SPEEDUP OF PARALLEL PROCESSING IN MODELING THERMAL DIFFUSION OVER A SURFACE

J. PUGHE-SANFORD

1. Model

The time evolution of temperature in a thermally conducting body is described by the heat equation

$$\frac{\partial \mathbf{U}}{\partial t} = \alpha \nabla^2 \mathbf{U}, \tag{1.1}$$

In this equation, $\mathbf{U} : \vec{r} \mapsto T$ for temperature T and positional argument $\vec{r}$ within a defined region. In other words, $\mathbf{U}$ represents the state of the heat distribution of a system and Equation 1.1 represents the time evolution of that state, in which α is a certain, constant rate of diffusion.

Using this equation, it is possible to solve for the state of a given system at any point in time through integration. To do so numerically, the second derivatives expressed by Equation 1.1 must be approximated by finite differences, as these partial derivatives are not known analytically. Therefore, for each spacial axis q_i in cartesian coordinates,

$$\frac{\partial^2}{\partial q_i^2}\mathbf{U}(\vec{q}, t) \approx \frac{\mathbf{U}(q_i + \epsilon) - 2\mathbf{U}(q_i) + \mathbf{U}(q_i - \epsilon)}{\epsilon^2} \equiv \mathbf{U}_{q_i q_i}(\vec{q}, t) \quad \text{as } \epsilon \to 0,$$

in which ϵ denotes some small step and defines the level of detail in the discrete representation of the system.

Furthermore, as in common in numerical integration, time is not treated as a continuous variable, but is approximated using a time step Δt in the limit of $\Delta t \to 0$. Given an stepping algorithm that steps the state of the system $\mathbf{U}(\vec{q}, t)$ to $\mathbf{U}(\vec{q}, t + \Delta t)$ in time $\eta(\vec{q})$ s, the runtime R, of integrating over the interval $[0, t_f]$ takes

$$R\left[\int_0^{t_f} \mathbf{U} dt\right] \approx \eta(\vec{q}) \frac{t_f}{\Delta t} \text{ s}.$$

Thus, as the integration limit t_f increases, the runtime increases linearly with it.

Integration over discrete space is not as trivial, however. For scalar valued function, the amount of time it takes to step forward the state is proportional to the discrete values that must be updated. Therefore, $\eta(q) \propto \frac{q}{\Delta q}$ s, which results in

$$R\left[\int_0^{t_f} dt \int_0^{r_0} \mathbf{U} dr\right] \approx C \frac{r_0}{\Delta r} \frac{t_f}{\Delta t} \text{ s},$$

for proportionality constant C, the cost of updating one cell using Equation 1.1. Notice that R remains linear here for larger spacial dimensions but that for integrals over higher dimensions, this does not hold. For an n-dimensional array,

$$\eta(\vec{q}) \propto \vec{q} \cdot \vec{\epsilon}\,\text{s}, \qquad \text{for } \epsilon_i = \frac{1}{\Delta q_i},$$

such that

$$R\left[\int_0^{t_f} dt \int_0^{r_0} dr_0' \cdots \int_0^{r_n} dr_n' \mathbf{U}\right] \approx C \frac{t_f}{\Delta t} \prod_{i=0}^{n} \frac{r_n}{\Delta r_n}\,\text{s},$$

which for an equilateral integration volume over n dimensions of side length a simplifies to $R \approx C \frac{t_f}{\Delta t} \left(\frac{a}{\Delta a}\right)^n$ s.

