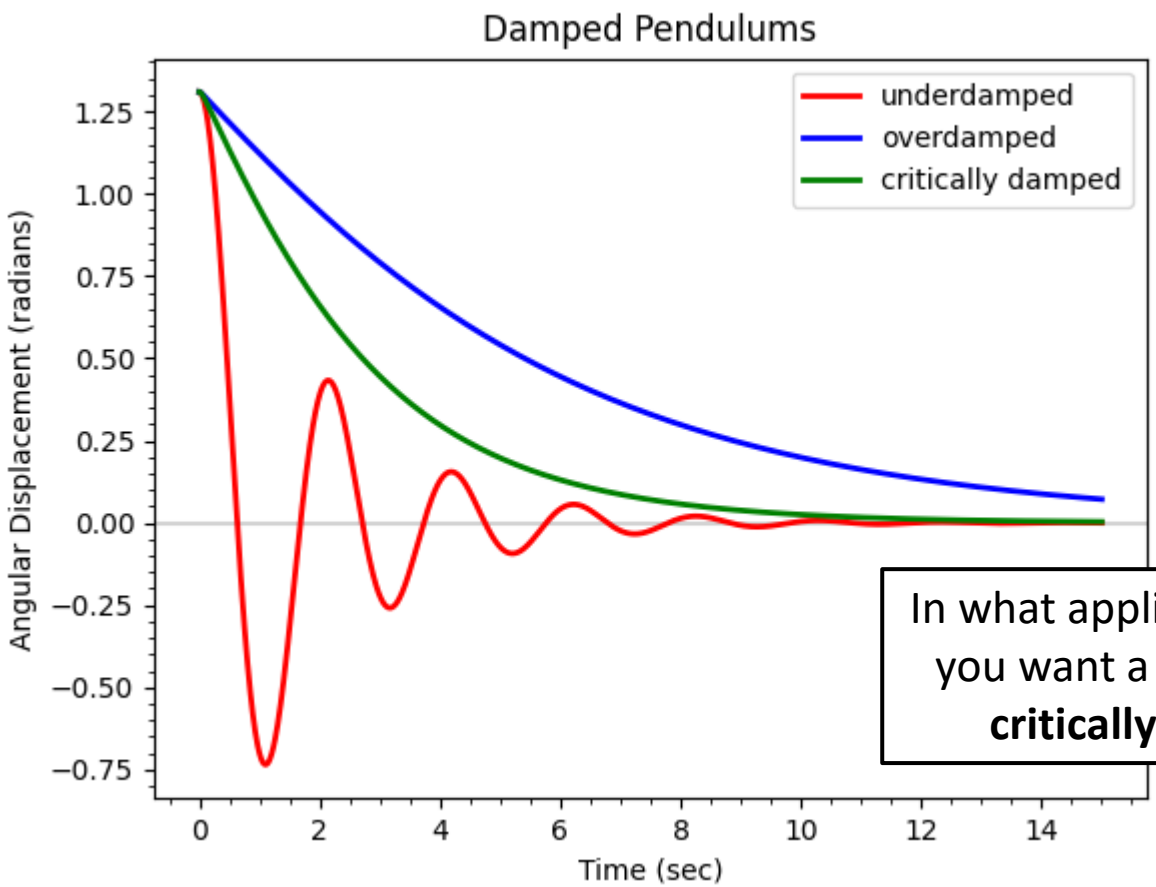
Run damped_pendulum.ipynb – **Cell 5**

Plot the angular displacment over time for each type of pendulum

```
[5]  # Cell 5
     plt.figure(figsize=(8, 6))
     plt.plot(time_steps, theta_underdamped,
         label="underdamped", c="r", lw=2, zorder=3)
     plt.plot(time_steps, theta_overdamped,
         label="overdamped", c="b", lw=2, zorder=3)
     plt.plot(time_steps, theta_critically_damped,
         label="critically damped", c="g", lw=2, zorder=3)
     plt.title("Damped Pendulums")
     plt.xlabel("Time (sec)")
     plt.ylabel("Angular Displacement (radians)")
     plt.axhline(y=0.0, color="lightgray")
     ax = plt.gca()
     ax.xaxis.set_minor_locator(AutoMinorLocator())
     ax.yaxis.set_minor_locator(AutoMinorLocator())
     ax.legend(loc="upper right")
     plt.show()
```

① ② ③ ④

Check damped_pendulum.ipynb – **Cell 5**



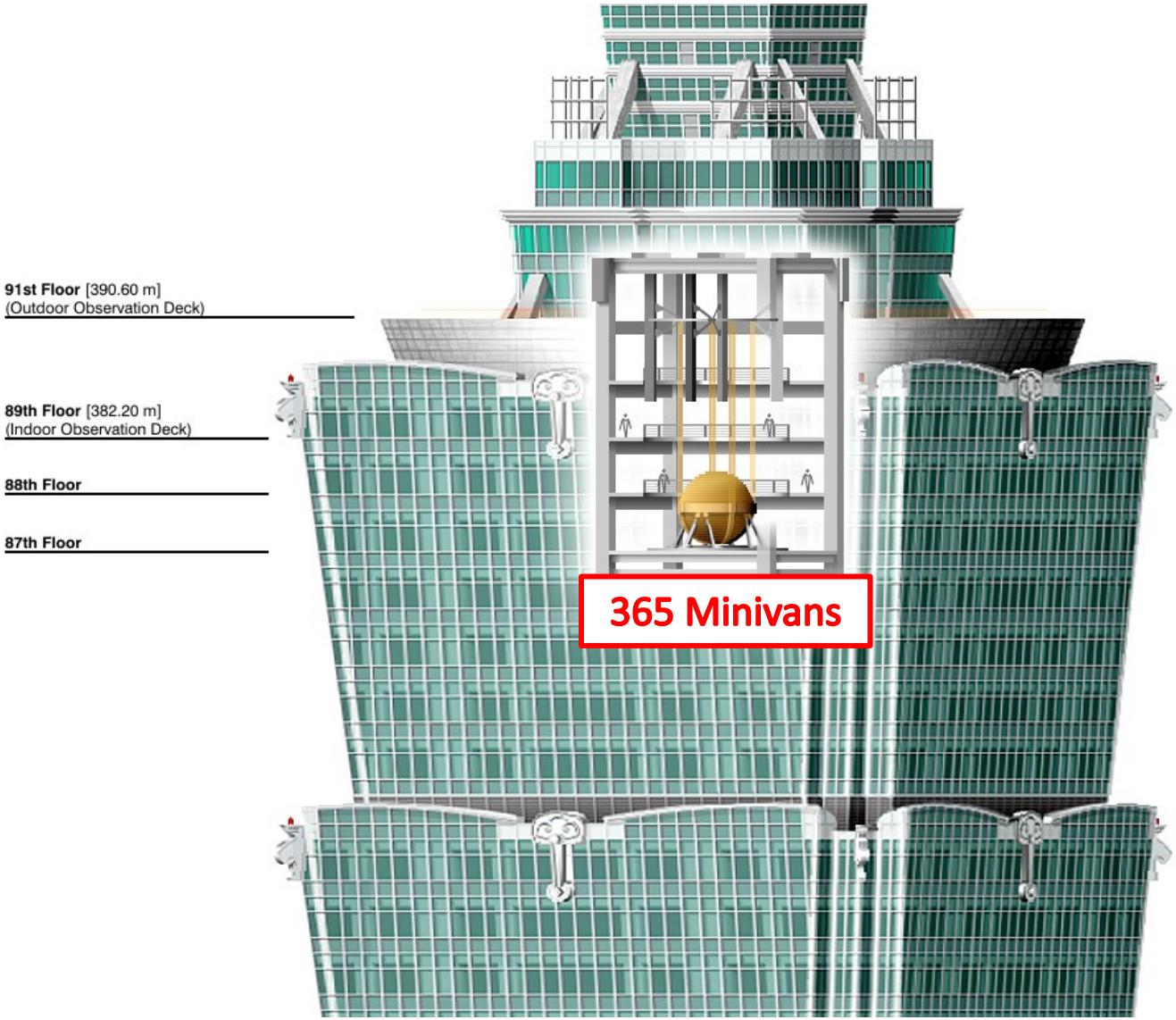
In what applications would you want a system to be **critically damped?**

