Energy

April 25, 2022

[157]:
```
import astropy.units as units
import astropy.constants as constants
import matplotlib.pyplot as plt
import sympy as sym
import numpy as np
import pandas as pd
import plotly.express as px
import requests
a, b, c, d, e, f, g, h, i, j, k, l, m = sym.symbols('a b c d e f g h i j k l m')
n, o, p, q, r, s, t, u, v, w, x, y, z = sym.symbols('n o p q r s t u v w x y z')
symbol_list = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v,
w, x, y, z)
A, B, C, D, E, F, G, H, I, J, K, L, M = sym.symbols('A B C D E F G H I J K L M')
N, O, P, Q, R, S, T, U, V, W, X, Y, Z = sym.symbols('N O P Q R S T U V W X Y Z')
symbol_list = (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,
               V, W, X, Y, Z)
#%matplotlib notebook #incompatible with mpmath
```

#Import half-lifes and energy per emission from databases Zotero Collection / Atomic Mass Data Center (AMDC)

[347]:
```
url = "https://www-nds.iaea.org/amdc/ame2020/mass_1.mas20.txt"
response = requests.get(url)
Atomic_mass_table_2020 = response.text
#Now we want to convert a string to a pandas dataframe
Atomic_mass_table_2020 = list(Atomic_mass_table_2020.split('\n'))
split_table = Atomic_mass_table_2020[36:]

def clean_uncertainty(uncertainty):
    uncertainty = uncertainty.replace('.', '')
    uncertainty = uncertainty.replace('a', '0')
    uncertainty = uncertainty.replace('#', '')
    uncertainty = float("0." + uncertainty)
    return uncertainty

def clean_row(row):
    while True:
        try:
            row[2] = int(row[2])
            number = row.pop(0)
        except:
            row.insert(0, number)
            break
    #The above while loop ensures the first column is the number of neutrons
    try:
        row[4] = float(row[4]) #if this fails, we the row is valid
        row.insert(4, "NA")
    except:
        pass
    try:
        row[10] = row[10].replace('#', '')
        row[10] = float(row[10]) #This means element 9 is *
    except:
        row.insert(11, "NA")
    #if not (len(row) == 15):
    #    print(row, len(row), row[9])
    #print(len(row), row)
    row[12] = float(row[12]) + clean_uncertainty(row[13])
    #this number was formatted weirdly, so we need to clean it up
    row.pop(13)

    return row

for i in range(len(split_table)):
    try:
        split_table[i] = clean_row(split_table[i].split())
    except:
        print(split_table[i].split())
#We know the column names are on row 34 (0-indexed)
#now we will make a pandas dataframe from the list of rows
#Annoyingly, the column names don't include the uncertainties, so we need to add␣
 ↪them
my_column_names = ["N", "Z", "A", "Elt.", "Orig.", "Mass excess (keV)", "Mass␣
 ↪excess (uncertainty)",
 "Binding energy per nucleon (keV)", "Binding energy per nucleon (uncertainty)",
 "Beta-decay Type", "Beta-decay energy (keV)",
 "Beta-decay energy (uncertainty)", "Atomic mass (u)",
 "Atomic mass (uncertainty)"]
Atomic_mass_table_2020 = pd.DataFrame(split_table, columns = my_column_names)
```

```
[]
```

[348]:
```
Atomic_mass_table_2020
```

```
[348]:          N      Z      A  Elt. Orig. Mass excess (keV)  \
0        1    0.0    1.0     n    NA           8071.32
1        0    1.0    1.0     H    NA           7288.97
2        1    1.0    2.0     H    NA           13135.7
3        2    1.0    3.0     H    NA           14949.8
4        1    2.0    3.0    He    NA           14931.2
...    ...    ...    ...   ...   ...               ...
3554   175  118.0  293.0    Og    -a           198802#
3555   177  117.0  294.0    Ts    -a           196397#
3556   176  118.0  294.0    Og    -a           199320#
3557   177  118.0  295.0    Og    -a           201369#
3558  None    NaN    NaN  None  None              None

     Mass excess (uncertainty) Binding energy per nucleon (keV)  \
0                      0.00044                              0.0
1                     0.000013                              0.0
2                     0.000015                        1112.2831
3                      0.00008                        2827.2654
4                      0.00006                       2572.68044
...                        ...                              ...
3554                      709#                            7078#
3555                      593#                            7092#
3556                      553#                            7079#
3557                      655#                            7076#
3558                      None                             None

     Binding energy per nucleon (uncertainty) Beta-decay Type  \
0                                         0.0              B-
1                                         0.0              B-
2                                      0.0002              B-
3                                      0.0003              B-
4                                     0.00015              B-
...                                       ...             ...
3554                                       2#              B-
3555                                       2#              B-
3556                                       2#              B-
3557                                       2#              B-
3558                                     None            None

     Beta-decay energy (keV) Beta-decay energy (uncertainty)  \
0                    782.347                          0.0004
1                          *                              NA
2                          *                              NA
3                     18.592                         0.00006
4                     -13736                           2000#
...                      ...                             ...
3554                       *                              NA
```

```
3555                  -2923                      811#
3556                      *                        NA
3557                      *                        NA
3558                   None                      None

     Atomic mass (u) Atomic mass (uncertainty)
0           1.008665                   0.00047
1           1.007825                  0.000014
2           2.014102                  0.000015
3           3.016049                   0.00008
4           3.016029                   0.00006
...              ...                       ...
3554      293.213423                      761#
3555      294.210840                      637#
3556      294.213979                      594#
3557      295.216178                      703#
3558             NaN                      None

[3559 rows x 14 columns]
```

