Numerical Solutions of the Fokker-Plank Equation

John A. Vahedi

April 4, 2020

Abstract

An introduction to Stochastic Partial Differential Equations "SPDEs" via the Fokker-Plank Equation. We model the problem around a deterministic physical problem and then introduce a stochastic driver. We use a couple methods to solve the constant coefficient problem in a 1 dimensional domain. Finally we look at the Black-Schole's equation and solve this position dependent problem with splitting methods. Included are relevant checks for stability and accuracy. We also address a simple parallelization of one of our methods and check increases in performance.

1 Introduction

Let us begin by explaining what is meant by "stochastic." While a regular variable holds a quantity that arises deterministically, a stochastic variable allows for a quantity that arises through randomness. For example, a particle's horizontal coordinate can be referred to as x. When given an initial condition for its position and some deterministic process, the quantity x may be able to be determined at future points in time. Yet even if one is given an initial position for particle but the process is not seen as deterministic, then the variable that holds the future position of such a particle is not deterministic but random, to an extent, in nature. Such a random variable, or stochastic variable, is usually denoted by a capital X.

What is an example of a process that is stochastic? Say we are looking at a particle in box that has an ideal fluid inside. Now if I knew the position and velocity vectors of all the other particles in the box, then process for the future position of my particle of interest is deterministic. Yet this an impossibly daunting task for any realistic case as the number particles in even a drop of water is something on the order of 10^{21}. Instead one may take a different statistical approach such as Einstein did when he dealt with this problem. Here the force acting on the particle through bombardments in magnitude, direction and frequency are considered to be random. Therefore the position of the particle becomes a probability distribution $p(x,t)$ over space and time where at the original time we have $p(x,0) = \delta(x - x_s)$, the delta distribution centered around

the original position. One can also describe several particles or a concentration of particles in a similar way as a (probability) distribution at the original time. We can call this function $F(x)$ where instead $p(x,0) = F(x)$. Such a probability distribution devolves in time, where each deterministic x has a probability of being the value of the stochastic variable, $X = x$. Remember that both variables x and X change in time, are functions of time, and are sometimes denote them as $x(t)$ and X_t.

A stochastic process can be described by a stochastic differential equation. Such a differential process can always be separated into a completely deterministic part and a purely stochastic part. Therefore, a first order stochastic differential can be written as $\frac{dX}{dt} = g(X,t) + h(X,t)\eta(t)$ where $g(X,t)$ is the deterministic part and $h(X,t)$ is the coefficient of a stochastic "white noise" part. The white noise $\eta(t)$ is purely random as opposed to colored noise, due to the separation of the deterministic part. If $h(X,t) = 0$ then we have an ordinary differential equation. The stochastic $\eta(t)$ is a Gaussian with mean $E[\eta(t)] = 0$ and covariance $E[\eta(t)\eta(s)] = \delta(t-s)$ and therefore $\eta \sim N(0,1)$. We rewrite the differential as [5]:

$$dX = \mu(X,t)dt + \sigma(X,t)dW. \tag{1}$$

Here a few changes have been made. The $g(X,t)$ has been replaced with $\mu(X,t)$ which represents the mean path of the stochastic process which also is the path of the deterministic ODE. Also the amplifying coefficient of the stochastic white noise is appropriately renamed as $\sigma(X,t)$ which for a Gaussian is completely described by its first two modes. Lastly, we can understand $\eta(t)t$ as $N(0,t^2)$ which is known as a Brownian or Wiener Process. Therefore the term $\eta(t)dt$ has been replaced with dW which is a differential in the Wiener random variable. Ultimately the differential equation is stating that our random variable X changes a little bit due to time changing a bit and also due to the random white noise that has been given a little bit of time to evolve. Each such part's contributions are controlled by the the governing functions μ and σ.

1.1 Stochastic Calculus

The solution the to our stochastic differential equation will require integration. Namely, given an initial condition or distribution X_0:

$$X_t = X_0 + \int_0^t \mu(X_s, s)ds + \int_0^t \sigma(X_s, s)dW_s. \tag{2}$$

Here I have given a subscript to random variables to note they are sensitive to an integration over time. This is no normal integral as it has stochastic terms and differentials. Therefore we further explore how to deal with the stochastic integration of $S = \int_0^T \sigma(X_t, t)dW_t$. We will assume that contributions from this term come at discrete points $t_i, i \in (1, 2, 3, ..., N-1)$, and cause rough jumps in the value. Therefore as a Riemann Sum can be rewritten as:

$$S = \lim_{N\to\infty} \sum_{i=1}^{N} \sigma(X_{t_{i-1}}, t_{i-1})(W_{t_i} - W_{t_{i-1}}). \tag{3}$$

This limiting form is known as Ito's Integral as it uses the left endpoint of the subinterval to evaluate the integrand and is not aware of the future values. There are other kinds that involve the right endpoint, or a midpoint type rule known as a Stratonovich Integral. As we will see, this choice is not trivial as the limiting case behaves differently than in classical calculus.

As an example let us look at

$$S = \int_0^T W_t dW_t = \lim_{N\to\infty} \sum_{i=1}^{N} W(t_{i-1})(W(t_i) - W(t_{i-1}))$$

$$= \lim_{N\to\infty} [\frac{1}{2}\sum_{i=1}^{N}(W^2(t_i) - W^2(t_{i-1})) - \frac{1}{2}\sum_{i=1}^{N}(W(t_i) - W(t_{i-1}))^2]$$

The first term is simply $\frac{1}{2}W^2(T)$. Within the second term $W(t_i) - W(t_{i-1})$, in the limit, would be expressed as the differential $(dW(t))^2$ which is just dt as it is a Brownian Process. Therefore the second term is $-\frac{1}{2}\int_0^T dt = -\frac{1}{2}T$. Finally we have:

$$S = \frac{1}{2}W_T^2 - \frac{1}{2}T.$$

The first term can be thought of what we might find in classical calculus but we have an extra term in Ito's calculus.

In general these types of stochastic integrals are difficult or tedious to solve. Instead we will take a different approach in solving our stochastic problem.

1.2 Ito's Lemma

We once again begin with our original differential, $dX = \mu(X,t)dt + \sigma(X,t)dW$. Now say you have a function $Y(X_t, t)$ which is another arbitrary process that is a function of the original process. Using a Taylor Expansion to get a differential in Y we get:

$$dY = \frac{\partial Y}{\partial t}dt + \frac{\partial Y}{\partial X}dX + \frac{1}{2}\frac{\partial^2 Y}{\partial X^2}dX^2 + \cdots$$

where we have not included second order terms except one which has a linear term as we will see. dX^2 has many square terms that will be dropped except the $\sigma^2(X,t)dW^2$ which is linear as $\sigma^2 dt$. Replacing dX and dX^2 we have:

$$dY = \frac{\partial Y}{\partial t}dt + \frac{\partial Y}{\partial X}(\mu dt + \sigma dW) + \frac{1}{2}\frac{\partial^2 Y}{\partial X^2}\sigma^2 dt.$$

Factoring we get Ito's Lemma:

$$dY = (\frac{\partial Y}{\partial t} + \mu\frac{\partial Y}{\partial X} + \frac{\sigma^2}{2}\frac{\partial^2 Y}{\partial X^2})dt + (\sigma\frac{\partial Y}{\partial X})dW \tag{4}$$

which actually is a similar stochastic process to our original differential setup. Here the first coefficient describes the mean path or the "drift" of the process and the second has to do with the randomness or "diffusion" of the process.