For any integration region with dimension $n > 1$, runtimes increase exponentially with the size of the region, causing larger integrals to be quite computationally expensive, especially if done sequentially. Running such a processes in parallel however can greatly reduce the runtime of such an expensive operation. By dividing the job among p processes, in which in takes R_{pre} s to divide the region into subregions, R_{comm} s to send the subregions to each process, and R_{post} s to accumulate the subregion results into an entire state, the runtime can be reduced to

$$R\left[\int_0^{t_f} dt \int_0^{r_0} dr_0' \cdots \int_0^{r_n} dr_n' \mathbf{U}\right]_p \approx \frac{t_f}{\Delta t}\left(R_{\text{pre}} + R_{\text{comm}} + \frac{C}{p}\prod_{i=0}^{n}\frac{r_n}{\Delta r_n} + R_{\text{post}}\right)\text{s}.$$

For processes in which

$$R_{\text{pre}} + R_{\text{comm}} + R_{\text{post}} < C\left(1 - \frac{1}{p}\right)\prod_{i=0}^{n}\frac{r_n}{\Delta r_n},$$

running in parallel is favorable. The speedup of running in parallel can therefore be stated as

$$S(p) = \frac{C\prod_{i=0}^{n}\frac{r_n}{\Delta r_n}}{R_{\text{pre}} + R_{\text{comm}} + \frac{C}{p}\prod_{i=0}^{n}\frac{r_n}{\Delta r_n} + R_{\text{post}}}.$$

2. Method and Assumptions

Now consider only two dimensional discrete surfaces. For such regions, dividing the region into subregions involves sending sub grids to each process, as well as a perimeter boundary for each grid, as shown in Figure 1. To minimize overhead, lets take only equilateral grids and $p \in \mathbb{N}^2$, such that sub grids are square, reducing their perimeter and therefore reducing the number of operations needed to prepare them for processing. In this best case scenario, a region with N total cells requires $N + 4\sqrt{Np}$ to prepare for parallel processing. Thus, $R_{\text{pre}} \approx C_a(N + 4\sqrt{Np})$, for the cost, C_a, of accessing and setting an array element. Furthermore, reconstructing a grid from the results of these processes involves copying the relevant information

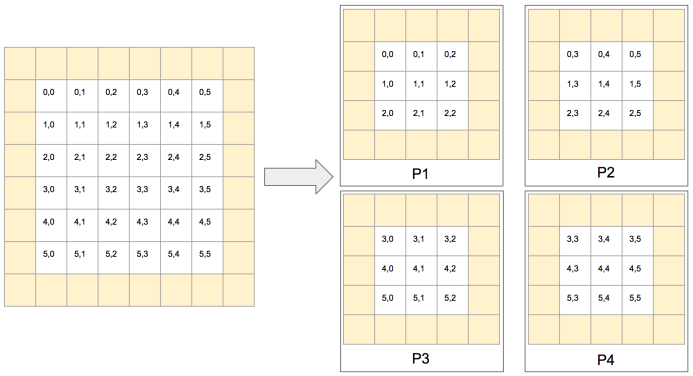

FIGURE 1. To divide work amongst worker processes, the entire grid can be divided into approximately equal subsections, depending on the divisibility of the grid size on p. as shown above, there is overhead incurred in this division, as boundary values must be appended onto each subgrid, shown in yellow. The values placed in these cells are dependent on the boundary conditions, described in Figure 2.

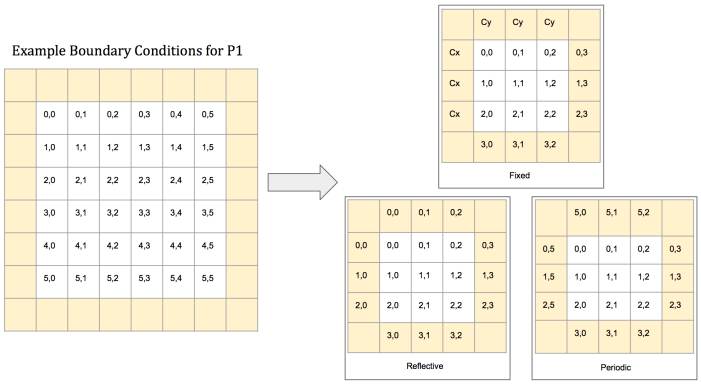

FIGURE 2. There are three types of boundary conditions, which apply only if a sub grid borders the edge of the original grid **U**, otherwise boundaries are constructed from neighboring sub grids. What distinguishes these boundary conditions is what occurs when a sub grid lies on the boundary of **U**. For a fixed boundary, constants can be set to border each edge of **U**. For reflective boundaries, the border of **U** will reflect the value the cells of **U** that lie along it. For period boundaries, the boundaries on each edge of **U** will interact with the cell values on the opposing edge.