[350]:
```
#Now we want to write the dataframe to a csv file
Atomic_mass_table_2020.to_csv("Atomic_mass_table_2020.csv")
```

#Now to add a column for half-life

#Calculations for Power vs. Half-life

[162]:
```
def half_life_to_energy(half_life, time, initial_mass, decay_type, molar_mass):
    #we assume half life and time have consistent units (e.g. both in seconds)
    #all masses are in grams
    initial_counts = initial_mass * 6.0221408 * (10**23) / molar_mass
    decay_count = initial_counts * (1-(.5 ** (time / half_life)))
    energy = decay_energy(decay_count, decay_type) / 2 #conservative estimate
    return (energy) #joules, counts

def decay_energy(decay_counts, decay_type):
  if(decay_type == "beta_minus"):
    return decay_counts * 2.7237003 * (10 ** -15) #Joules

half_life_U_238 = 1.41*(10 ** 17) #seconds
half_life_CA_48 = 6.4 * (10 ** 26.5) #seconds
#around 10^9.5 years :)
half_life = 10 ** 11
mass = 10 ** -3 #grams
time = 1 #second
power = half_life_to_energy(half_life, time, mass, "beta_minus",
                    63)
mass, power, time = mass * units.g, power * units.W, time * units.s
```

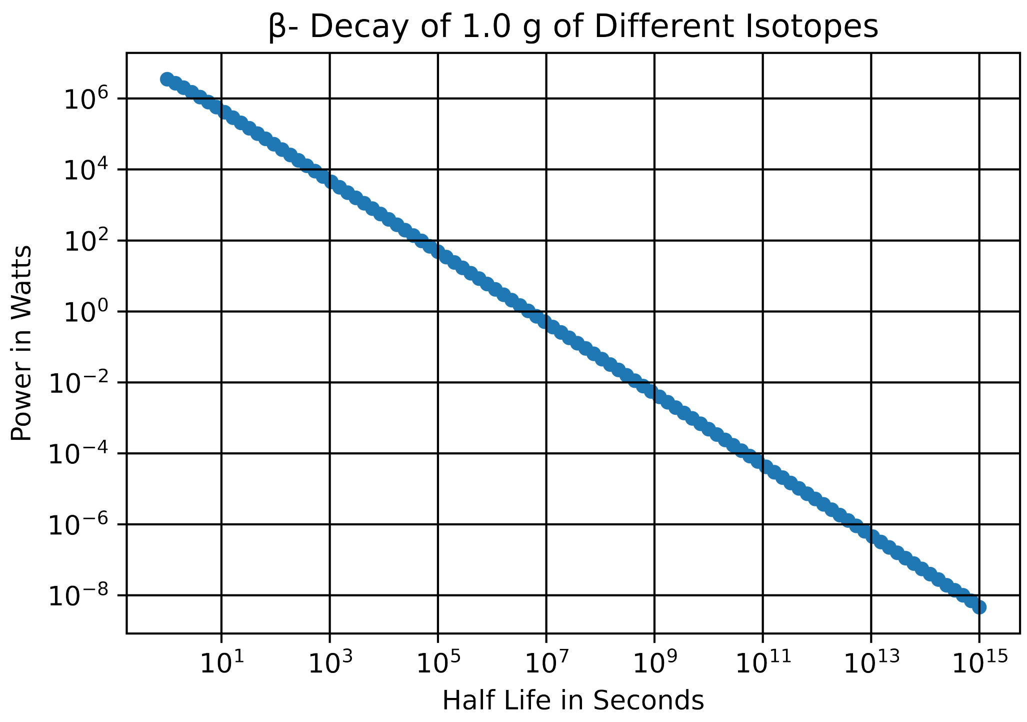
```
print(mass, "generates", power, "and loses", mass,
      "of mass over the first\nsecond :).\n Half of the energy" +
      " will be lost in", (time).to(units.year))
```

```
0.001 g generates 9.023273319236257e-08 W and loses 0.001 g of mass over the
first
second :).
 Half of the energy will be lost in 3.168808781402895e-08 yr
```