1.3 Stochastic Differential Equation

For an arbitrary function of this process that does not depend on time directly $Y = f(X_t)$, using Ito's Lemma we have [6]:

$$df = (\mu f_x m + \frac{\sigma^2}{2} f_{xx})dt + (f_x \sigma)dW.$$

Here I use the subscript to denote a partial derivative and use lower case x for simplicity. Next we take an expectation to remove the stochastic term

$$E[df] = E[(\mu f_x + \frac{\sigma^2}{2} f_{xx})dt + (f_x \sigma)dW] = E[\mu f_x + \frac{\sigma^2}{2} f_{xx}]dt$$

as the $E[dW]$,expectation of a Brownian, is 0. Yet instead we will see that the stochasticity will be preserved as the expectation introduces the all encompassing probability function. Using the integral definition of the expectation and passing an integral into the derivative by Leibniz's Rule :

$$\frac{d}{dt}\int_{-\infty}^{\infty} f(x)p(x,t)dx = \int_{-\infty}^{\infty} \mu(x,t)f_x(x)p(x,t)dx + \frac{1}{2}\int_{-\infty}^{\infty} \sigma^2(x,t)f_{xx}(x)p(x,t)dx. \tag{5}$$

Using integration by parts, the first term on the RHS is equivalent to

$$f(x)\mu(x,t)p(x,t)|_{x=-\infty}^{x=\infty} - \int_{-\infty}^{\infty} f(x)\frac{\partial}{\partial x}(\mu(x,t)p(x,t))dx$$

$$= -\int_{-\infty}^{\infty} f(x)\frac{\partial}{\partial x}(\mu(x,t)p(x,t))dx.$$

The first term is here is zero because the probability function must vanishes trending towards infinity. Similarly the second term from the RHS of the original integral equation, upon using two applications of integration by parts, is equivalent to

$$\frac{1}{2}\int_{-\infty}^{\infty} f(x)\frac{\partial^2}{\partial x^2}(\sigma^2(x,t)p(x,t))dx.$$

Putting all of it together we have:

$$\frac{d}{dt}\int_{-\infty}^{\infty} f(x)p(x,t)dx$$

$$= \frac{1}{2}\int_{-\infty}^{\infty} f(x)\frac{\partial^2}{\partial x^2}(\sigma^2(x,t)p(x,t))dx - \int_{-\infty}^{\infty} f(x)\frac{\partial}{\partial x}(\mu(x,t)p(x,t))dx.$$

Moving terms around and combing the integrals:

$$\int_{-\infty}^{\infty} f(x)\left(\frac{\partial}{\partial t}p(x,t) + \frac{\partial}{\partial x}(\mu(x,t)p(x,t)) - \frac{1}{2}\frac{\partial^2}{\partial x^2}(\sigma^2(x,t)p(x,t))\right)dx = 0$$

Let $D(x) = \frac{\partial}{\partial t}p(x,t) + \frac{\partial}{\partial x}(\mu(x,t)p(x,t)) - \frac{\partial^2}{\partial x^2}(\sigma^2(x,t)p(x,t))$ Note that our integral is equal to 0. Since f(x) was chosen arbitrarily, you can use bump functions to show that the right term in product of the integrand must be zero everywhere. The "Bump Theorem" [9] works by letting a small unitary function that is zero everywhere except at a narrow region be $f(x)$. Now since the function is arbitrarily narrow we can approximate the $D(x,t)$ as a constant and then the integral is just this constant times the area under the bump. The area, by construction, is positive and therefore $D(x,t)$ must be zero in this region. Since we could have chosen the bump to be anywhere, $D(x,t)$ must be zero everywhere, at all locations. Therefore we are left with the partial differential equation:

$$\frac{\partial}{\partial t}p(x,t) + \frac{\partial}{\partial x}(\mu(x,t)p(x,t)) - \frac{1}{2}\frac{\partial^2}{\partial x^2}(\sigma^2(x,t)p(x,t)) = 0 \tag{6}$$

Here we are interested in solving for $p(x,t)$ which determines the distribution of the particle, in the first place. If $\mu(x,t) = 0$ then we have an in place familiar heat diffusion equation with a weighted diffusion that is space and time dependent. If instead $\sigma(x,t) = 0$ then we have something akin to the transport equation which would have been form preserving of the initial condition and moved it at some speed. Yet because of the time and specifically the spatial dependence of μ we have instead an advection equation that can result in "shocks" and "rarefactions" in the solution. The difference between these two forms is that if I knew the exact position of my original particle in my initial condition the latter would be deterministic while the former would introduce stochastic uncertainty in time. Assuming all terms are non-zero we have an advection-diffusion equation that both is transports and diffuses naturally. This general partial differential equation is known as the Fokker-Plank Equation and should well represent the evolution of a stochastic process.

2 Numerical Methods

We will look here at different types of important problems and numerical methods for their solutions. We will include an analysis on their stability and accuracy. We will stick to 1-D problems as was established in the theory from the introduction.

2.1 Constant Diffusion

We begin with the simplest problem in the Fokker-Plank set of equations. We will set $\sigma(x,t) = 1$ and $\mu(x,t) = 0$. This is the well known diffusion problem. Since we suppose we know the initial value of say a stock to be exact, we have the the initial distribution: $p(x,0) = \delta(x - x_s)$. For simplicity we will shift the axis by x_0 so that the initial value is zero and the **true zero** is a negative number at $-x_s$. Also assume that x_s is a large number so that we do

not evolve into that region $x < -x_s$ with a high likelihood during our simulation time. Additionally we have homogeneous Dirichlet boundary conditions, namely: $p(-\infty, t) = p(\infty, t) = 0$. The exact solution is known [3]:

$$p(x,t) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}}. \tag{7}$$

Since we will use Finite Difference Methods there is a few technical issues we must tackle first. First, our initial distribution the delta function has infinite height and infinitesimal width whose product must is one. Instead we use the true solution to devolve the initial distribution for a sufficiently small time to $p(x, .4)$. Our simulation time will begin from $t = .4$ and run to $t = 5.4$. For an unknown problem, a better method must be explored. Second, since our simulation must be calculated on a finite domain we can not include the infinite boundary domain. Instead we set the Dirichlet Boundary sufficiently far away from the origin at $p(-20, t) = 0$ and $p(20, t)$. Due to the error caused by this superficial boundary we will discard 25 percent of the domain at the edges and only retain the central 75 percent about the origin.