from these grids into a new grid, such that $R_{\text{post}} \approx C_a N$. Thus, for two dimensional grids,

$$S(p) = \frac{CN}{R_{\text{comm}} + \frac{CN}{p} + 2C_a(N + 2\sqrt{Np})} \equiv \frac{R_s}{R_p}, \tag{2.1}$$

where R_s is sequential runtime and R_p is parallel processing.

The actual algorthm each process will be running on its own sub grid is

$$\mathbf{U}(x, y, t + \Delta t) = \mathbf{U}(x, y, t) + \alpha \left[\mathbf{U}_{xx}(x, y, t) + \mathbf{U}_{yy}(x, y, t)\right], \tag{2.2}$$

per cell, which is a discrete approximation of Equation 1.1 over two dimensions at point (x, y). As is necessary with any discrete approximation of continuous phenomena, precautions must be taken in choosing the value of any parameter that may cause large step sizes for large $\frac{d}{dt}\mathbf{U}$. For these reasons, the collective effects of Δt and α must be taken into account. While explicit limits for acceptable parameters were not examined, nor were precautions taken in the code to reduce such effects, there do exist criterion for excluding poorly parametrized integrals. By the conservation of energy and the behavior of Equation 1.1, equilibrating isolated thermal systems should never surpass there minimum or maximum thermal values at $t = 0$ at any point within there volume for $t > 0$. However, as shown for a one dimensional system in Figure 3, ill parametrized integrators can violate this behavior. Thus, one criterion used for distinguishing correctly parametrized integrals in that they equilibrate in the limit of $t \to \infty$. Incorrectly parametrized integrals do not, and produce very notably divergent heat maps.

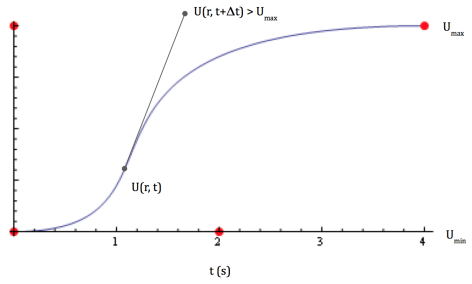

FIGURE 3. Above is an example how overly large step sizes can inaccurately approximate a function. The value for α linearly effects of the size of the approximated derivative of $\mathbf{U}$ at any given point. Therefore is it very important to choose a step size that is sufficiently small enough to avoid overstepping.

3. VALIDATION

The heat maps generated for visualizing the diffusion process scan the state of the system at $t = 0$ to find minimum and maximum values for the

heat of the system within the region. These values are fixed and subsequently used in the mapping over the course of the entire simulation. If an integral was poorly parametrized, the resultant plot would at some point over the course of the simulation exhibit properties like those described in Figure 4. None such integrals were used in any of the subsequent analysis.

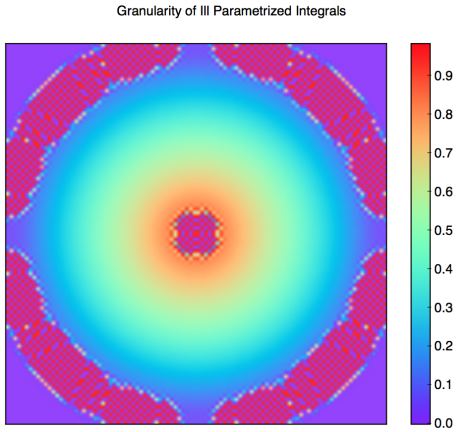

FIGURE 4. By plotting the surface on a heatmap, ill parametrized integrations can be identified by there divergent behavior. Thermal systems such as these should reach equilibrium in the limit of $t \to \infty$. Ill parametrized integrals produce states that diverge from an equilibrium system due to the integrator stepping too aggressively over $\frac{\partial \mathbf{U}}{\partial t}$ to accurately approximate the derivative. These divergent regions are visible in the figure above, represented by the granular areas with temperatures corresponding to temperatures outside the minimum and maximum values at $t = 0$.