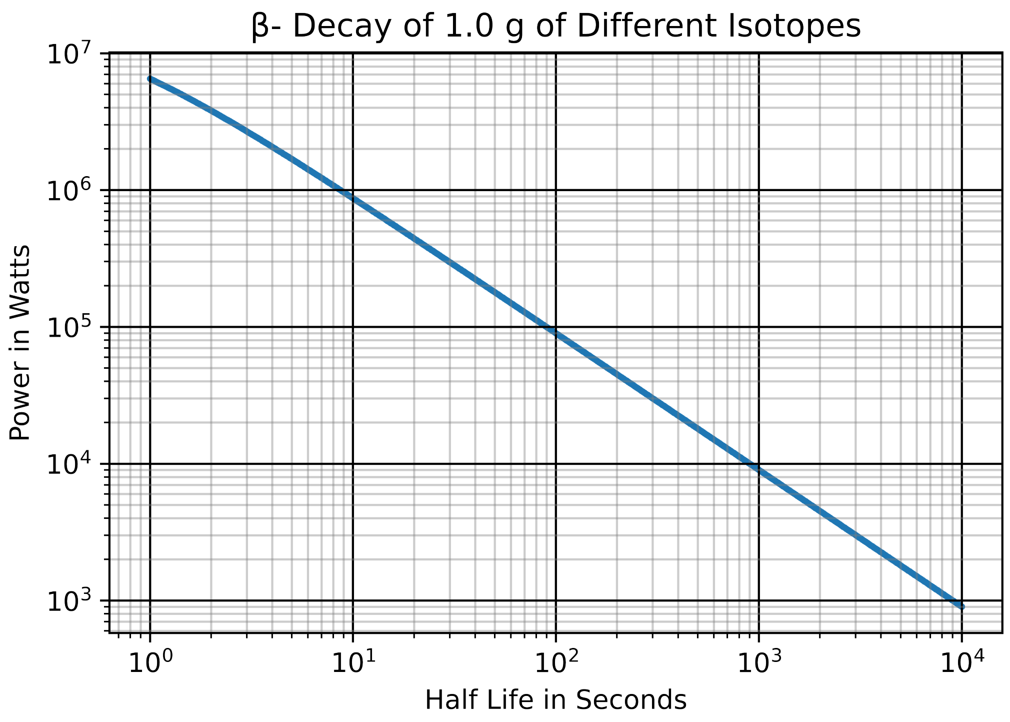
[163]:
```
'''
Here we will plot half-life versus power generated in first second of existence.
Based on the plot this makes, since there are about 10^9.5 seconds in a century,
the ideal half life is around 10^10 seconds. This will mean after a century,
it will only half of the remaining mass (and thus presumably only produce half
of the energy indicated in this plot).

'''
def calc_half_power(min_half_life, max_half_life, steps, mass, molar_mass):
  precise_half_lives, power_array = [], []
  time = 1
  for exponent in np.linspace(min_half_life, max_half_life, steps):
        half_life = 10 ** exponent
        precise_half_lives.append(half_life)
        power = half_life_to_energy(half_life, time, mass, "beta_minus",
                                    molar_mass)
        power_array.append(power)
  return (precise_half_lives, power_array)

def plot_power_vs_half_life(min_half_life, max_half_life, steps, mass,
                            molar_mass, point_size, provide_fit, dpi):
  '''
  Note that the half-lifes give are assumed to be a power of 10
  '''
  precise_half_lives, power_array = calc_half_power(min_half_life,
                                    max_half_life, steps, mass, molar_mass)
  plt.figure(dpi = dpi)
  plt.scatter(x = precise_half_lives, y= power_array, s = point_size)
  plt.xscale("log"), plt.yscale("log")
  plt.title("- Decay of " + str(mass * units.g) + " of Different Isotopes")
  plt.ylabel("Power in Watts")
  plt.xlabel("Half Life in Seconds")
  plt.grid(which='major', color='black')
  plt.grid(which='minor', color='grey', alpha=0.4)
  plt.minorticks_on()
  logX, logY = np.log10(precise_half_lives), np.log10(power_array)
  plt.show()
  if provide_fit:
    return np.polyfit(logX, logY, 1)
plot_power_vs_half_life(0, 15, 100, 1, 118, point_size = 20,
                        provide_fit = False, dpi = 1000)
```