Damped Harmonic Oscillator

Tuned Mass Dampers

Taipei 101 – Taiwan
The Worlds Tallest Building
(2004-2009)

Tuned Mass Dampers



Model Rocket

- This Model Rocket has <u>two</u> stages, with **each** stage containing its *own* F15 rocket motor
- After the 1st (booster) stage is ignited, the rocket will blast off and begin to rise using just the booster motor
- After the 1st stage motor burns out, the booster stage (with its spent motor) is ejected, the 2nd stage F15 motor ignites, and the rocket continues to ascend
- The 2nd stage and its motor remain part of the rocket throughout the duration of the flight

Model Rocket

- The Stage 1 booster has a mass of 0.0519 kg (not including its F15 motor)
- The Stage 2 main rocket has a mass of 0.1729 kg (not including its F15 motor)
- An F15 model rocket motor has a mass of 0.101 kg, generates a thrust of 17.2 Newtons, and burns for 1.6 seconds
- For this simulation, we will assume the thrust of a motor is constant throughout its burn, assume there is no delay between the ejection of the booster and the ignition of the Stage 2 motor, and ignore air friction

Model Rocket

- We must model the trajectory of the rocket over the first **four** seconds of its flight: from liftoff to booster separation and into its unpowered glide phase
- We want to calculate the rocket's mean velocity (mph) and altitude (feet) over time (seconds) and determine its average speed and maximum height through **four** seconds
- We will assume the mass of each motor reduces linearly throughout its burn (100% of mass at ignition and 0% of mass at moment of burnout)
- The force of gravity resisting the rocket is:

$G = Gravitational\ Constant$

$M_E = Mass\ of\ Earth$

$R = Radius\ of\ Earth$

$$F_g = \frac{G \times M_E \times m}{(R+h)^2}$$

$m = Mass\ of\ rocket$

$h = Height\ of\ rocket$

Run model_rocket.ipynb – **Cells 1...2**

Import needed packages/modules

```
[1]  # Cell 1
     from pathlib import Path

     import matplotlib.pyplot
     import numpy as np
     from scipy.integrate impo

     import pandas as pd
```

①

Set simulation parameters and initial conditions

1. The simulation will last **4** seconds
2. The rocket will start at height = 0 meters and launch from rest $v_0 = 0.0$
3. The mass of each stage of the rocket does not include the mass of its motor

```
[2]  # Cell 2
     tf = 4.0  # Simulation time (s)

     G = 6.67430e-11  # Gravitational constant (m^3/kg/s^2)
     M = 5.972e24  # Mass of the Earth (kg)
     R = 6.371e6  # Radius of the Earth (m)

     # For this model rocket
     STAGE1_MASS = 0.0519  # kg
     STAGE2_MASS = 0.1729  # kg

     # For an F15 model rocket engine
     ENGINE_MASS = 0.101  # kg
     ENGINE_THRUST = 17.2  # Newtons
     ENGINE_BURNOUT = 1.6  # seconds
```

②

③

④

⑤

Run model_rocket.ipynb – **Cell 3**

Define functions to return the instantaneous thurst and mass of the rocket ①

1. The variable t is the elapsed time (in seconds) since liftoff
2. The thrust of a motor is constant throughout its burn
3. The mass of eac
4. At motor burnout
5. There is no delay
6. Air friction is not

```
[3] # Cell 3
    def thrust_func(t):                                          ②
        # returns thrust in Newtons
        if t < ENGINE_BURNOUT * 2:
            return ENGINE_THRUST                                 ③
        return 0                                                 ④

    def rocket_mass_func(t):                                     ⑤
        # returns weight in kilograms
        if t <= ENGINE_BURNOUT:                                  ⑥
            motor_mass = ENGINE_MASS * (ENGINE_BURNOUT - t) / ENGINE_BURNOUT
            return STAGE1_MASS + motor_mass + STAGE2_MASS + ENGINE_MASS
        if t <= ENGINE_BURNOUT * 2:                              ⑦
            motor_mass = ENGINE_MASS * (ENGINE_BURNOUT * 2 - t) / ENGINE_BURNOUT
            return STAGE2_MASS + motor_mass
        return STAGE2_MASS                                       ⑧

    print(f"Liftoff Thrust = {thrust_func(0):.2f} N")            ⑨
    print(f"Liftoff Mass = {rocket_mass_func(0):.2f} kg")

    Liftoff Thrust = 17.20 N                                     ⑩
    Liftoff Mass = 0.43 kg
```

Run model_rocket.ipynb – **Cell 4**

Define the model function containing the differential equation:

$$\frac{d^2s}{dt^2} = \frac{(F_{\text{thrust}} - F_g)}{m}$$ ①

m (mass of rocket in kg) = depends on elapsed time

F_{thrust} (in Newtons) = depends on elapsed time

$F_g = \frac{G \times M_E \times m}{(R_E + h)^2}$ where height h (in meters) depends on elapsed time

Using linked first order differential equations:

1. $\frac{dv}{dt} = \frac{(F_{\text{thrust}} - F_g)}{m}$ ②
2. $\frac{ds}{dt} = v$ ③

```
[4]  # Cell 4
     def model(t, state_vector):                          ④
         v, h = state_vector # h (height) = distance      ⑤
         m = rocket_mass_func(t)                          ⑥
         F_thrust = thrust_func(t)                        ⑦
         F_gravity = G * M * m / (R + h) ** 2
         d_v = (F_thrust - F_gravity) / m                 ⑧
         d_h = v
         return d_v, d_h                                  ⑨
```

Run model_rocket.ipynb – **Cell 5**

Use scipy's `solve_ivp()` to numerically estimate the ODE using the RKF45 Method ①

1. We will limit the solver to a *maximum* time step of 0.01 second
2. The actual time values used will be returned by the solver
3. The solver will return the velocity and height each time value

```
[5]  # Cell 5
     sol = solve_ivp(model, (0, tf), [0, 0], max_step=0.01)
     t = sol.t
     v, h = sol.y

     v *= 2.23  # Convert m/s to mph
     h *= 3.28  # Convert m to feet

     # Display the first 10 time and displacement values
     pd.DataFrame({
         'Time (s)': t[:10],
         'Velocity (mph)': v[:10],
         'Height (ft)': h[:10],
     })
```

② ③ ④

	Time (s)	Velocity (mph)	Height (ft)
0	0.0000	0.000000	0.000000e+00
1	0.0001	0.006797	4.998742e-07
2	0.0011	0.074775	6.048872e-05
3	0.0111	0.755290	6.163369e-03
4	0.0211	1.437137	2.228541e-02
5	0.0311	2.120323	4.844625e-02
6	0.0411	2.804849	8.466560e-02
7	0.0511	3.490722	1.309632e-01
8	0.0611	4.177943	1.873589e-01
9	0.0711	4.866519	2.538726e-01