2.1.1 Forward Euler

The first method we use is the one step Forward Euler discretization. First order in time and second order centered in space, this explicit method is dubbed FTCS. If we spatial dicretize our domain into n parts, let: $x_0 = -20, x_1 = x_0 + \Delta x, ..., x_j = x_0 + j\Delta x, ..., x_n = 20$. Similarly for the temporal discretization, let: $t_0 = 0, t_1 = t_0 + \Delta t, ..., t_i = t_0 + i\Delta t, ...$. Define $P_i^j = p(x_j, t_i)$. This explicit method uses the update:

$$P_{i+1}^j = P_i^j + \frac{\sigma^2 \Delta t}{2\Delta x^2}(P_i^{j-1} - 2P_i^j + P_i^{j+1}) \tag{8}$$

The boundaries are trivially handled as continual updates as zero at each iteration. To maintain stability of the solution [7], we need $\Delta t \leq \frac{\Delta x^2}{\sigma^2}$. In the supplied code I use the largest time step possible given a spatial discretization. If I violated this condition by taking too large of a time step my solution would begin to oscillate uncontrollably after a sufficient number of steps. Otherwise, the solution worked well and converged squared to the spatial discretization.

2.1.2 Crank-Nicolson

In contrast to the former method, Crank-Nicholson is partially implicit method. This method is unconditionally stable and allows for much greater time steps. Still this may come at a cost to accuracy of our solution. The implicit update scheme goes as follows:

$$P_{i+1}^j = P_i^j + \frac{\Delta t}{2\Delta x^2}(P_{i+1}^{j-1} - 2P_{i+1}^j + P_{i+1}^{j+1}) \tag{9}$$

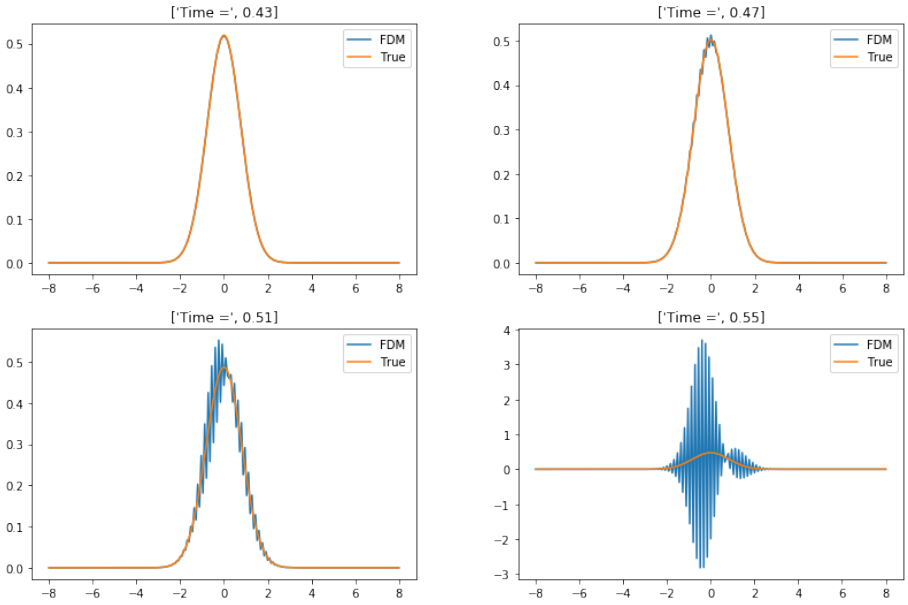

Figure 1: Oscillations for Large Time Steps [FE]

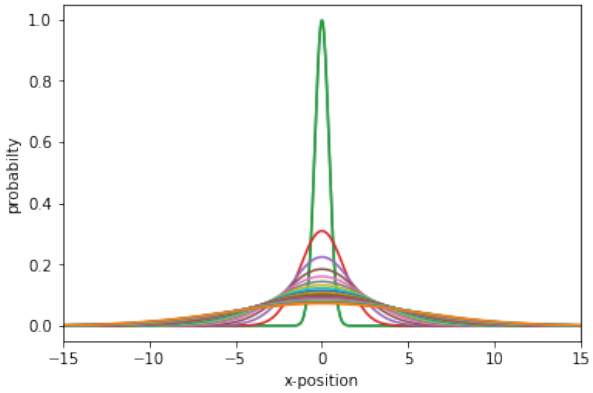

Figure 2: Evolution of Probability in Time [FE]

Since the update scheme on the right includes information about the future, the linear system of all the points must be solved simultaneously. Letting $R = \frac{\Delta t}{2\Delta x^2}$, the relevant matrix, A is tridiagonal with $[R, 1-2R, R]$ across, as $P_i = AP_{i+1}$. The boundary conditions can be handled by reducing the size of the matrix and solution vectors by two, those being the boundary values. You can re-update the those as zero after solving for $P_{i+1} = A^{-1}P_i$. This method converged second order in its spatial discretization as well, yet for the same iterations had a slightly *less* accurate solution. This may be due to how the implicit method may increase the stability domain but reduces it's the accuracy.

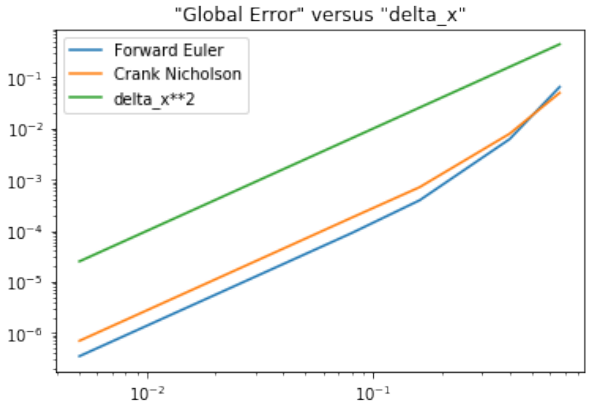

Figure 3: Global Error Convergence

2.2 Constant Diffusion and Drift

Now we include non-zero, yet constant, μ term. If the diffusion term is small and or zero then the equation is just the transport equation that transports the initial distribution at some constant speed $c = \mu$. For FTCS method, we can use Von Neumann analysis to show that [10] this would be always unstable.We look instead to the Upwind Method which is first order in both time and space.The update formula is:

$$P_{i+1}^j = P_i^j - \frac{\mu \Delta t}{\Delta x}(P_i^{j+1} - P_i^j) \tag{10}$$

Where for stability we require [10] $\Delta t \leq \frac{\Delta x}{|\mu|}$. Otherwise information would be traveling faster than the drift speed. We can also use a similar method, Lax-Friedrichs; its discretization leads to:

$$P_{i+1}^j = \frac{1}{2}(P_i^{j+1} + P_i^j) - \frac{\mu \Delta t}{\Delta x}(P_i^{j+1} - P_i^j) \tag{11}$$

where we have taken the average of the spatial values instead and results in the same stability condition. This method tends to have no directional problems

but does have lower accuracy since it has some unwanted diffusive term in its modified equation. [1] even more so than the original upwind method. A correction may be possible for this using a higher derivative.[???]