[164]:
```
'''
Interestingly, zooming in we have a half-life of 10^9 seconds (30 years)
corresponds to about 1 W/g.
Conveniently, on this log-log graph, the slope is also about -1.
'''
plot_power_vs_half_life(3, 7, 500, 1, 63, 2, provide_fit = True, dpi = 10**3)
```



```
[164]: array([-0.9999807 ,  6.95525217])
```

```
[165]: half_life_Si_31 = (157 * units.minute).to(units.s) / units.s
       power_Si_31_per_gram = half_life_to_energy(half_life_Si_31, 1, 1, "beta_minus",
                                                  31) * units.W
       power_Si_31_per_gram
```

```
[165]: 1946.6018 W
```

```
[166]: half_life_Ni_63 = (100.1 * units.year).to(units.s) / units.s
       power_Ni_63_per_gram = half_life_to_energy(half_life_Ni_63, 1, 1, "beta_minus",
                                                  63) * units.W
       power_Ni_63_per_gram #200kg could power a car for 50 years :)
```

```
[166]: 0.0028564536 W
```

#Background physics For the nuclear reaction x + X → y + Y we define the Q as the energy from
mass loss:

$$Q = [x_{mass} + X_{mass} - y_{mass} - Y_{mass}]c^2 \tag{1}$$

For β^- decay we have

$$\begin{aligned} n &\rightarrow p + e^- + \bar{\upsilon} \\ Q &= (n_{mass} - p_{mass} - e^-_{mass} - \bar{\upsilon}_{mass})c^2 \\ Q &= .782MeV - \bar{\upsilon}_{mass}c^2 \end{aligned} \tag{2}$$

Assuming a massless neutrino this simplifies to .782 MeV

[167]:
```python
#half_life_to_energy(.987 * 3.1536 * (10 ** 9), 1, 10**-6)[0]
avg_energy_per_decay = 17 * units.keV
mass_predecay = constants.m_n
mass_postdecay = constants.m_p + constants.m_e
Ni63_half_life = 1.02 * 3.1536 * (10 ** 9) * units.s
time = 1 * units.s
initial_mass = (10 ** -6) * units.kg
recalc_energy_per_decay = (mass_predecay - mass_postdecay) * (constants.c ** 2)
recalc_energy_per_decay.to(units.keV) / avg_energy_per_decay
```

[167]:
```
46.019612
```

#Potential Supply Chain https://drive.google.com/file/d/1Mhe_WbmmahkeAE_JPnbYRvphh6IAWPwn/view?u

This is very similar to how neutrons are currently produced in particle accelerators :)

https://drive.google.com/file/d/1wPwC2eu6CqND1JRK_Dgwao7nJKzyh7Xj/view?usp=sharing

https://drive.google.com/file/d/1t76jd7mjup9C_4SqJ5opnoNNZjVcNmSs/view?usp=sharing

Thermal neutrons (0.025 eV) are sufficient (neutron temperature, Bryskin et al. 2004. Figure 4.)

[168]:
```python
class ni63_setup:
  '''
  This will call helper classes depending on the different steps using in the
  production
  '''

  def __init__(self, **kwargs):
    self.assign_attributes(**kwargs)
    self.solar_panel_dict = {}

  def assign_attributes(self, **kwargs):
      for key in kwargs:
          setattr(self, key, kwargs[key])
```

[169]:
```python
class solar_panel:
  '''
  Use to create a solar_panel object with a given efficiency, area, and solar␣
 ↪flux
  '''
  def __init__(self, **kwargs):
    self.assign_attributes(**kwargs)

  def assign_attributes(self, **kwargs):
      for key in kwargs:
        setattr(self, key, kwargs[key])

  def calc_voltage(self):
      '''
    By definition
    power = solar_flux * efficiency * area
    power = voltage * current
    Taking the equivalent resistance of the entire circuit, we have
    voltage = current * resistance
    current = voltage / resistance
    power = voltage^2 / resistance
    voltage = sqrt(power * resistance)
    voltage = sqrt(solar_flux * efficiency * area * resistance)
    '''
    voltage =  np.sqrt(self.solar_flux * self.solar_panel_efficiency *
                       self.solar_panel_area * self.resistance)
    self.voltage = voltage.to(V)

  def calc_charge_plane(self):
  #Modify appropriately for the shape of the capacitor
    self.charge = self.capacitance * self.voltage

  def calc_gamma_ray_flux(self):
    self.gamma_ray_flux = ((self.charge**2) * (self.charge_e_distance**-4) *
                           self.brehm_coeff * self.cathode_ray_flux)

  def calc_ni63_production_speed(self):
    self.ni63_production_speed = (self.gamma_ray_flux * self.donor_cross_section
                                  * self.target_cross_section)
```

[170]:
```
def calc_brehm_coeff():
    echarge = 1.6021766 * (10 ** -19) * C
    return ((echarge**4) / (96 * ((math.pi * c * 8.8541878128 * (10**-12) * F /␣
  ↪m)**3) *
                     (m_e ** 2)))

def calc_cathode_ray_flux():
    return
```

[171]:
```
'''
https://www.thermofisher.com/order/catalog/product/1517021A

50 W is sufficient for 10^8 n/s
'''
```