For validation of boundary conditions, consider Figure 5, which shows the results of various boundary conditions for an intuitive initial state $\mathbf{U}(\vec{r}) = r$. The boundaries are placed as expected, using edges of other sub grids when possible and defaulting to the boundary condition when a sub grid lies on the edge of the original grid.

Figure 6 compares the results of sequential processing to to the results of running in parallel. As is visible, the results are identical, indicating that the partitioning of data between cores is done effectively and correctly.

4. Results and Discussion

Running in parallel improves the runtime of a simulation step, as shown in Figure 7, and therefore the entire simulation but only for an adequate grid size. Experimentally, $N > 70$ grids produce faster results when run in parallel. The improvement made by running in parallel past this point improves linearly with N.

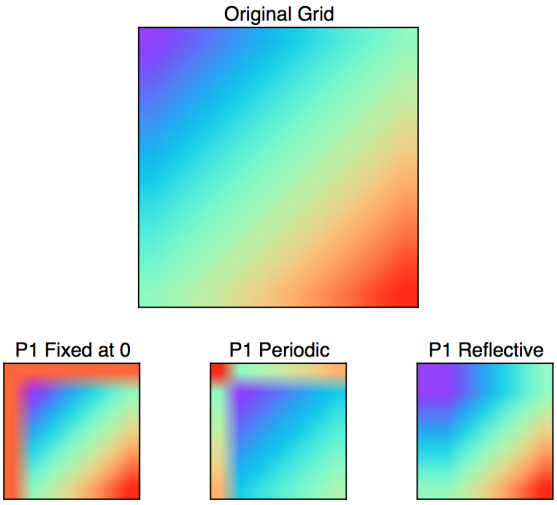

FIGURE 5. Visualized above are the three main types of boundary conditions, as described, as well as the prepared upper left sub grid passed to P1 when the original grid is divided amongst four processes.

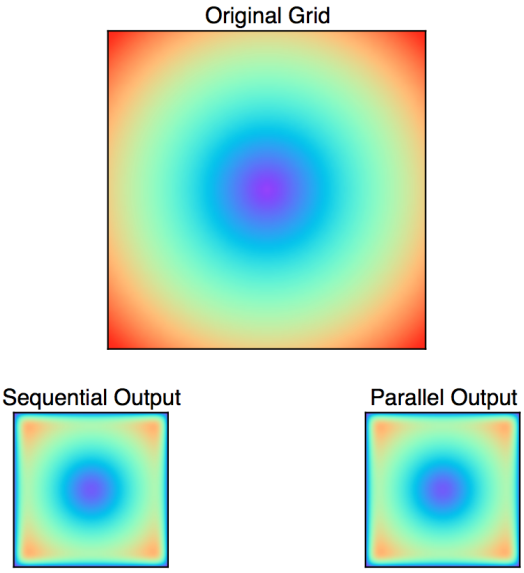

FIGURE 6. Above are the resultant outputs of both parallel and sequential processes for fixed boundaries at 0 and the initial grid $\mathbf{U}(r) = r$.