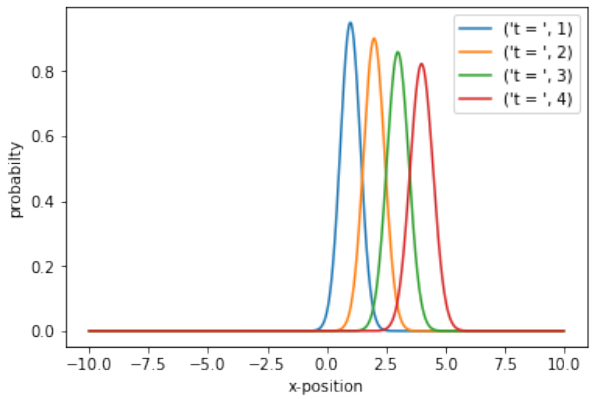

Figure 4: Note Diffusion in Pure Upwind Method in (a) $\Delta x = I/1000$

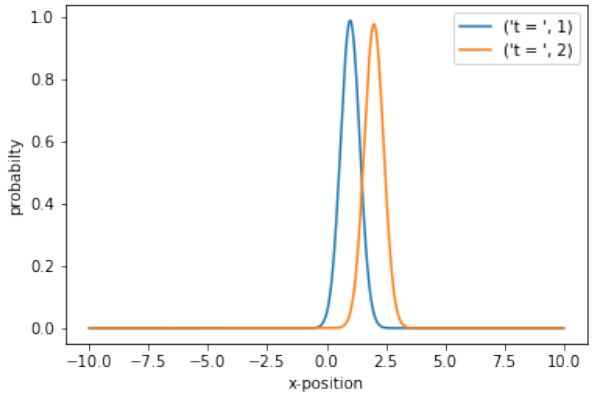

Figure 5: (b)$\Delta x = I/5000$

Now, we need to choose a method that works for both the diffusion and advection. Our diffusion methods did not work for the advection part. The advection type method is not possible for the diffusion term because of its order, unless we go to a higher odd term. In order to keep the overall method simple will use something else called a splitting method.

2.2.1 Godunov Splitting

Let us call the induced matrix arising from the discretized diffusion term independently A, where $Ap = cp_{xx}$, $c = \frac{\sigma^2}{2}$. Similarly we will call the induced matrix arising from only the advection term B, where $Bp = dp_x$, $d = -\mu$. Here the theoretical equation for the discretization would have been $p_{i+1} = (A+B)p_i$. The splitting method applies each of these in sequence as opposed to together, namely $ABp_i = p_{i+1}$. It can be shown [12] that this is exact if the commutation $[A, B] = AB - BA = 0$. Otherwise the method is first order accurate.The two methods used will be FTCS for the diffusion and Upwind method for the advection. Looking at the actual matrices for these discretization schemes, we find that $AB - BA$ is almost zero everywhere except for two terms at the one at each end of the diagonal (this is true for L-W as well).

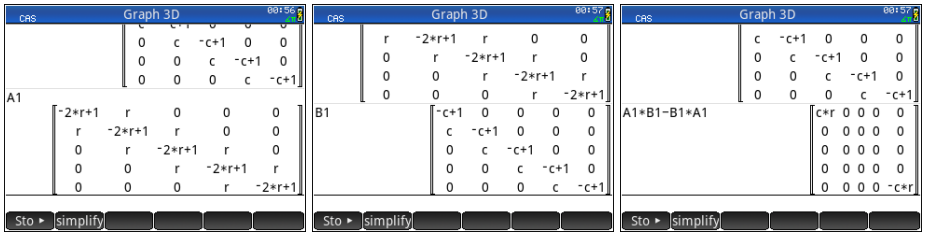

Figure 6: Commutator Example with FTCS 'A1' and Upwind 'B1'

This likely arises due to how part of the discretization is cut off at the boundaries. Alternatively, one can show that $ABp = c(dp_x)_{xx} = cdp_{xxx} = BAp$.

The true solution to the diffusion advection problem given our regular initial delta distribution, $\sigma(x,t) = 1$, and $\mu(x,t) = \mu$ (a constant) :

$$p(x,t) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{(x-\mu t)^2}{2t}}. \tag{12}$$

This is just simple transformation in one direction with speed μ in time. Our splitting method uses $\mu = .75$. A precursory eyeball norm of the graph looks good at different times in its evolution. It does dip at the tip somewhat which attribute to the upwind method's unwanted diffusivity. Seemingly the solution convergence first order in Δx.

2.3 Geometric Brownian Motion

The final problem that we will attempt to solve had variable values for the drift and diffusion functions. Usually these are $\sigma(x,t) = \sigma x$ and $\mu(x,t) = \mu x$, where both are linear only in x and not time dependent. This is known as Geometric Brownian Motion as the deterministic case has a solution that is geometric or exponential. A common model for stocks is to would grow exponentially. Yet in this model the variance or stochasticity also has this property that it is exaggerated on its current value. We could instead have a

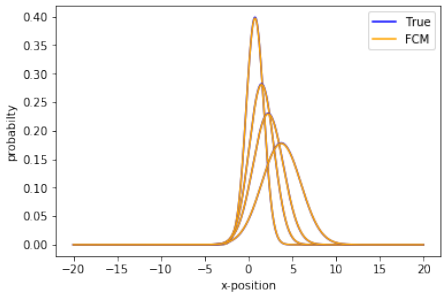

Figure 7: Splitting Method for times $t = 1, 2, 3, 5$ and $\Delta x = I/3000$

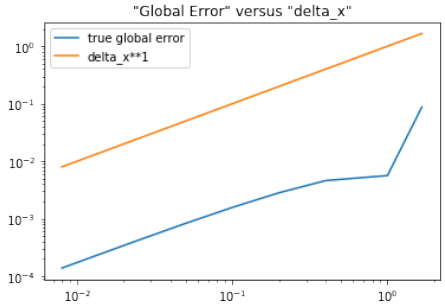

Figure 8: Global Error Convergence

process that is linear in x for its drift but constant for its diffusion. This would more closely resemble Ornstein–Uhlenbeck equation, although it has a negative coefficient for its drift and therefore decays instead of growing.

2.3.1 Strang Splitting

Here because the matrices likely no longer satisfy the nice commutation property from the constant example before, so we have to be more careful to choose a good method. Luckily there is a second order [12] type of the splitting method called *Strang Splitting*. Very similar to the previous method, we still alternate between each method except we split the first step into two half time steps and wrap the sequence front and back with them instead. For this we can use the subscript on a matrix to denote the time step. Previously we had, $(A_{\Delta t}B_{\Delta t})p_i = p_{i+1}$. Now we will have, $(A_{\Delta t/2}B_{\Delta t}A_{\Delta t/2})p_i = p_{i+1}$. We also have to watch out for the stability condition as the diffusivity is not longer a constant and may be violated as the x our stock price grows.