[171]:
```
'\nhttps://www.thermofisher.com/order/catalog/product/1517021A \n\n50 W is
sufficient for 10^8 n/s\n'
```

#Theoretical Maximum Production per m^2 solar panel

With a power input of P and a E_γ joules for each gamma ray, assuming each gamma ray has a ρ probability of producing ^{63}Ni we have

$$N = \frac{P\rho}{E_\gamma} \tag{3}$$

[172]:
```python
def g_per_year(solar_flux, area, efficiency, energy_per_neutron, molar_mass):
  power = solar_flux * area * efficiency
  nGamma = power / energy_per_neutron
  isotopes_per_second = nGamma.to(units.s ** -1)
  molPerS = (isotopes_per_second / (constants.N_A)).to(units.mol * units.s ** -1)
  return (molPerS * (molar_mass)).to(units.g * units.year ** -1)

solar_flux = 1000 * units.W / (units.m ** 2)
area = 1 * units.m ** 2
efficiency = 1 #any real world factors that affect the production rate
energy_per_neutron = .075 * units.MeV
molar_mass = 63 * units.g / units.mol
rate = g_per_year(solar_flux, area, efficiency, energy_per_neutron, molar_mass)
rate
```

[172]: $274.74004\ \frac{\mathrm{g}}{\mathrm{yr}}$

##It will take 5,000 years/m^2 to make enough for a car to work for 50 years. We have a theoretical maximum of 10^{-6} mol nickel-63 per second for every square meter of sunlight collected per second = 36 moles per year > 2kg. Every car will need 5 years. We need 400 million m^2

[173]:
```python
earth_rad = 6400 * units.km
earth_surface_area = np.pi * 4 * (earth_rad ** 2)
panels_area = 400 * (10**6) * (units.m ** 2)
panels_area.to(units.km ** 2) / earth_surface_area
```

[173]: 7.7712375×10^{-7}

#We need to cover 1-millionth of the Earth in solar panels. More realistically, 400 mi^2

##Caltech People who have done photoneutron work * S.R. Golwala * T. Aralis

##What is the energy per gamma ray and probability of Ni63 production needed to make this technology competiting in the long term compared to current forms of energy storage?

The emission of radiation by accelerating charges is derived in Chapter 14 and 15 of the 2nd edition of Jackson's E&M.

[174]:
```python
'''
Source:
https://www.iea.org/reports/key-world-energy-statistics-2021/final-consumption
```

```
'''
annual_global_energy_consumed = (450 * units.EJ).to(units.W * units.year)
global_power_consumed = annual_global_energy_consumed / (1 * units.year)
efficiency = .2
power_per_area = 500 * units.W / (units.m ** 2) * efficiency
area_needed = (global_power_consumed / power_per_area).to(units.km ** 2)
area_needed
```

[174]: $142596.4\ \mathrm{km}^2$

[175]:
```
global_power_consumed / power_Si_31_per_gram
```

[175]: 7.3254015×10^{9}

[176]:
```
global_power_consumed
```

[176]: $1.425964 \times 10^{13}\ \mathrm{W}$

[ ]:
```
```

[177]:
```
area_California_state = 423970.694 * (units.km**2)
area_California_state / area_needed
```

[177]: 2.9732217

#If we cover 1/3 of California in solar panels, we could power the world.

#We need to cover half a million square miles with solar panels and rapidly replant native flora.

[178]:
```
#https://doi.org/10.1103/PhysRevX.7.041003
input = 40 * units.PW
Egamma= 2 * units.MeV
max_photon_flux = (input/Egamma).to(units.s ** -1)
max_photon_flux
```

[178]: $1.2483018 \times 10^{29}\ \frac{1}{\mathrm{s}}$

##Rewrite clas to be based on generating the right energy neutrons for Ni63 production.

[179]:
```
frequency = ((2 * units.MeV).to(units.J) / constants.h).to(units.Hz)
frequency
```

[179]: $4.8359785 \times 10^{20}\ \mathrm{Hz}$

#A system that can convert between different isotopes depending on the power demand would be ideal.

- Requires a fast, compact, and energy efficient way to convert between emitted electrons and neutrons.

Possibilities include using the Brehmsstrahlung effect to create gamma rays which can then be used to generate photoneutrons. This seems like the crux of the system.

