In [2]:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import time

class State:

    def __init__(self, positions, masses=1, timestep=0.001, nsteps=10000):
        '''
        Initialize the initial position of each atom\n
        Based on intial positions, calculate interatomic forces acting on the XYZ of each atom
        '''
        self.positions       = positions
        self.natm, self.ndim = positions.shape
        self.update_interatomic_forces()

        self.masses     = np.ones(shape=self.natm) * masses
        self.timestep   = timestep
        self.velocities = np.zeros(shape=(self.natm, self.ndim))
        self.squared_cartesian_distances = np.zeros(shape=(self.natm, self.natm))

        self.nsteps     = nsteps
        self.trajectory = np.zeros(shape=(nsteps, self.natm, self.ndim))
        self.potential_energies = np.zeros(nsteps)
        self.kinetic_energies   = np.zeros(nsteps)
        self.total_energies     = np.zeros(nsteps)

    def step(self):
        '''
        F = m*dv/dt
        v(t + dt) = v + F*dt/m
        x(t+dt)   = x + v*dt
        '''
        self.velocities += self.forces * self.timestep / 2 / self.masses
        self.positions  += self.velocities * self.timestep
        self.update_interatomic_forces()
        self.velocities += self.forces * self.timestep / 2 / self.masses

    def run(self):
        '''
        The following is the only (one) Python loop used in this program.
        The heavy lifting is done by NumPy operations.
        '''
        for step in range(self.nsteps):
            self.step()
            self.trajectory[step]         = self.positions
            self.potential_energies[step] = self.compute_potential_energy()
            self.kinetic_energies[step]   = self.compute_kinetic_energy()
            self.total_energies[step]     = self.potential_energies[step] + self.kinetic_energies[step]

    def update_interatomic_forces(self):
        '''
        Given the xyz coordinates of n atoms,\n
        Calculate the distances between every pair of atoms.\n
        For each atom, calculate the forces that other atoms exert on it on the XYZ directions
        '''
        #https://stackoverflow.com/questions/25965329/difference-between-every-pair-of-columns-of-two-numpy-
        each_atom_coordinates            = self.positions.reshape(self.natm, 1, self.ndim)#from stack overfl
        interatom_distances_per_dimension_per_atom = each_atom_coordinates - self.positions

        self.squared_cartesian_distances  = np.sum(interatom_distances_per_dimension_per_atom**2,axis=2)
        gradients_between_atoms      = State.gradient_return(self.squared_cartesian_distances, 1, 1)
        interatom_gradients_per_atom = gradients_between_atoms.reshape(self.natm, self.natm, 1)

        force_per_dimension_per_atom = interatom_gradients_per_atom * interatom_distances_per_dimension_per_
        force_per_dimension_per_atom[np.isnan(force_per_dimension_per_atom)] = 0
        self.forces = np.sum(force_per_dimension_per_atom,axis=0)

    def compute_potential_energy(self):
        potentials_between_atoms     = State.potential_return(self.squared_cartesian_distances, 1, 1)
        return np.sum(np.triu(potentials_between_atoms,1))

    def compute_kinetic_energy(self):
        return 0.5*np.sum(np.dot(self.masses, self.velocities**2))

    @staticmethod
    def gradient_return(r_t, epsilon, sigma_t):
        z = sigma_t/r_t
        u = z*z*z
        return 24 * epsilon * u * (1 - 2 * u) / r_t

    @staticmethod
    def potential_return(r_t,epsilon,sigma_t):
        z = sigma_t/r_t
        u = z*z*z
```

```
86          return -4*epsilon*u*(1-u)
```

Below I execute the program and test the speed for 10000 steps. It is around 1.9 seconds

In [64]:
```
positions = np.array([
    [0.5391356726,0.1106588251,-0.4635601962], #atom1
    [-0.5185079933,0.4850176090,0.0537084789], #atom2
    [0.0793723207,-0.4956764341,0.5098517173], #atom3
])
state = State(positions)
start  = time.time()
state.run()
print(f"Duration: {time.time()-start} seconds")
```

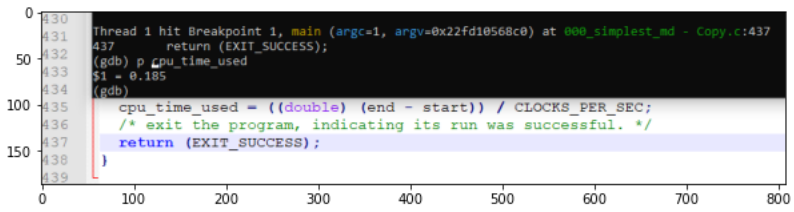
```
<ipython-input-2-340bf0d24bd7>:73: RuntimeWarning: divide by zero encountered in true_divide
  z = sigma_t/r_t
<ipython-input-2-340bf0d24bd7>:60: RuntimeWarning: invalid value encountered in multiply
  force_per_dimension_per_atom = interatom_gradients_per_atom * interatom_distances_per_dimension_per_atom
<ipython-input-2-340bf0d24bd7>:79: RuntimeWarning: divide by zero encountered in true_divide
  z = sigma_t/r_t

Duration: 1.905918836593628 seconds
```

Costa's program takes 0.185 seconds

In [69]:
```
costaPlot = Image.open('CostaEnergyPlot.png')
costaTime = Image.open('CostaTime.png')
fig,ax    = plt.subplots(figsize=(10,10))
ax.imshow(costaTime)
```

Out[69]: <matplotlib.image.AxesImage at 0x17d2e006790>



The plot for the energies is similar to the one produced by Costa's C program.

In [66]:
```
fig = plt.figure()
fig.set_figheight(15)
fig.set_figwidth(15)
gs  = gridspec.GridSpec(100, 100)
ax1 = fig.add_subplot(gs[32:64,:40])
ax2 = fig.add_subplot(gs[:,50:])
ax1.plot(range(10000), state.kinetic_energies, label='kinetic')
ax1.plot(range(10000), state.potential_energies, label='potential')
ax1.plot(range(10000), state.total_energies, label='total')
ax1.set(ylabel='Energy', xlabel='Timesteps')
ax1.legend()
ax1.set(title='Mine')
ax2.imshow(costaPlot)
ax2.set(title='Costa\'s')
ax2.axis('off')
```

Out[66]: (-0.5, 579.5, 420.5, -0.5)