Run model_rocket.ipynb – **Cell 6**

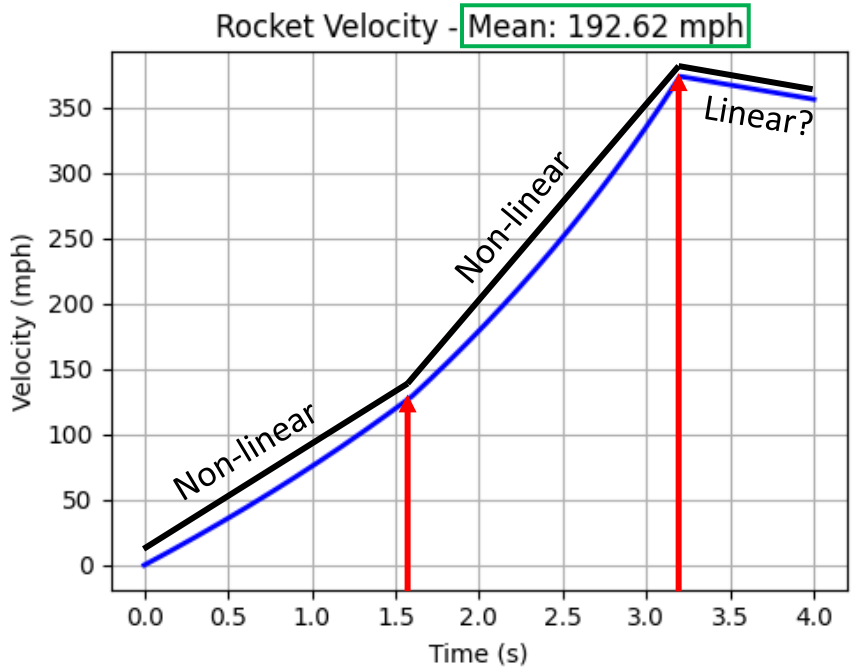
Plot the rocket velocity and altitude over time ①

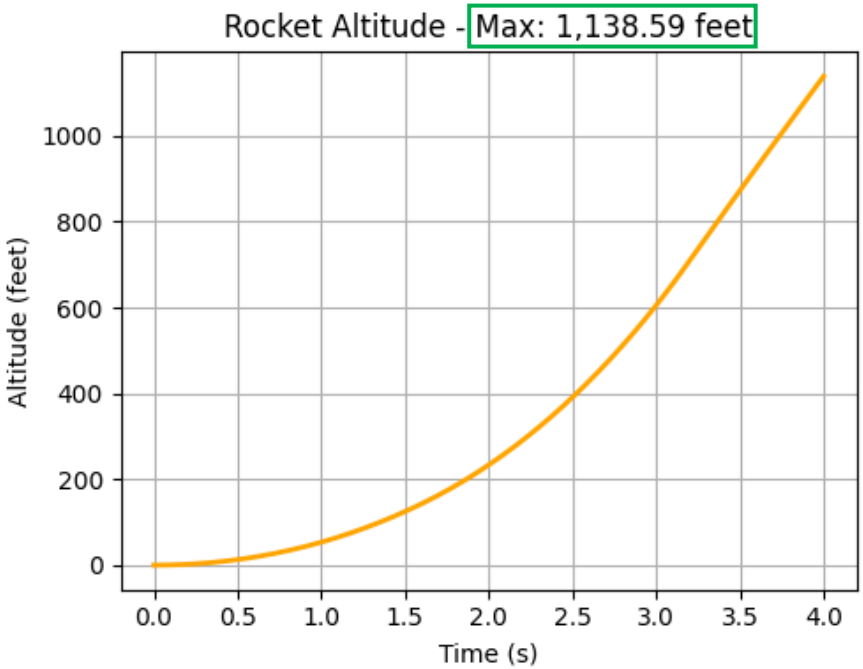
Include the `mean velocity` and `maximum height` over four seconds of flight

```
[6]  # Cell 6
     plt.figure(figsize=(10, 4))
     ax = plt.subplot(1, 2, 1)
     ax.plot(t, v, color="blue", lw=2)
     ax.set_title(f"Rocket Velocity - Mean: {np.mean(v):.2f} mph")
     ax.set_xlabel("Time (s)")
     ax.set_ylabel("Velocity (mph)")
     ax.grid("on")
     ax = plt.subplot(1, 2, 2)
     ax.plot(t, h, color="orange", lw=2)
     ax.set_title(f"Rocket Altitude - Max: {np.max(h):,.2f} feet")
     ax.set_xlabel("Time (s)")
     ax.set_ylabel("Altitude (feet)")
     ax.grid("on")
     plt.tight_layout()
     plt.show()
```

② ③

Check model_rocket.ipynb – **Cell 6**



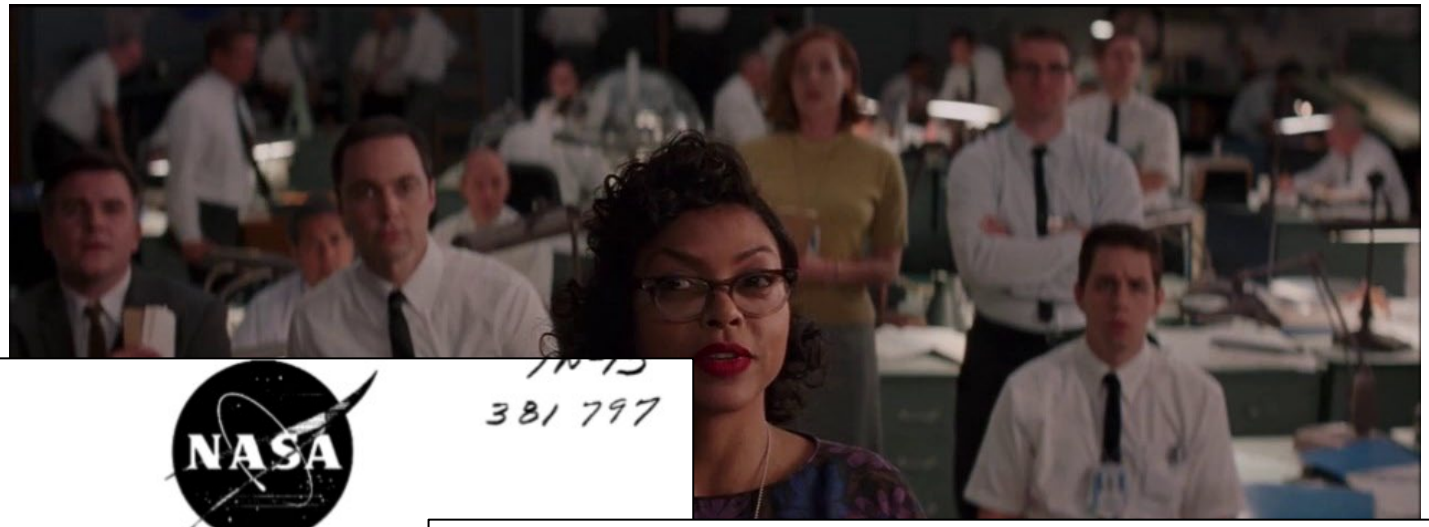


1st stage booster

2nd stage main

No single analytic equation can be derived that models the entire flight

Hidden Figures, Fox 2000 Pictures, 2016



381 797

NASA

TECHNICAL NO

D-233

DETERMINATION OF AZIMUTH ANGLE AT BURN

SATELLITE OVER A SELECTED EARTH

By T. H. Skopinski and Katherine G.

Langley Research Center
Langley Field, Va.