-

[180]:
```
def total_brehmsstrahlung_power(velocity, charge, acceleration):
  '''
  Source:
  https://en.wikipedia.org/wiki/Bremsstrahlung#Total_radiated_power
  '''
  beta = velocity / constants.c
  gamma = (1 - (beta ** 2)) ** -.5
  beta_dot = (acceleration / constants.c)
  beta_term = (beta_dot ** 2) + ((beta * beta_dot) ** 2)/(1 - (beta ** 2))
  power = (charge ** 2) * (gamma ** 4) * beta_term / (6 *
                float(sym.N(sym.pi)) * constants.eps0 * constants.c)
  return power.to(units.W)
```

[181]:
```
total_brehmsstrahlung_power(.0000001 * constants.c, 1 * constants.e.si, .0000001␣
 ↪*
                          constants.c / units.s)
```

[181]: $5.1303882 \times 10^{-51}$ W

[182]:
```
total_energy_stored = half_life_Ni_63 * units.s * power_Ni_63_per_gram
total_energy_stored.to(units.TW * units.hour) #underestimate, and also per gram
```

[182]: 2.5064712×10^{-9} TW h

#Using the technology of the Andasol Solar Power Station, an area half the size of Pennsylvania could power the entire world.

[183]:
```
#https://en.wikipedia.org/wiki/Andasol_Solar_Power_Station
andasol = 2000 * units.kW * units.hour / ((units.m ** 2 ) * units.year)
andasol_efficiency = andasol.to(units.W / (units.m ** 2))
andasol_efficiency
```

[183]: $228.15423 \ \frac{\mathrm{W}}{\mathrm{m}^2}$

[184]:
```
needed_area = (global_power_consumed / andasol_efficiency).to(units.km ** 2)
needed_area
```

[184]: 62500 km^2

[185]:
```
pennsylvania_area = 119281.9 * (units.km ** 2)
needed_area / pennsylvania_area
```

[185]: 0.52396885

[186]:
```
neutrino_mass = (1 * units.keV / (constants.c ** 2)).to(units.g)
neutrino_mass
```

[186]: $1.7826619 \times 10^{-30}$ g

#Simualted Spectrum Using the Fermi Distribution Approximated Energy Spectrum

$$N(T_e) = \frac{C}{c^5}(Q - T_e)^2(T + m_e c^2)\sqrt{T_e^2 + 2T_e m_e c^2} \tag{4}$$

Integrating this from 0 to Q and then normalizing the distribution

$$\int_0^{T_e \ max} N(T_e) dT_e$$
$$\int_0^{T_e \ max} \frac{C}{c^5}(Q - T_e)^2(T_e + m_e c^2)\sqrt{T_e^2 + 2T_e m_e c^2 dT_e} = 1 \tag{5}$$
$$C = \frac{c^5}{\int_0^Q (Q - x)^2(x + a)\sqrt{x^2 + 2adx}}$$

For some reason, WolframAlpha won't evaluate the integral. So using sympy, where Q = .782 MeV, a = m$_e$c^2, and $x = T_e$ All units are in kg-m-s SI.

[187]:
```
Q_fermi_distr = float((.782 * units.MeV).to(units.J) / units.J)
Q_fermi_distr
```

[187]:
```
1.252902127788e-13
```

[188]:
```
unnormalized_fermi_fun = ((Q - x) ** 2) * (x + a) * sym.sqrt(
    (x ** 2) + (2 * a))
denominator = sym.integrate(unnormalized_fermi_fun, (x, 0, sym.oo))
```

[189]:
```
a_fermi_distr = float((constants.m_e * (constants.c ** 2)).to(units.J)
/ units.J)
a_fermi_distr
```

[189]:
```
8.187105776823886e-14
```

[190]:
```
c_to_the_fifth = float((constants.c * units.s / units.m) ** 5)
c_to_the_fifth
```

[190]:
```
2.4216061708512208e+42
```

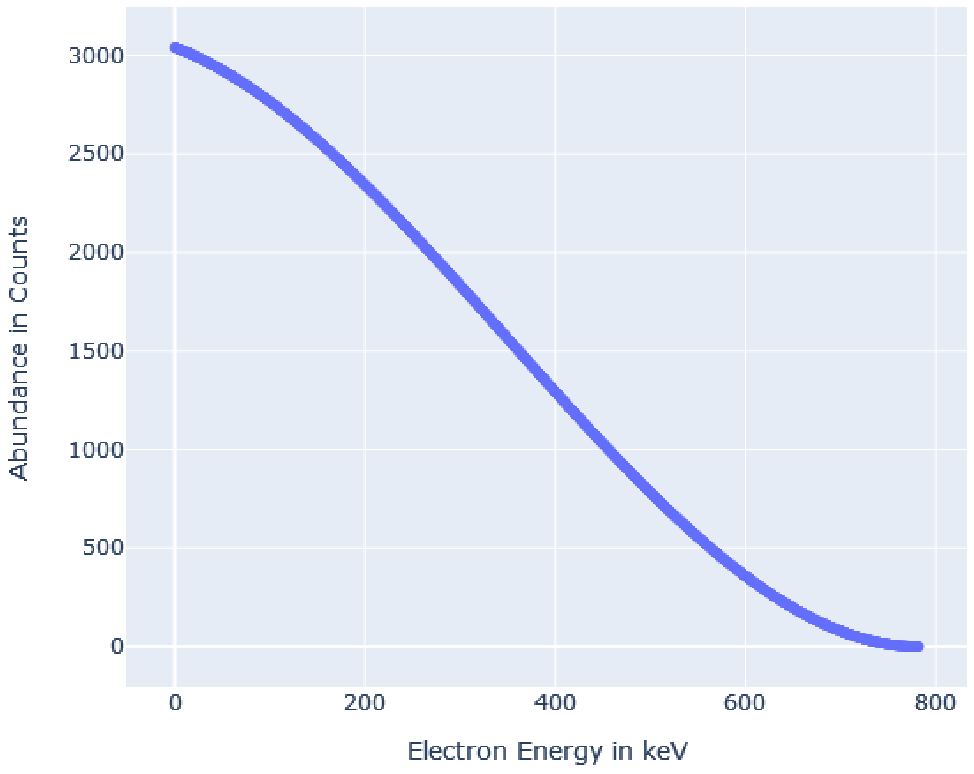
[191]:
```
integrated_dist = sym.lambdify((Q, a), denominator)
fermi_distr_const = c_to_the_fifth / integrated_dist(Q_fermi_distr,
                                                      a_fermi_distr)
```