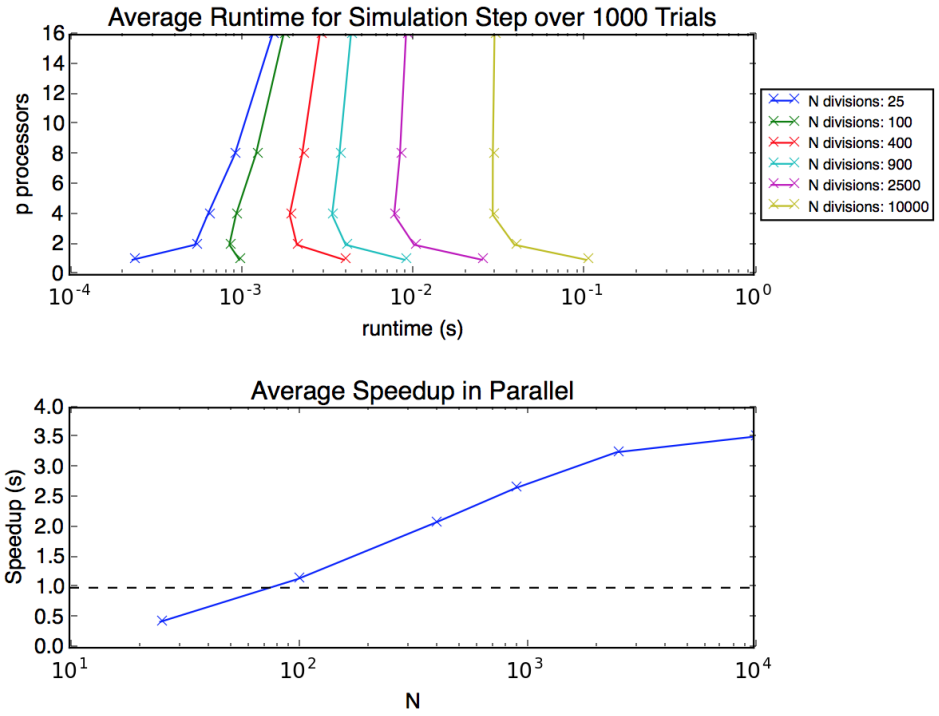

FIGURE 7. Plotted (a) is a comparison of the average runtime for one integration step for various square $\sqrt{N} \times \sqrt{N}$ grids, and various numbers of parallel processes. Notice that for each grid size, there is a minimum runtime for a finite number of processors. Below this number, sequential processing incurs more processing time, while above this number of processors increases runtime as well due to increased communication between processes. Also plotted (b) is the speedup attained by running in parallel, with p_{optimal} processors, over running in sequence. For grids of $N \lesssim 70$, sequential processing is actually faster due to the absence of any intercommunication between processors. As a whole, the speed up appears to be weakly logarithmic.

Assuming that $R_{\text{comm}} \propto \beta p$, for some constant β, the optimal number of processors that reduces the runtime in parallel can be found to be

$$0 = \frac{d}{dp} R_p = \beta - \frac{CN}{p_{\text{optimal}}^2} - 2C_a \sqrt{\frac{N}{p_{\text{optimal}}}}, \tag{4.1}$$

which can be solved numerically for p_{optimal} given known β, C, and C_a. Benchmark tests would have to be run to evaluate these explicitly, which

will not be performed here. While these values remain explicitly unknown, Figure 7.a clearly displays an optimal $p-$value for each plot, as predicted by Equation 4.1.

Similarly, Equation 2.1 can be used to numerically find the cutoff grid size at which parallel outperforms sequential processing at $S(p) > 1$, but again, β, C, and C_a must be known. These constants can vary significantly from machine to machine, as well as for language and libraries used. Regardless, Equation 2.1 is a good starting point when working a machine with known speeds looking to optimize a algorithm involving iterations over two dimensional grid, such as the heat equation. In general, if $N > 70$, parallel processing should be used.

5. Appendix A: Source Code

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from multiprocessing import Pool
from functools import partial
import time


def process_step(d, dx, dt, u):
    """
    Function used by processes during parallel Diffusion
        evaluation.
    This values of d, dx, and dt are passed in automatically,
        equal to
    those of the diffusion instance that calls this method.
    :param d: Diffusion constant. If this value is too high,
        the resolution of
    the integrator will be too small to accurately predict the
        result of diffusion.
    Values that are too large can be very easily spotted
        visually using the animate
    method. During such simulations, the heat map will, over
        time, become granular
    and polarized, containing only maximum or minimum
        temperature values.
    :param dx: the interval of discrete space to use,
        represented by each cell.
    :param dt: the interval of time to use, during the
        simulation step. Similar
    to d, id this parameter is too long, the simulation will
        lose accuracy.
    :param u: The 2D array of temperatures representing the
        material.
    :return: 2D array representing U(t + dt)
    """
    tmp = np.empty((u.shape[0] - 2, u.shape[1] - 2))
    for x in np.arange(1, u.shape[0] - 1):
        for y in np.arange(1, u.shape[1] - 1):
            tmp[x - 1, y - 1] = u[x, y] + dt * d *\
                                (u[x - 1, y] + u[x, y - 1] - 4
                                    * u[x, y] +
                                 u[x + 1, y] + u[x, y + 1]) /
                                     dx**2
    return tmp
```