Time from perigee is expressed as

$$t(\theta) = \frac{T}{2\pi}(E - e \sin E) \tag{8}$$

Eccentric anomaly (fig. 1(b)) is given by

$$E = 2 \tan^{-1}\left(\sqrt{\frac{1 - e}{1 + e}} \tan \frac{\theta}{2}\right)$$

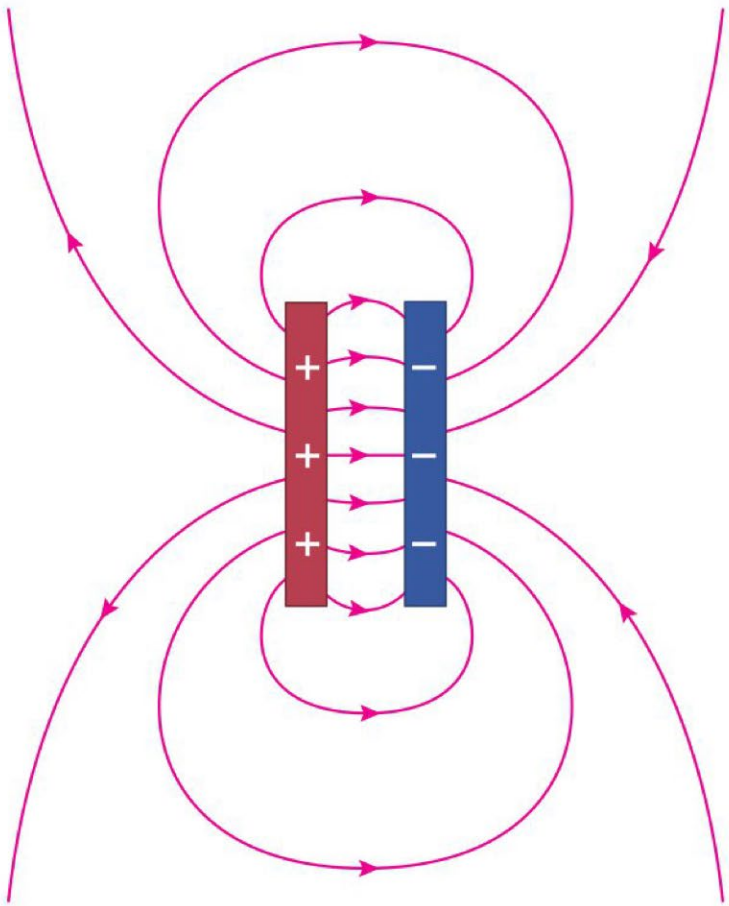
Runge-Kutta Method

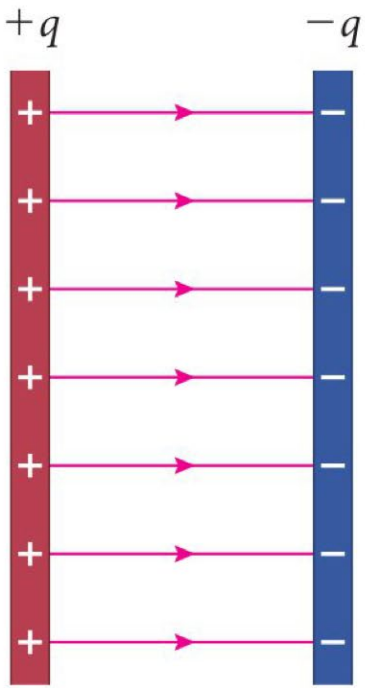
In the use of equations (19) and (20) an iterative procedure is required, since the time $t(\theta_{2e})$ from perigee to the equivalent position is not known initially. A satisfactory first approximation is to assume that

Electrostatic Field Between Parallel Plates

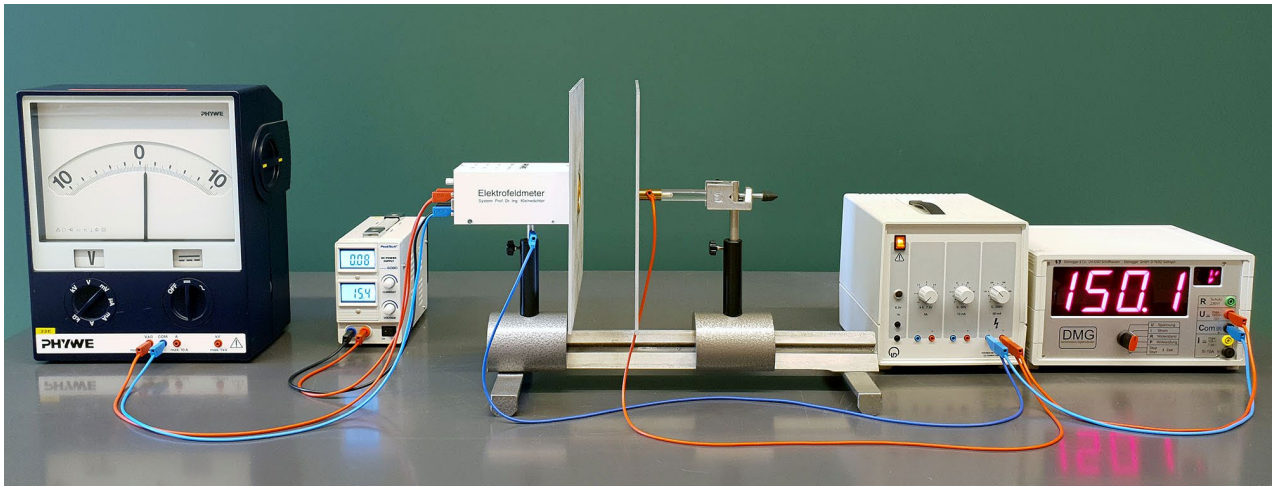
- Model the **electrostatic field** around the **charged parallel plates** inside a capacitor where the surrounding *walls* are conductors (fixed potential) or insulators (fixed charge)
- Numerically estimate solutions to the **Laplace** 2nd order partial differential equation using both **Neumann** and **Dirichlet** boundary conditions
- Use a **convolution kernel** (a stencil) to perform **Jacobi** relaxation over a **discretized grid**

Electrostatic Field in Parallel Plates

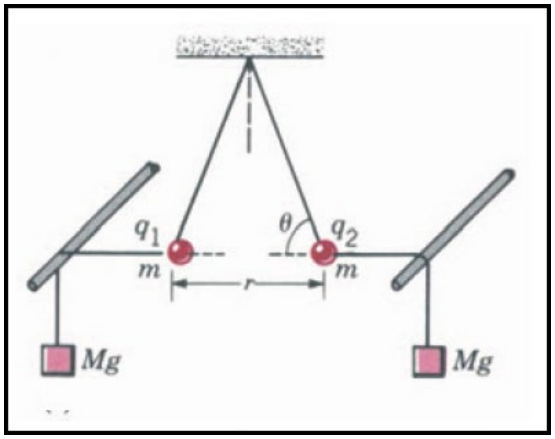




Electrostatic Field in Parallel Plates



Electrostatic Fields



Coulomb's Law

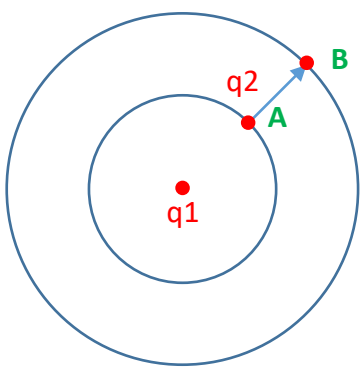
$$F \propto \frac{q_1 q_2}{r^2} \Longrightarrow F = k\frac{q_1 q_2}{r^2}$$

$$k = \frac{1}{4\pi\varepsilon_0} \ (Coulomb's\ constant)$$

$$F = \frac{q_1 q_2}{4\pi\varepsilon_0 r^2}$$

q = Electric charge
ε_0 = Permittivity of free space

*Electric **Potential** (Voltage)*



$$W_{A\to B} = \int_A^B F\,ds$$

$$E(r) = \frac{q_1 q_2}{4\pi\varepsilon_0}\int_A^B \frac{1}{r^2}dr$$

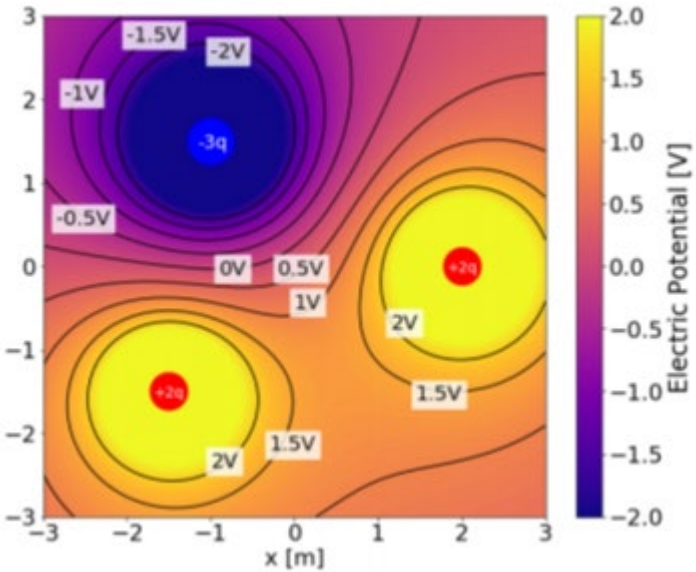
$$\int \frac{1}{r^2} = -\frac{1}{r} \qquad \frac{-1}{r_B} - \frac{-1}{r_A} = \boxed{\frac{1}{r_A} - \frac{1}{r_B}}$$