2.3.2 Home-brew Method

Another subtle issue is how to handle the PDE. Since we now have non-constants, we can not pull them out of the derivatives. It is best not to use the product rule to expand them [7] but rather discretize the product in the derivatives and make a finite difference of this product. For example we have:

$$\frac{\partial^2(x^2p(x,t))}{\partial x^2} = 2p + 4x\frac{\partial p}{\partial x} + x^2\frac{\partial^2 p}{\partial x^2}$$

while if you you discretize first (and use Δx to relate adjacent points) and then use the difference methods you get something very similar:

$$(P^{j+1} + P^{j-1}) + 4x^j\frac{P^{j+1} - P^{j-1}}{\Delta x} + (x^j)^2\frac{P^{j+1} - 2P^j + P^{j-1}}{\Delta x^2}.$$

Interestingly, this seems like a second order finite difference except when it comes to the first term. Similarity we have:

$$\frac{\partial(xp(x,t))}{\partial x} = p + x\frac{\partial p}{\partial x}$$

but here the discretization is the same for upwind:

$$P^{j-1} + x^j\frac{P^j - P^{j-1}}{\Delta x}.$$

Letting $s = \sigma^2/2$ and putting everything together we get:

$$\frac{\partial p}{\partial t} = -\mu\frac{\partial}{\partial x}(xp(x,t)) + s\frac{\partial^2}{\partial x^2}(x^2p(x,t))$$
$$-\mu\left(P^{j-1} + x^j\frac{P^j - P^{j-1}}{\Delta x}\right) +$$

$$s\left((P^{j+1}+P^{j-1})+4x^j\frac{P^{j+1}-P^{j-1}}{\Delta x}+(x^j)^2\frac{P^{j+1}-2P^j+P^{j-1}}{\Delta x^2}\right)$$

$$=(sP^{j+1}+(s-\mu)P^{j-1})+\frac{x^j}{\Delta x}(4sP^{j+1}-\mu P^j+(\mu-4s)P^{j-1})+s\frac{(x^j)^2}{\Delta x^2}(P^{j+1}-2P^j+P^{j-1})$$

Clearly we have broken out of our well known schemes and no longer can definitively do the splitting intelligently. On top of that in the past we referred to the true zero at some negative value. We will have to change that now back to simplify the problem. Therefore our new discretization will be $x_j = j\Delta x$. Finally we have the explicit update scheme (the RHS will have all subscript i):

$$P_{i+1}^j = P_i^j + \Delta t(sP^{j+1}+(s-\mu)P^{j-1})+$$
$$\frac{j\Delta t}{\Delta x}(4sP^{j+1}-\mu P^j+(\mu-4s)P^{j-1})+s\frac{j^2\Delta t}{\Delta x^2}(P^{j+1}-2P^j+P^{j-1}) \quad (13)$$

We could write this differently as one scheme where:

$$P_{i+1}^j = \alpha_j P^{j-1}+\beta_j P^j+\gamma_j P^{j+1} \quad (14)$$

$$\alpha_j = \Delta t(s-\mu)+\frac{j\Delta t}{\Delta x}(\mu-4s)+s\frac{j^2\Delta t}{\Delta x^2}$$
$$\beta_j = 1-\mu\frac{j\Delta t}{\Delta x}-2s\frac{j^2\Delta t}{\Delta x^2}$$
$$\gamma_j = \Delta t(s)+\frac{j\Delta t}{\Delta x}(4s)+s\frac{j^2\Delta t}{\Delta x^2}$$

The method is highly unstable and quickly finds oscillations in this scheme. A different approach must be found to dicretization or a different implementation. In theory an implicit method may be a good idea due to the instability. The matrix sparse so inverses should not be computationally expensive [Figure 9]. (There is the off chance I made an algebra mistake).

3 Conclusion

The different methods and types of sub-classes to the advection diffusion highlight what are some difficulties one might come across and should be aware of. As an alternative I may have approached this problem through monte-carlo simulation at least in comparison, but this may (?) have been computationally expensive to generate all that data.

There are some other ways one could extend this project and some questions to answer.(1) We showed convergence in the spatial discretization and maintained the minimum temporal one for stability. One could decouple them and check for convergence in each.(2) Another possibly computational expensive task was that I had to extend my domain way beyond the "compactness" of my

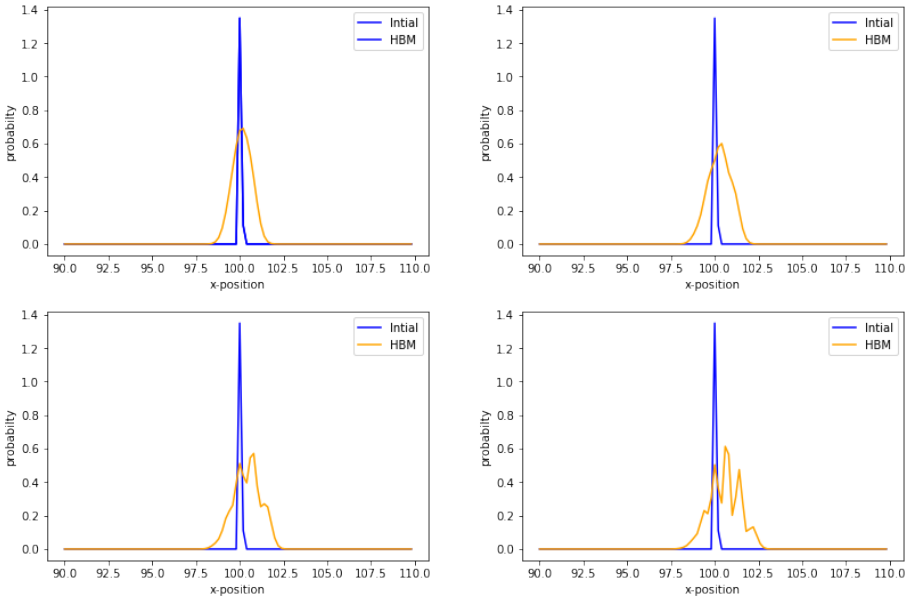

Figure 9: Rapidly Growing Oscillations for t = ,005,.01,.015,.02 [HBM] with $\sigma = .1, \mu = .5$ and, $\Delta x = 20/100$

distribution or what i was interested in and threw away the edges at the end for the error. Instead I found something called "transparent boundary conditions" which mimic a psudo- boundary. Yet, this likely would be analytically difficult because the diffusion term gets information from the entire domain instantly. If we are ignoring everything past the boundary, the boundary will have to make up for all that information. (3) Yet another element to check with our methods was how well they conserved probability. We could have used a trapezoidal method to see how the global probability was shifting. Additionally we could have employed a recovery effort that would have possibly redistributed the lost probability intelligently. (4) The initial condition in this paper was handled trivially by simulating the real solution for a small time. Instead we could use a sequence of function that approximate the delta function and see which is best. Additionally, we could see what type of error they would contribute.(5) A small piece we could have added would be a correction term for the upwind method using a higher derivative trick by calculating for it error in derivatives explicitly removing it. (5) Computationally when the probability approached the true zero we should have removed those possibilities as stock value does not bounce back from zero and moreover can not be negative. This would be similar to the gambler's ruin problem.