```
class Diffusion:
    """
    A class used for modelling the diffusion of heat over a 2D
        surface with
    discrete time and discrete space parameters. This class
        comes with built
    in methods for viewing the diffusion as an animation, `
        animate`, as well as methods
    for iterating over a given interval and returning the end
        result, `run`.
    """
    # varying color options for the animation procedure
    blue_to_red = plt.cm.coolwarm
    black_to_copper = plt.cm.copper
    rainbow = plt.cm.rainbow
    white_to_blue = plt.cm.Blues
    black_to_yellow = plt.cm.hot

    def __init__(self, U, numProcesses=1, dt=.1, dx=1.,
        boundary_conditions='horizontal_bar'):
        """
        Initializes the diffusion instance
        :param U: initial 2D nd array describing the heat
            distribution at t=0
        :param numProcesses: the number of processes to use for
             simulation propagation.
        default is 1
        :param dt: the interval of time to use, during the
            simulation step. Similar
        to d, id this parameter is too long, the simulation
            will lose accuracy.
        :param u: The 2D array of temperatures representing the
             material. Can be
        overridden by passing in the optional argument to the `
            run` or `animate` method,
        otherwise they will default to this value
        :param dx: step in discrete space represented by each
            cell in the 2D array.
        defaults to one
        :param boundary_conditions: type of boundary conditions
             to use. can be
        'fixed', 'reflective', 'periodic', 'periodic_horizontal
            ', or 'periodic_vertical'
        :return:
        """
        self.dt = dt
        self.dx = dx
        self.grid = U
```

```
        self.nx, self.ny = U.shape
        self.quad = None
        self.boundary_conditions = boundary_conditions
        if numProcesses <= 1:
            self.pool = None
        else:
            self.numProcesses = numProcesses
            self.pool = Pool(numProcesses)
        self.boundary_pos_y = 0.
        self.boundary_neg_y = 0.
        self.boundary_pos_x = 0.
        self.boundary_neg_x = 0.
        self.D = 1.

    def set_diffusion_constant(self, d=1.):
        """
        Sets the diffusion constant for this instance, to be
            used in all
        subsequent simulations to this call
        :param d: Diffusion constant. If this value is too high
            , the resolution of
        the integrator will be too small to accurately predict
            the result of diffusion.
        Values that are too large can be very easily spotted
            visually using the animate
        method. During such simulations, the heat map will,
            over time, become granular
        and polarized, containing only maximum or minimum
            temperature values.
        :return:
        """
        self.D = float(d)

    def set_fixed_boundary_constants(self, pos_x=0., neg_x=0.,
        pos_y=0., neg_y=0.):
        """
        Sets the new boundary constants for this instance, to
            be used in all
        subsequent simulations to this call, in which 'fixed'
            boundary conditions are used
        :param d: new constant
        :return:
        """
        self.boundary_pos_y = pos_y
        self.boundary_neg_y = neg_y
        self.boundary_pos_x = pos_x
        self.boundary_neg_x = neg_x
```