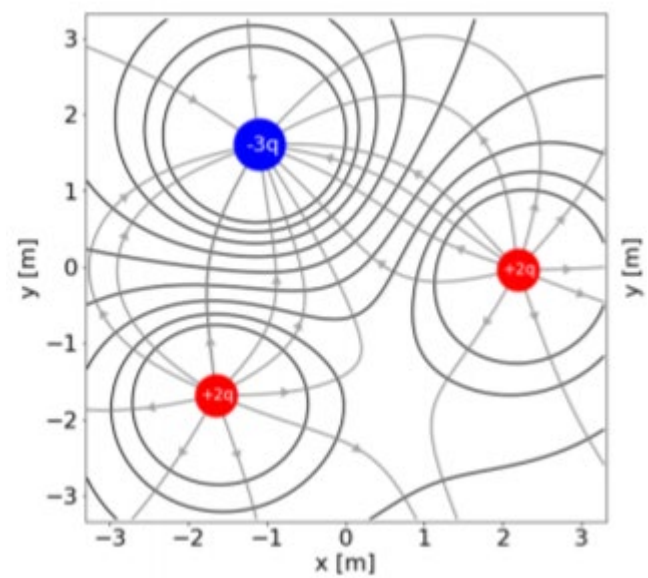
$$E = \frac{q_1 q_2}{4\pi\varepsilon_0}\boxed{\left[\frac{1}{r_A} - \frac{1}{r_B}\right]}$$

A as reference point $\therefore r_B = \infty$

$$\boxed{E_A = \frac{q_1 q_2}{4\pi\varepsilon_0 r_A}}$$

Electric Field Potential Between Charges



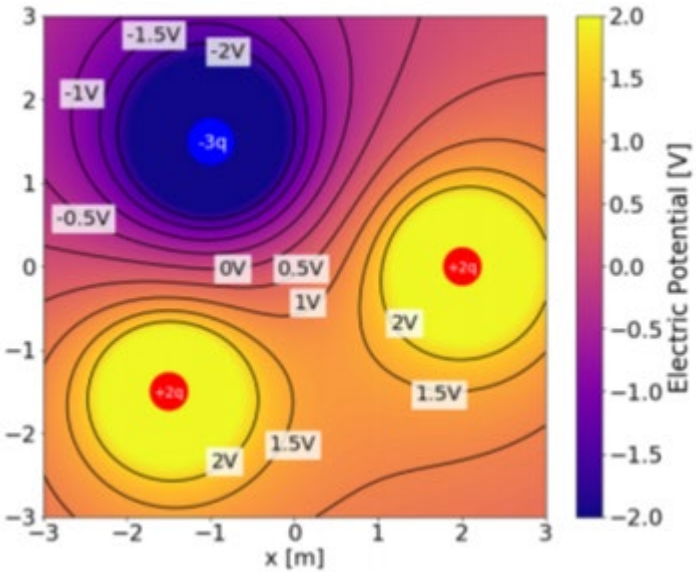


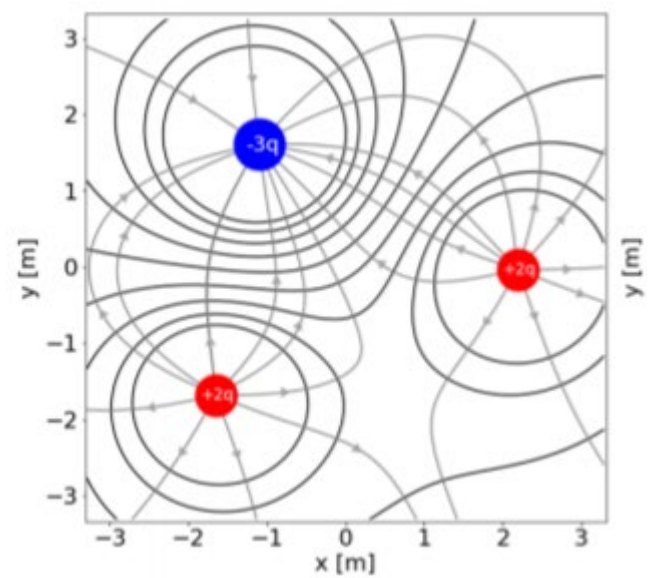
Electric *Potential* (V)
A **scalar** field

?

Electric Field (V/m)
A **vector** field

Electric Field Potential Between Charges





Electric *Potential* (V)
A **scalar** field

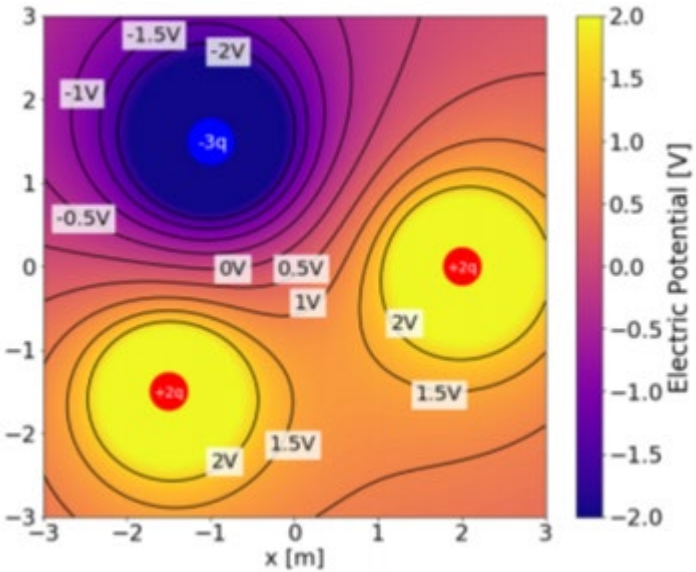
$$\boxed{\vec{E} = -\nabla V}$$

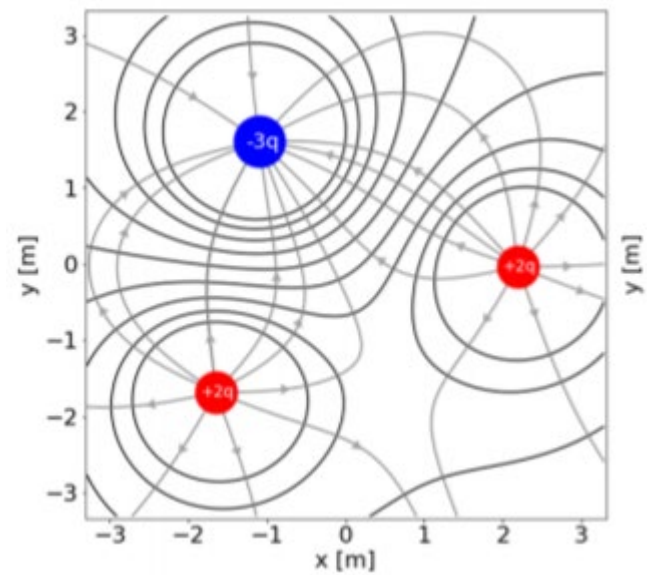
Electric Field (V/m)
A **vector** field

The **negative** sign indicates that the electric field points towards a *decreasing* potential