4 Parallelization

Previously, our iterations were performed on one core to attain our results. In this section I attempt to breakdown a couple of the previous problems into pieces that can be run in parallel over multiple different cores simultaneously.

In larger problems or if your problem requires a high level of refinement or accuracy, this is a necessity as one core can only do so much. Parallelization is the obvious next step. The speed in producing our calculations may be increased by factors, as we may only be using a fraction of our computational power available. These theoretical speedups are bounded by Amdahl's Law.

4.1 Amdahl's Law

While these problems can, in different ways, be broken down into parallel pieces, only a fraction of the problem can be parallelized. In other words some part of the problem will have to remain processed in series and this will put an upper-bound to our speedup.

For example, if I run my problem over two cores you might expect .5 times the run time. Yet, if we imagine that about 10 percent of the problem has to be executed in series then only 90 percent of the run-time is available to be halved through parallelization. Therefore in this case the new run time over two cores would be .55 times the original. No matter how many cores I throw at the problem I would only get closer and closer to the .10 times the run-time, instead of closer to zero. If we note that speedup is just one over the new percentage of run time (or old run-time over new run-time) then this theoretical speedup s is captured by Amdahl's Law [4]:

$$s = \frac{1}{(1-p) + \frac{p}{c}} \tag{15}$$

where p is the parallelizable fraction and c is the number of cores.

Yet, we are better served by referring to Amdahl's Law as a bound. One relevant complication in our case will be that the execution size will not remain static when parallelizing our code. There are increases in required amount of executions to communicate between cores if we split up the problem in these examples. If the overhead outweighs the benefit of the parallelization then this may not be the way to go forward. If there is a benefit when switching to two cores from one, I will assume there will continue to be a benefit with higher cores. This is a linearity assumption, meaning: that that the overhead grows linearly with the number of cores and the is no non-linear interaction inside this phenomenon. Such edits to the law can be described as:

$$s_{edit} = \frac{1}{(1-p) + \frac{p(1+k_1 c + k_2 c^2)}{c}}. \tag{16}$$

Here we have a added a piece that increases the necessary execution time for each core. The coefficients k_1 describes the ratio, as compared to the size of

the parallel execution, that is additionally required to parallelize. As we will see in our case, each core has to only communicate with its left and right domain neighbors not with every other core. Therefore while we have a linear increase in the load we also have the same amount of core obviously to handle it, so it just results in a constant addition to the run-time. Still, there is a *gather* command that has to interact with all the other cores. Since only one core has to execute this you might think it would be linear still. This is misleading because all the other cores are idle during this time which means we could in practice consider all the cores as executing this piece. Another way to think about it is as part of the series section of the code.

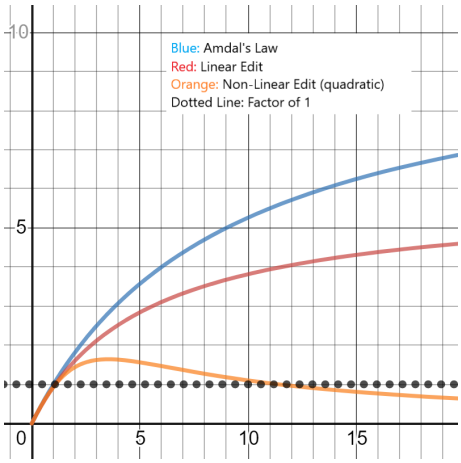

Figure 10: Speedup versus Cores
$p = .9$, (Linear) $k_1 = .08, k_2 = 0$; (Non-Linear) $k_1 = 0, k_2 = .08$

Looking at the sample graph [Figure 10] we can, first off, see that Ahmdal's Law bounds the other curves from above. Secondly, if we attain a non-linear growth as a function of the cores we could lose all benefit gained from parallelization and even dip below 1 factor line. This would mean the overhead outweighs the benefit and we do worse than running it on a single core. Another thing we might expect is that the ratio of overhead to original execution code will shrink ad the size of the problem increases in certain ways. We will explore these ideas in the following subsections.

4.2 Forward Euler (Diffusion)

We begin by taking the simplest example, constant diffusion and parallelizing the execution. We do this by describing the spatial domain using N points (not including the boundaries) and dividing these up into c pieces. Since it is possible that N does not divide evenly into c we assign each core $(N + c - 1)/c$ and the last core the remaining points. In practice, I did run into the issue where the code complained when the number of spatial points did not divide evenly into

the number of cores. Since a local machine often has a limited number of cores, I used the fact that 60 is a number that is divisible by many numbers (as recognized by the Babylonians). Therefore I always used an N that was a multiple of 60 to avoid this issue (you can construct other numbers that would satisfy this behavior).

Since we will be using a centered space method we will need left and right adjacent points to step forward in time. This requirement creates a problem at the edge of each sub-domain as one of the points will be missing. TO address this, each of these cores were assigned two extra points that overlap into their adjacent neighbors domain called halo points. In the case of the edge of the domain these points on one side became the boundary point. ' Once we initial-

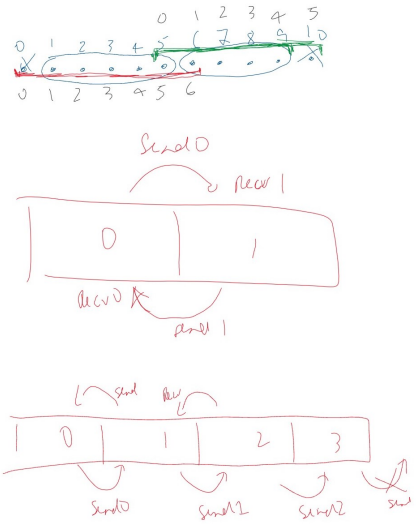

Figure 11: Divided Spatial Domain over Cores

ized all the values correctly each point can be updated in time using the regular explicit method discussed in the earlier section. Once this is done, each core must then communicate their updated values to their neighbors halo points for next iteration.