```
    def get(self, x, y):
        """
        Returns a temperature value of the material at the
            index x,y. Values
        outside the size of the 2D array are accepted and the
            value returned
        by such calls is determined by the type of boundary
            configurations
        set up for this instance.
        :param x: x index
        :param y: y index
        :return:
        """
        u = self.grid
        xmax = self.nx-1
        xmin = 0
        ymax = self.ny-1
        ymin = 0
        if self.boundary_conditions == 'fixed':
            if y > ymax:
                return self.boundary_pos_y
            if y < ymin:
                return self.boundary_neg_y
            if x > xmax:
                return self.boundary_pos_x
            if x < xmin:
                return self.boundary_neg_x
            else:
                return u[x, y]
        if self.boundary_conditions == 'periodic_horizontal':
            if y < ymin:
                return self.boundary_neg_y
            if y > ymax:
                return self.boundary_pos_y
            if x < 0:
                while x < 0:
                    x += xmax
                return u[x, y]
            elif x > xmax:
                while x >= xmax:
                    x -= xmax
                return u[x, y]
            else:
                return u[x, y]
        if self.boundary_conditions == 'periodic_vertical':
            if x < xmin:
                return self.boundary_neg_x
            if x > xmax:
```

```
                return self.boundary_pos_x
            if y < 0:
                while y < 0:
                    y += ymax
                return u[x, y]
            elif y > ymax:
                while y >= ymax:
                    y -= ymax
                return u[x, y]
            else:
                return u[x, y]
        if self.boundary_conditions == 'periodic':
            if y < 0:
                while y < 0:
                    y += ymax
            elif y > ymax:
                while y >= ymax:
                    y -= ymax
            if x < 0:
                while x < 0:
                    x += xmax
                return u[x, y]
            elif x > xmax:
                while x >= xmax:
                    x -= xmax
            return u[x, y]
        if self.boundary_conditions == 'reflective':
            if y < 0:
                y = -y
            elif y > ymax:
                y = 2*ymax - y
            if x < 0:
                x = -x
            elif x > xmax:
                x = 2*xmax - x
            return u[x, y]

    def ddu_x(self, x, y):
        """
        Returns the derivative of U with respect to x at the
            point x,y
        :param x: x index
        :param y: y index
        :return:
        """
        return self.get(x-1, y) - 2.*self.get(x, y) + self.get(
            x+1, y)
```

```
    def ddu_y(self, x, y):
        """
        Returns the derivative of U with respect to y at the
            point x,y
        :param x: x index
        :param y: y index
        :return:
        """
        return self.get(x, y-1) - 2.*self.get(x, y) + self.get(
            x, y+1)

    def divide_grid(self):
        """
        Divides the grid into rectangular sections as close to
            the number of processes
        desired as possible, while minimizing the perimeter of
            each subgrid. This method
        looks to, for n processes, divide the grid into x by y
            subsections such that
        x*y = n and 2x + 2y is minimized. For rectangular
            subgrids, this minimization occurs
        at x = y = sqrt(n), so this method approximates that.
            Unfortunately, in its current
        form it does so in a way that reduces the number of
            processes to an even number.
        Keep that in mind until further iterations of this
            method are implemented.
        :return:
        """
        num = self.numProcesses
        px = int(np.sqrt(num))
        py = int(self.numProcesses*px**-1)
        return px, py

    def sequential_step(self):
        """
        propagates the simulation state using sequential
            processing
        :return:
        """
        u = self.grid
        shape = u.shape
        tmp = np.zeros(shape)
        for i in range(shape[0]):
            for j in range(shape[0]):
                tmp[i, j] = self.get(i, j) + self.dt*self.D*(
                    self.ddu_x(i, j) + self.ddu_y(i, j))/self.
                    dx**2
```

```
        return tmp

    def diffusion_step(self):
        """
        Propagates the simulation state. Runs in parallel or
            sequence depending on the
        configuration of the difussion instance. To run in
            parallel, numProcesses
        needs to be greater then one.
        :return:
        """
        if self.pool is None:
            return self.sequential_step()
        else:
            data = []
            func = partial(process_step, self.D, self.dx, self.
                dt)
            nx, ny = self.nx, self.ny
            px, py = self.divide_grid()
            snx, sny = int(nx/px), int(ny/py)
            remainderx = nx%snx
            remaindery = nx%sny
            # partition into subgrids
            for i in range(px):
                for j in range(py):
                    if i+1 == px:
                        width = snx + remainderx
                    else:
                        width = snx
                    if j+1 == py:
                        height = sny + remaindery
                    else:
                        height = sny
                    subg = np.zeros((width+2, height+2))
                    for m in range(width+2):
                        for n in range(height+2):
                            subg[m, n] = self.get(i*snx+m-1, j*
                                sny+n-1)
                    data.append(subg)
            # 2. divide among workers
            results = self.pool.map(func, data)
            return np.vstack([np.hstack([results[j] for j in
                range(i*py, (i+1)*py)]) for i in range(px)])