[192]:
```
fermi_distr_fun = C * unnormalized_fermi_fun / c_to_the_fifth
fermi_distr = sym.lambdify((Q, a, C, x), fermi_distr_fun)

electron_energies = []
abundance = []
```

```
log_res = 3
for electron_energy in np.linspace(0, Q_fermi_distr, 10 ** log_res):
  abundance.append(fermi_distr(Q_fermi_distr, a_fermi_distr,
                              fermi_distr_const, electron_energy))
  electron_energy_keV = float((electron_energy * units.J).to(units.keV)
        / units.keV)
  electron_energies.append(electron_energy_keV)
xlabel, ylabel = 'Electron Energy in keV', 'Abundance in Counts'
data = {xlabel: electron_energies, ylabel: abundance}
Ni63_spectrum = pd.DataFrame.from_dict(data)
fig = px.scatter(Ni63_spectrum, x = xlabel, y = ylabel)
fig.show()
```

[193]:
```
fermi_distr_fun
```



[193]: $4.12949063327043 \cdot 10^{-43} C \left(Q - x\right)^{2} \left(a + x\right) \sqrt{2a + x^{2}}$

[194]:
```
average_fun = (sym.integrate(fermi_distr_fun, (x, 0, fermi_distr_fun)) /
               fermi_distr_fun)
average = sym.lambdify((Q, a, C, x), average_fun)
average(Q_fermi_distr, a_fermi_distr, fermi_distr_const, Q_fermi_distr)
```

```
<lambdifygenerated-7>:2: RuntimeWarning:

invalid value encountered in double_scalars
```

[194]:
```
nan
```

[195]:
```
def normalized_counts(T_e, Q):
  count = (np.sqrt(((T_e **  2) + 2 * (T_e * m_e * (c ** 2))))
            * ((Q - T_e) ** 2) * (T_e + m_e * (c ** 2)) / (c ** 5))
  return count
```

[196]:
```
fermi_distr_const
```

[196]:
```
0.0
```

[197]:
```
'''
We need to account for the fact that not all of the electrons will have the
maximum energy (Q value)!
stuck for right now. Use
https://github.com/MarcosP7635/Computing-and-Formatting/blob/main/
 ↪error_propagation.py
as a sympy reference
They average out to 17 keV. Rewrite to draw from a database of beta emission
spectra.
'''
```

[197]:
```
'\nWe need to account for the fact that not all of the electrons will have the
\nmaximum energy (Q value)!\nstuck for right now.
Use\nhttps://github.com/MarcosP7635/Computing-and-
Formatting/blob/main/error_propagation.py\nas a sympy reference\nThey average
out to 17 keV. Rewrite to draw from a database of beta emission \nspectra. \n'
```

[198]:
```
global_power_consumed
```

[198]: 1.425964×10^{13} W

[199]:
```
energy_Ni_63_per_gram = (power_Ni_63_per_gram * 50 * units.year).to(units.J)
'''
The best lithium-ion batteries store less than 1 kJ/g
Source: https://doi.org/10.1039/D0EE02681F
'''
energy_Ni_63_per_gram
```

[199]:

4507141 J

[200]:
```
'''
if google drive won't mount to the colab session, then
you need to download this current python notebook and upload
it to the colab session, then right click on it to copy the
path for the command below.
'''
!jupyter nbconvert --to LaTeX /Energy.ipynb
#The above line makes a .tex file to format this Jupyter Notebook
```

```
This application is used to convert notebook files (*.ipynb) to various other
formats.

WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options

-------



[NbConvertApp] WARNING | pattern '/Energy.ipynb' matched no files


Arguments that take values are actually convenience aliases to full
Configurables, whose aliases are listed on the help line. For more information
on full configurables, see '--help-all'.


--debug

    set log level to logging.DEBUG (maximize logging output)

--generate-config

    generate default config file

-y

    Answer yes to any questions instead of prompting.

--execute

    Execute the notebook prior to export.

--allow-errors

    Continue notebook execution even if one of the cells throws an error and
```

```
include the error message in the cell output (the default behaviour is to abort
conversion). This flag is only relevant if '--execute' was specified, too.

--stdin

    read a single notebook file from stdin. Write the resulting notebook with
default basename 'notebook.*'

--stdout

    Write notebook output to stdout instead of files.

--inplace

    Run nbconvert in place, overwriting the existing notebook (only
    relevant when converting to notebook format)

--clear-output

    Clear output of current file and save in place,
    overwriting the existing notebook.

--no-prompt

    Exclude input and output prompts from converted document.

--no-input

    Exclude input cells and output prompts from converted document.
    This mode is ideal for generating code-free reports.
--log-level=<Enum> (Application.log_level)

    Default: 30

    Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR',
'CRITICAL')

    Set the log level by value or name.

--config=<Unicode> (JupyterApp.config_file)

    Default: ''

    Full path of a config file.

--to=<Unicode> (NbConvertApp.export_format)

    Default: 'html'
    The export format to be used, either one of the built-in formats
    ['asciidoc', 'custom', 'html', 'html_ch', 'html_embed', 'html_toc',
    'html_with_lenvs', 'html_with_toclenvs', 'latex', 'latex_with_lenvs',
    'markdown', 'notebook', 'pdf', 'python', 'rst', 'script', 'selectLanguage',
    'slides', 'slides_with_lenvs'] or a dotted object name that represents the
    import path for an `Exporter` class
--template=<Unicode> (TemplateExporter.template_file)
    Default: ''
    Name of the template file to use
--writer=<DottedObjectName> (NbConvertApp.writer_class)
    Default: 'FilesWriter'
    Writer class used to write the  results of the conversion
--post=<DottedOrNone> (NbConvertApp.postprocessor_class)
    Default: ''
    PostProcessor class used to write the results of the conversion
--output=<Unicode> (NbConvertApp.output_base)
    Default: ''
    overwrite base name use for output files. can only be used when converting
    one notebook at a time.
--output-dir=<Unicode> (FilesWriter.build_directory)
    Default: ''
    Directory to write output(s) to. Defaults to output to the directory of each
    notebook. To recover previous default behaviour (outputting to the current
    working directory) use . as the flag value.
```

```
--reveal-prefix=<Unicode> (SlidesExporter.reveal_url_prefix)

    Default: ''

    The URL prefix for reveal.js (version 3.x). This defaults to the reveal CDN,

    but can be any url pointing to a copy  of reveal.js.

    For speaker notes to work, this must be a relative path to a local  copy of

    reveal.js: e.g., "reveal.js".

    If a relative path is given, it must be a subdirectory of the current

    directory (from which the server is run).

    See the usage documentation

    (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-html-

    slideshow) for more details.

--nbformat=<Enum> (NotebookExporter.nbformat_version)

    Default: 4

    Choices: [1, 2, 3, 4]

    The nbformat version to write. Use this to downgrade notebooks.

To see all available configurables, use `--help-all`

Examples
--------

    The simplest way to use nbconvert is

    > jupyter nbconvert mynotebook.ipynb

    which will convert mynotebook.ipynb to the default format (probably HTML).

    You can specify the export format with `--to`.
    Options include ['asciidoc', 'custom', 'html', 'html_ch', 'html_embed',
'html_toc', 'html_with_lenvs', 'html_with_toclenvs', 'latex',
'latex_with_lenvs', 'markdown', 'notebook', 'pdf', 'python', 'rst', 'script',
'selectLanguage', 'slides', 'slides_with_lenvs'].
```

```
> jupyter nbconvert --to latex mynotebook.ipynb

Both HTML and LaTeX support multiple output templates. LaTeX includes
'base', 'article' and 'report'.  HTML includes 'basic' and 'full'. You
can specify the flavor of the format used.

> jupyter nbconvert --to html --template basic mynotebook.ipynb

You can also pipe the output to stdout, rather than a file

> jupyter nbconvert mynotebook.ipynb --stdout

PDF is generated via latex

> jupyter nbconvert mynotebook.ipynb --to pdf

You can get (and serve) a Reveal.js-powered slideshow

> jupyter nbconvert myslides.ipynb --to slides --post serve

Multiple notebooks can be given at the command line in a couple of
different ways:

> jupyter nbconvert notebook*.ipynb
> jupyter nbconvert notebook1.ipynb notebook2.ipynb

or you can specify the notebooks list in a config file, containing::

    c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

> jupyter nbconvert --config mycfg.py
```

```
[ ]:
```