Physically, this means that a **positive charge** will naturally move from regions of higher potential to regions of *lower potential*

Electric Field Potential Between Charges





Electric *Potential* (V)
A **scalar** field

$$\boxed{\vec{E} = -\nabla V}$$

Electric Field (V/m)
A **vector** field

$$\nabla \cdot \vec{E} = -\nabla^2 V = 0$$

For an electro<u>static</u> field (the field isn't changing)

Electrostatic Field in Parallel Plates

$$\nabla^2 V = 0$$

For a function to be static, it is not enough for the first derivative to be zero – think of the pendulum at the far left or right of its swing…

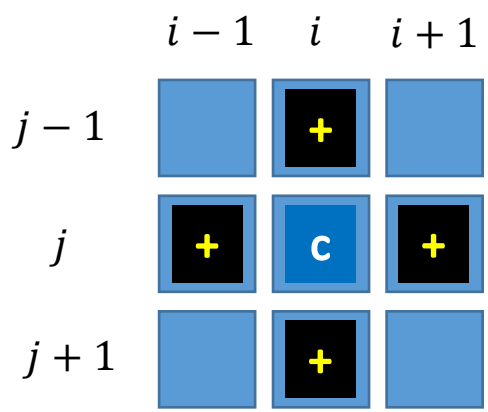
Its 2nd derivative must also be zero!

2D Continuous

$$\nabla^2 V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

Discretization over a 2D Grid

$$V_{i,j} \rightarrow \frac{1}{4}\left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}\right) = 0$$



Electrostatic Field in Parallel Plates

$$\nabla^2 V = 0$$

For a function to be static, it is not enough for the first derivative to be zero – think of the pendulum at the far left or right of its swing…

Its 2nd derivative must also be zero!

2D Continuous

$$\nabla^2 V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

Discretization over a 2D Grid

$$V_{i,j} \rightarrow \frac{1}{4}\left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}\right) = 0$$

A convolution **kernel** (stencil)

$$\begin{bmatrix} 0 & 1/4 & 0 \\ 1/4 & 0 & 1/4 \\ 0 & 1/4 & 0 \end{bmatrix}$$

Electrostatic Field in Parallel Plates

- If we apply the **convolution** of the **kernel** to successive overlapping **3 x 3** ***swatches*** of the field grid (across both the x and y dimensions), we will find the **"next in time"** value of the electric field for each point in the grid
  - A **convolution** involves multiplying each cell in the **3 x 3 kernel** with the corresponding cell in the *current swatch location* (within the grid) and then summing those nine products
  - We **convolve** the entire 2D field grid with the kernel enough times to ensure every **"next in time"** grid cell value is equal to its **"prior in time"** grid cell value – meaning there is **no change over time**
- When this condition is realized, we have reached a steady state and therefore have an electro**static** field. This iterative approach is called the **Jacobi Relaxation Method**

Electrostatic Field in Parallel Plates

- With ordinary differential equations (ODEs), we must specify the **initial conditions** for an exact solution. We must specify **boundary conditions** with partial differential equations (PDEs)
- In our simulation of the **electrostatic field around parallel plates**, we have two choices for boundary conditions:
  - **Neumann boundary conditions**: the plates are surrounded by walls of conductors, thus preventing any **electric potential** from building up along the walls, so the cells around the outermost edges of the grid are forced to maintain the same field strength as their neighboring cells. This makes a **zero gradient** along the walls
  - **Dirichlet boundary conditions**: the plates are surrounded by walls of insulators, thus preventing any **electric charge** from building up along the walls, so the cells around the outmost edges of the grid are forced to maintain a **zero voltage**

Run electrostatic_fields.ipynb – **Cells 1...2**

Import needed packages/modules

```
[1]  # Cell 1
     from pathlib import Path

     import matplotlib.pyplot as plt
     import numpy as np
     from matplotlib.patches import Rectangle
     from scipy.ndimage import convolve, generate_binary_structure
```

①

Define a function to enforce Neumann boundary conditions ②

1. If the walls are `conductors`, the edges must have a zero gradient $(\nabla = 0)$
2. We can force $\nabla = 0$ by ensuring the outermost grid cells have the same potential as their next adjacent inner grid cells ③
3. Note that $\nabla = 0$ does mean the edges must be at zero *voltage* ④
4. Carl Neumann (1832-1925)

```
[2]  # Cell 2
     def conductor_edges(a):
         # Carl Neumann (1832-1925)
         # A conductor forces the edges to
         # have zero potential (gradient=0)
         a[0, :] = a[1, :]
         a[-1, :] = a[-2, :]
         a[:, 0] = a[:, 1]
         a[:, -1] = a[:, -2]
         return a
```

⑤ ⑥ ⑦

Run electrostatic_fields.ipynb – **Cell 3**

Define a function to enforce Dirichlet boundary conditions ①

1. If the walls are `insulators`, the edges must have zero potential $(V = 0)$
2. We can force $V = 0$ by setting the outermost edges in the array to be zero ②
3. Note that $V = 0$ does not mean $\nabla = 0$ at the walls ③
4. Johann Dirichlet (1805-1859)

```
[3]  # Cell 3
     def insulator_edges(a):
         # Johann Dirichlet (1805-1859)
         # An insulator forces the edges to
         # have a fixed charge (voltage=0)
         a[0, :] = 0
         a[-1, :] = 0
         a[:, 0] = 0
         a[:, -1] = 0
         return a
```

④ ⑤ ⑥

Run electrostatic_fields.ipynb – **Cell 4**

Define a function to solve the Laplacian 2nd order PDE $\nabla^2\phi = 0$ for electrostatic fields ①

1. The function receives a matplotlib axes object in which to render the plot
2. The fucntion receives a **function** to enforce the boundary conditions ②
3. The function also receives the voltage of the left and right plates

```
[4]  # Cell 4
     def solve_laplace(ax, boundary_func, left_volts, right_volts):
       N = 100 # Number of 2D grid cells in the x & y directions
       grid = np.zeros((N, N))
       grid[30:70, 29:30] = left_volts
       grid[30:70, 70:71] = right_volts
       # Create masks for plates using boolean indexing
       mask_left = grid == left_volts
       mask_right = grid == right_volts
       # Create a convolution kernel to apply over grid each iteration
       kern = generate_binary_structure(2, 1).astype(float) / 4
       kern[1, 1] = 0
```

③ ④ ⑤ ⑥ ⑦

Run electrostatic_fields.ipynb – **Cell 4**

```
for _ in range(5000): # Number of Jacobi relaxation iterations  ①
    # Create a new grid by using convolution to average
    # every four neighbor cells in the current grid
    grid_next = convolve(grid, kern, mode="constant")  ②
    # Reapply the boundary conditions
    grid_next = boundary_func(grid_next)  ③
    # Reapply the plate voltages
    grid_next[mask_left] = left_volts  ④
    grid_next[mask_right] = right_volts
    # The "next" grid now becomes the current grid
    grid = grid_next  ⑤
# Render a colored contour plot of the electrostatic field potential
surf = ax.contourf(range(N), range(N), grid, cmap="rainbow", levels=20)  ⑥
ax.get_figure().colorbar(surf, ax=ax, shrink=0.5)
# Blacken the two parallel plates
ax.add_patch(Rectangle((29, 30), 1, 40, edgecolor="k", facecolor="k"))
ax.add_patch(Rectangle((70, 30), 1, 40, edgecolor="k", facecolor="k"))  ⑦
# Title the graph
if boundary_func == conductor_edges:  ⑧
    ax.set_title("Conductor Edges")
else:
    ax.set_title("Insulator Edges")
ax.set_aspect("equal")
```

Run electrostatic_fields.ipynb – **Cell 5**

Define a function to display the electrocstatic field between two parallel plates ①