To accomplish this, I used the Message Passing Interface (MPI) inside a python environment to parallelize the tasks. The package I used is *mpi4py* [11] [8]. (This is not the most efficient method as loops need to be ported to C language I believe, and are done so in a way that requires the compiler to extract them beforehand. Regardless do to the ease/intuition of python and built in graphing software we choose to use this language to gauge the increase in performance.We focus on small scale scalability, number of cores used versus increase in performance given a problem size. Memory, may be another issue

if we need to reload and reload local information that may take longer and be inefficient. We will not focus on memory issue here although they may be occurring and skewing results)

4.2.1 Run Time Data (N = 60,000 M = 10)

Using the built in timer package in *mpi4py*. We time only a subset of the process for the purposes of gathering data. We skip over some of the initialization and the write sequence at the end. Additionally we run the timer once to not include the *gather* command mentioned before. The idea is to see if there is some extreme overhead in executing such a possibly nonlinear command that may outweigh the benefit of parallelization for say smaller problems.

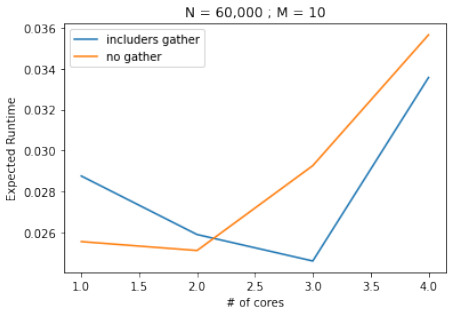

Figure 12: 60,000 spatial, 10 temporal points of discretization

We run the calculation 2000 times and use their average to determine an expectation of our run time. We can see the results for both in [Figure 12] and it is important to note that these are two separate executions. There are several strange features within our results. For the one core and two cores runs the data seems well behaved where the 'no gather' lies below the other. The run times also decrease here when run over 2 cores. Yet there are some obvious and not so obvious issues with the results. We note, first off, that the graphs differ for one core substantially even though the gather task has nothing to actually to gather from any other process. The gather execution continues to decrease over three cores but jumps up dramatically at four cores. Wanting to check if this was some sort of gather feature we can see that it strangely jumps already at three cores even above the one that includes gather! This is so bizarre that this needs a more technical explanation.

To this last point, we must assume that during that execution the CPU was overburdened and was running slower since they technically were two separate executions. Yet this point, with some further explanation can also shed light on how the run time increased towards the end when using most or all of our cores. There are a few indicators of this which will follow later.

To explain why run time strangely increased the theory requires understanding that the process was run on a local machine that had other tasks. Now if

my process only requires one core then I assume that the dynamic scheduler assigns the task to a free core. This way it avoids random interrupts in completing the calculations. But if I am using most my cores or all my cores, it becomes impossible to avoid interruptions as the local machine must complete background tasks. Now, even if we assume that the scheduler is not assigning tasks intelligently we will still face a problem. If it is assigning tasks randomly to different cores then there is a probability p that our specific task on this specific core will be interrupted. Let c be the number of cores processing our task simultaneously. Then the probability that i many interrupts happen is $P(X = i) = \binom{c}{i} p^c (1-p)^{c-i}$. The expectation of interruptions in such a binomial distribution is cp. Yet one could argue that even though the expected number interrupts goes up the number of cores running the execution and this ratio would keep us at p, the same as if one core was running the execution. To understand why this is not the case one must realize that when one core gets interrupted it in fact has some significant probability of interrupting its neighbor's execution. This is due to the communication that has to occur between them. If the cores to not execute in step the other cores must wait to receive the necessary information to move on. Therefore if one task stalls then the neighbors stall and their neighbors from there and so on. This creates a likely quadratic effect from each stall and would create unnecessary slowdowns at a higher number of cores.

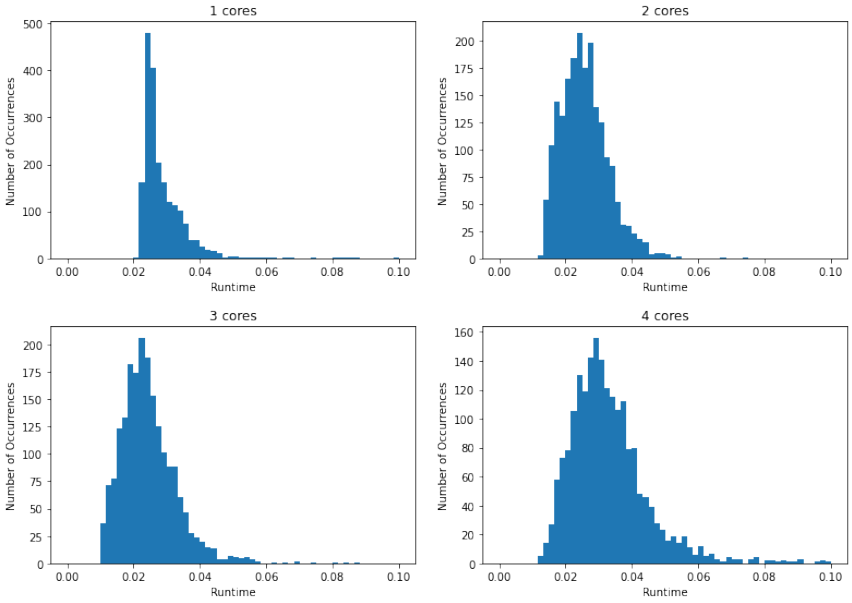

Figure 13: Histogram Distribution of Runtimes for 'gather' Runs

Alternatively, one could argue that we do not need many cores to be interrupted but just one by the above argument to create the slowdown. There are

two ways to handle this. One is to say that we are assuming that only one other task is running in the background that is causing this interruption and this is accounted for in p. Therefore p is more accurately the probability of the background task interrupting this task on this core and not the other cores (a nestled geometric probability). Otherwise, to evaluate the probability of at least one core having been interrupted we can use the CDF of a geometric distribution: $P(X \geq 1) = 1 - (1-p)^c$. The total number of expected interruption in this scenario, because all the cores are interrupted, is $C * (1 - (1-p)^c)$ and we can see how as the number of cores raises the number of interrupts non-linearly.

We are going to see how further the data suggests these interruptions are occurring. If the interruptions are random, not only should they be pushing out their expected run time, but the spread of run times should have a higher variance as we add cores, it should be less deterministic than lower cores. Also, we can look at the minimum time of all the runs in a set as a signal that we are not reaching theoretical speeds for our computation generally. Looking at the [Figure 13] we can see the variance jumps substantially as the expected run time increases dramatically. Additionally the minimum run time to consistently decrease as the number of cores are increased suggested that sometimes interruptions do not occur and we are able to get the theoretical speedup we are expecting. Yet on average that is not what we can expect on a local machine with background tasks. We can likely determine the value of p (where we suppose that each interruption take an certain average amount of time) to fit the curve using a Poisson like distribution which is a good approximation to the sum of many binomial random variables. The different binomial random variables occur from each sub-task required to complete the total runtime.