    def animate_iterator(self, iter):
        """
        private method that controls animation sequencing
        :param iter:
```

```
        :return:
        """
        self.grid = self.diffusion_step()
        self.quad.set_array(self.grid)
        plt.title("Diffusion t: %.2f" % float(iter*self.dt))
        return self.quad

    def animate(self, tf, dt=None):
        """
        Animates the diffusion of the 2D surface
        :param tf: final time of simulation
        :param dt: time step
        :return:
        """
        self.quad = plt.imshow(self.grid, cmap=Diffusion.
            rainbow)
        if dt is None:
            step = self.dt
        else:
            self.dt = step = dt
        plt.colorbar()
        plt.title("Diffusion t: %.2f" % 0)
        self.anim = animation.FuncAnimation(plt.gcf(), self.
            animate_iterator, frames=int(tf/step)+1, interval
            =5, blit=False, repeat=False)
        plt.gca().get_xaxis().set_visible(False)
        plt.gca().get_yaxis().set_visible(False)
        plt.show()

    def run(self, t, dt=None):
        """
        Simulates without plotting any visuals.
        :param t: can be either an nd array of a scalar. If t
            is a scalar,
        t will be interpreted as the time at which to step
            simulating,
        stepping through at interval dt. If t is an array, the
            simulation
        will step through each each point in time in the array,
             using
        dt = t[i] - t[i-1] as the timestep.
        :param dt:
        :return:
        """
        if type(t) is int or type(t) is float:
            if dt is None:
                step = self.dt
            else:
```

```
                step = dt
            for i in np.arange(0, t, step):
                self.grid = self.grid = self.diffusion_step()
        if isinstance(t, np.ndarray):
            for i in range(1, t.size):
                self.dt = t[i] - t[i-1]
                self.grid = self.grid = self.diffusion_step()
        return self.grid

    @staticmethod
    def view_static(grid):
        """
        Plots the state of the grid as a heat map to visualize
            the state of the system.
        :param grid: the state to be plotted.
        :return:
        """
        plt.imshow(grid, cmap=Diffusion.rainbow)
        plt.colorbar()
        plt.gca().get_xaxis().set_visible(False)
        plt.gca().get_yaxis().set_visible(False)
        plt.show()

'''
# Sequential example
diff = Diffusion(U, boundary_conditions='reflective')
diff.set_diffusion_constant(.25)
t0 = time.time()
diff.diffusion_step()
print "Sequential", time.time() - t0

# Parallel example
diff = Diffusion(U, 4, boundary_conditions='reflective')
diff.set_diffusion_constant(.25)
t0 = time.time()
diff.diffusion_step()
print "Parallel", time.time() - t0

# animate example
a = 10
da = 100
y, x = np.meshgrid(np.linspace(-a, a, da), np.linspace(-a, a,
    da))
U = np.sin(x**2+y**2)/5
U[U < 0] = 0.
diff = Diffusion(U, 4, boundary_conditions='reflective')
diff.set_diffusion_constant(.15)
diff.animate(10, .05)
```

```
# run example
a = 10
da = 100
y, x = np.meshgrid(np.linspace(-a, a, da), np.linspace(-a, a,
    da))
U = np.sin(x**2+y**2)/5
U[U < 0] = 0.
diff = Diffusion(U, boundary_conditions='reflective')
diff.set_diffusion_constant(.25)
end_result = diff.run(1, .05)
diff.view_static(end_result)
'''
```