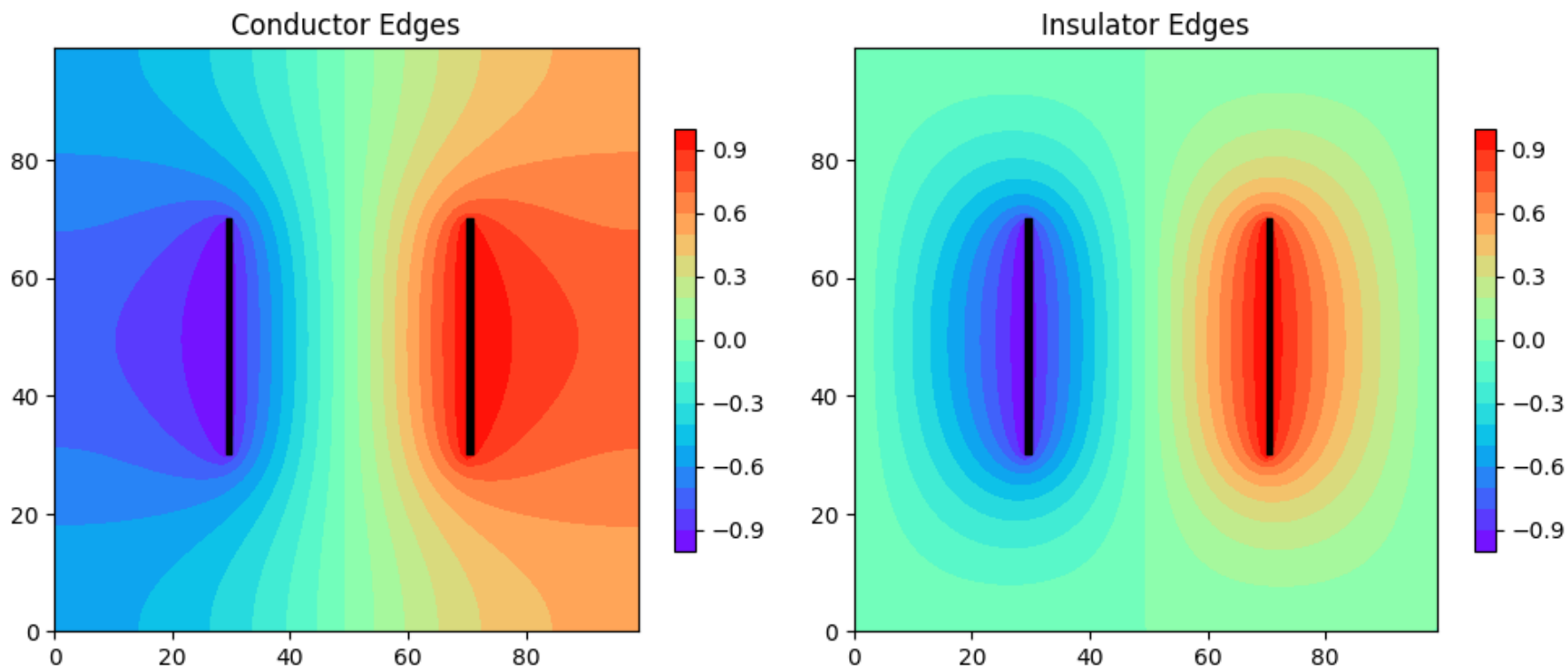
1. The function receives the requested left and right plate voltages
2. The function will display the results of the field between the two plates
3. The situations where the walls are conductors and insulators are both shown

Simulate LEFT plate $= -1V$ and RIGHT plate $= +1V$

```
[5]  # Cell 5
     def plot_plates(left_volts, right_volts):
       plt.figure(figsize=(10, 5.5))
       ax1 = plt.subplot(1, 2, 1)
       ax2 = plt.subplot(1, 2, 2)
       solve_laplace(ax1, conductor_edges, left_volts, right_volts)
       solve_laplace(ax2, insulator_edges, left_volts, right_volts)
       plt.tight_layout()
       plt.show()

     # Show the results where left plate is -1V and right plate is +1V
     plot_plates(-1, 1)
```

② ③ ④

Check electrostatic_fields.ipynb – **Cell 5**



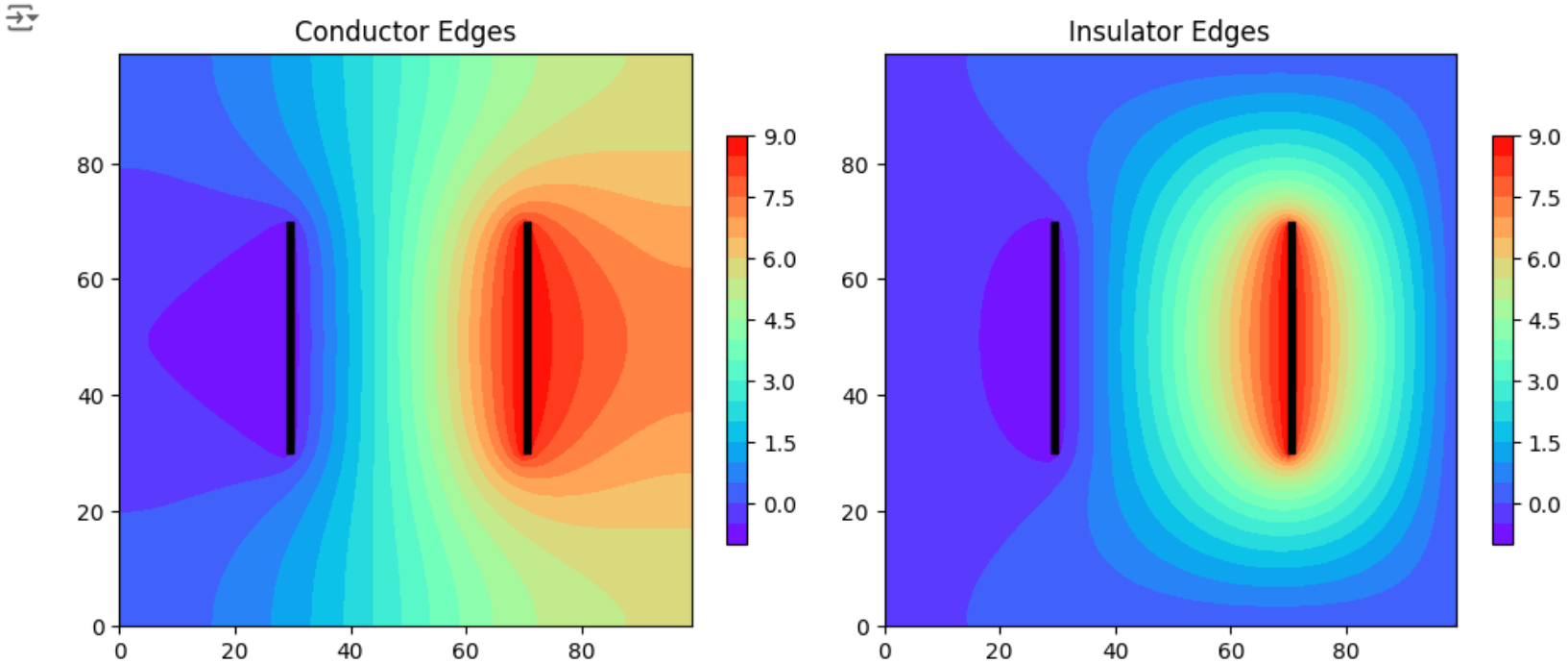
Simulate LEFT plate $= -1V$ and RIGHT plate $= +1V$

Run electrostatic_fields.ipynb – **Cell 6**

Simulate LEFT plate $= -1V$ **and RIGHT plate** $= +9V$

```
[6]  # Cell 6
     plot_plates(-1, 9)
```

①

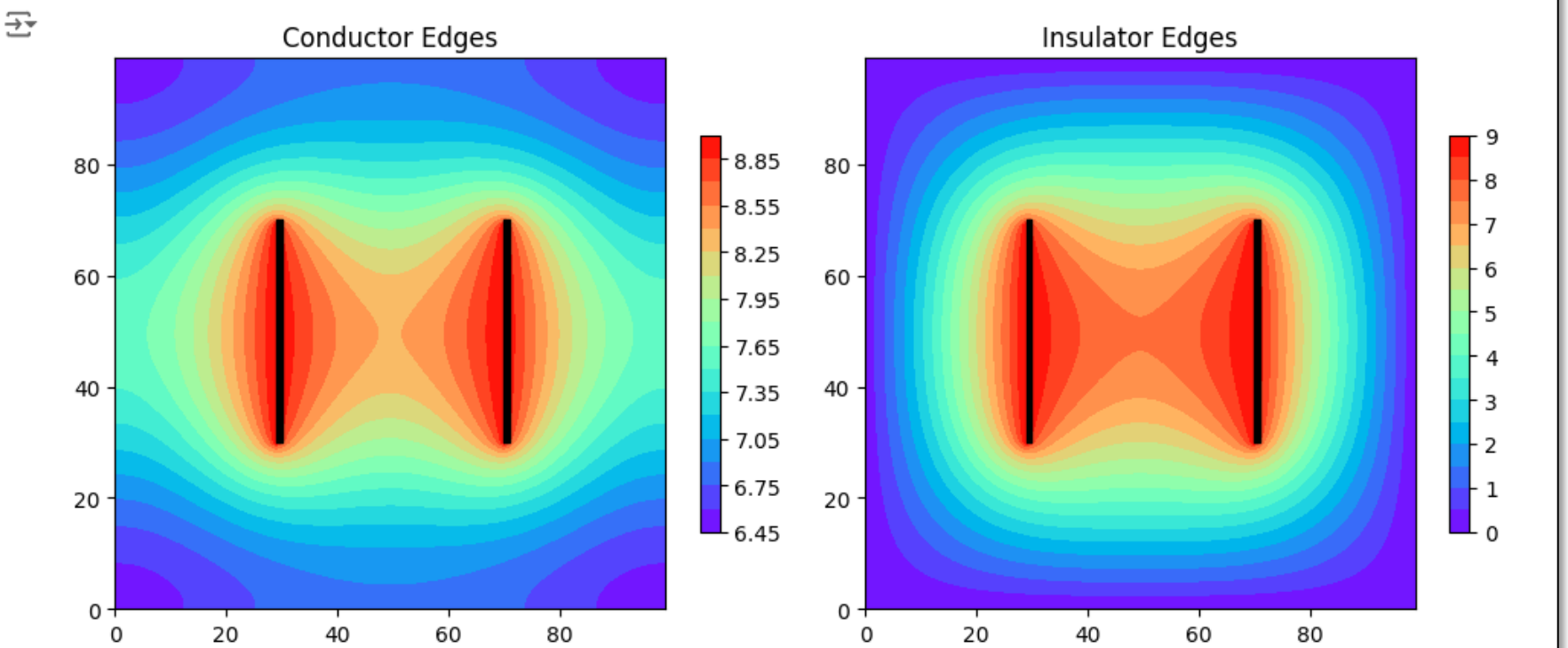


Run electrostatic_fields.ipynb – **Cell 7**

Simulate LEFT plate $= +9V$ **and RIGHT plate** $= +9V$

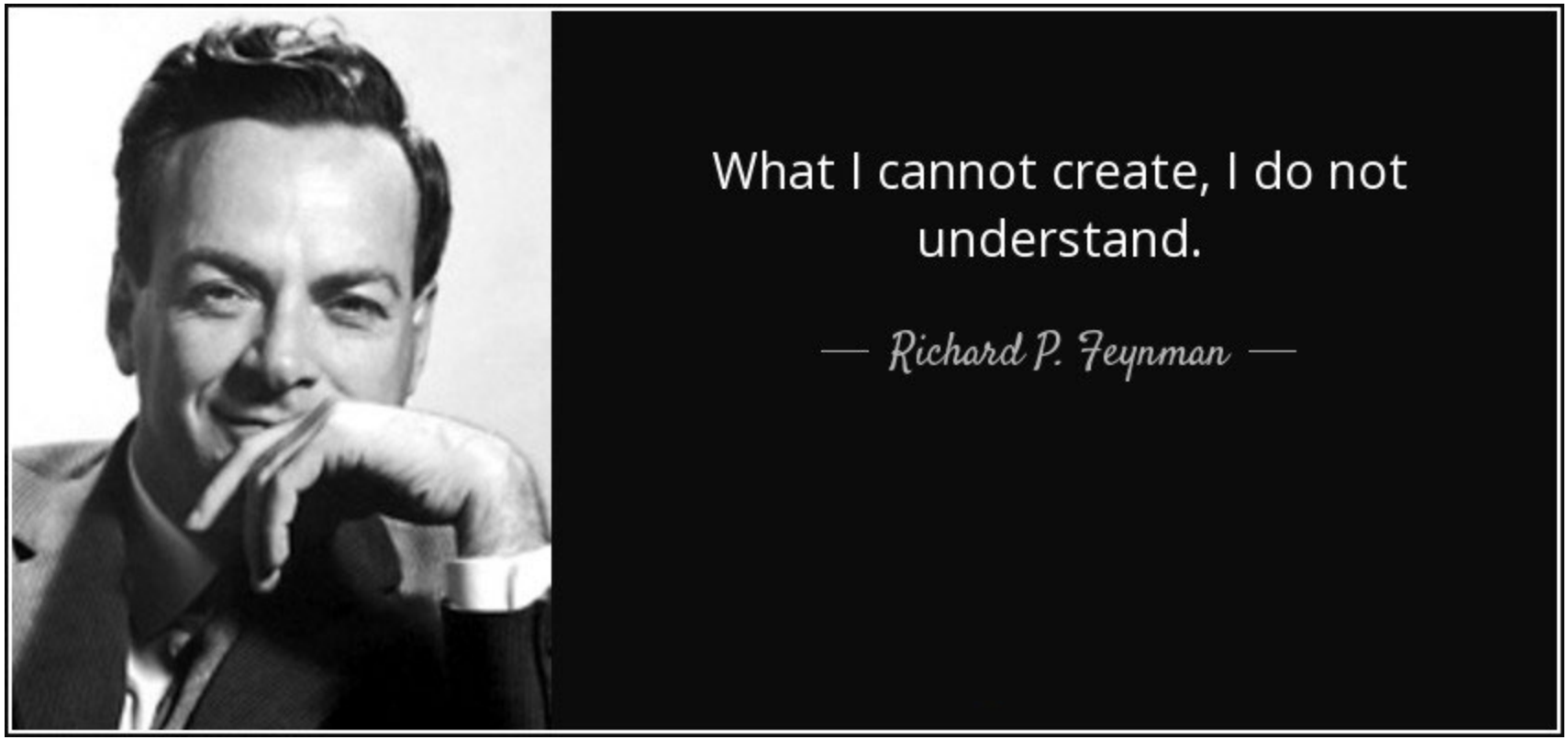
```
[7]  # Cell 7
     plot_plates(9, 9)
```

①



Session **04** – Now You Know…

- Use **SciPy** to simulate numerical solutions to ordinary differential equations (**ODE**s) representing physical laws
- Model the radioactive decay of **Fluorine-18**
- Model a **damped oscillator** to appreciate the impact of **critical damping**
- Simulate the **Dynamical Kinematics** of a **Model Rocket** through lift-off, where the system mass *decreases* as solid motor fuel is burnt to produce thrust
- Model the **electrostatic field** around the **charged parallel plates** inside a capacitor where the surrounding *walls* are <u>conductors</u> (fixed potential) or <u>insulators</u> (fixed charge)



THANK YOU!