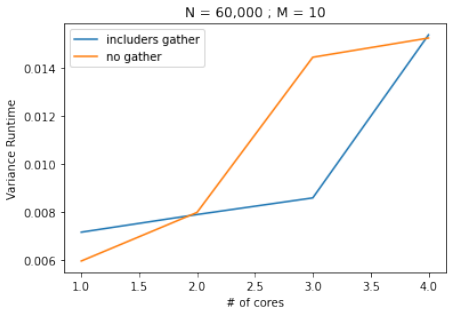

Figure 14: Variance on 60,000 spatial, 10 temporal points of discretization

There are other factors that suggest interruptions are occurring. If we plot the runtimes in series of how they were generated we may be able to see the interruptions happening during heavier load times for the cpu. In [Figure 16] we can see how the runtimes jump around the 213th iteration stay higher for a time, not completely random.

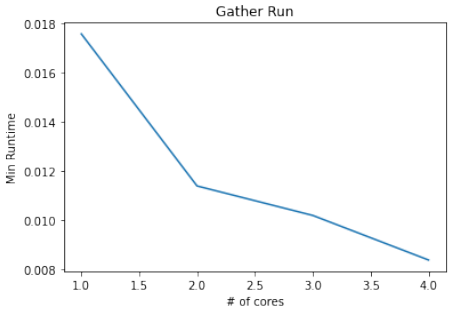

Figure 15: Minimum Runtimes

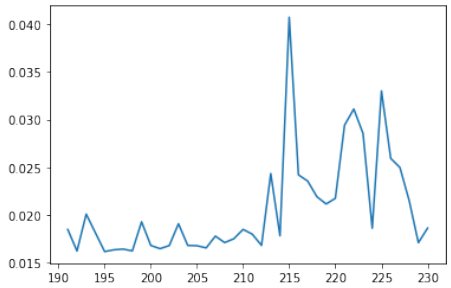

Figure 16: Runtimes versus Iteration Number in Execution for one Core

4.2.2 Run Time Data (N = 600,000 M = 100)

On a different execution I increased the size of the problem in both spatial and temporal steps. I appended to each text files at different times (every one hundred iterations) and we can see some interesting features. If we look at the histogram of the runtimes, they seem quite bifurcated. This is likely due to the different times effected what load the cpu was handling at that time. Looking at [Figures 17 18] we can see this sudden change in runtime. The four core execution has a telling jump even at a time where the series of executions were uninterrupted by myself around the 65th iteration.

On a different execution I increased the size of the problem in both spatial and temporal steps. I appended to each text files at different times (every one hundred iterations) and we can see some interesting features. If we look at the histogram of the runtimes, they seem quite bifurcated. This is likely due to the different times effected what load the cpu was handling at that time. Looking at [Figures 17 18] we can see this sudden change in runtime. The four core execution has a telling jump even at a time where the series of executions were uninterrupted by myself around the 65th iteration.

We can also see the same type of behavior as in the last execution where at four cores the runtime increases. Yet here we can see that it is less dramatic.

4.2.3 Run Time Data (N = 6,000,000 M = 5000)

In this execution we further increased the size and resolution of the problem. Here yet again we saw a similar behavior to the previous cases. This would mean that the behavior is not due the small size of the problem. Next we try to reduce only the time steps required but keep the spatial size of the problem large.

4.2.4 Run Time Data (Large N, Small M)

Surprisingly in the case where we increase the size of the problem spatially but not temporally we get the original expectations closer to Amdahl's Law. What we see is that as we increase M the temporal number of points the the bottom of the curve flattens out. The ideas is that if M is increased further it creates a minimum in the curve not at the endpoint. One might think that this is strange since the complexity of the problem should not matter if it is the spatial or in the temporal. Yet this is not true as there is not a symmetry between them. In each temporal jump there is a needed communication whereas the spatial increase has no such requirement. Therefore the question now becomes why should communications matter more as a complexity than the actual spatial size of the problem. My guess is that the interrupts are more likely handled or inserted at the communication breaks (or stalls) than the random iterations over the spatial dimension which has no natural break. To test out these ideas I would need an independent server that does not have background tasks or not implementing all its cores to see if I get similar behavior. Unfortunately I was not able to get the correct packages installed on our University server to test these out independently but that would be the next natural place to go.

4.3 Diffusion (Implicit Methods)

In case where we are looking for more stable methods will will inevitably look to implicit methods. Since we generate a system of unknowns generally such as in the single order backwards Euler, or Crank-Nicolson higher order method which means that we will have to theoretically invert a matrix to solve for our unknowns. Of course we never actually invert a matrix to solve for our unknowns as this is inefficient and of a cubic complexity. Yet depending on the size of the problem we can open up this question and again and ask can we use parallelization to solve the problem or invert the matrix faster. There seems to be literature [2] on this already as the inversion of matrices comes up in many practical applications and we would like to use all our available computational power to speed up such a process. These are not simple and often to not load balance well. In a method like OpenMPI or any MPI method where dynamic distribution of tasks is not available, your increase in performance takes a hit. This must be done intelligently and trying different ways to slice up the problem that make numerical sense and sense in terms of load.

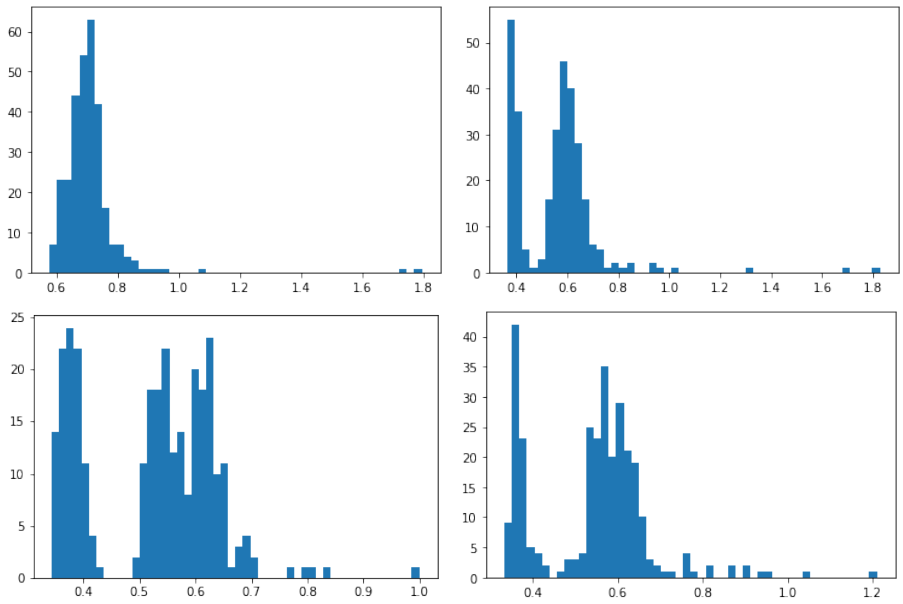

Figure 17: Histogram Distribution of Runtimes , increasing in cores left to right, form top to down

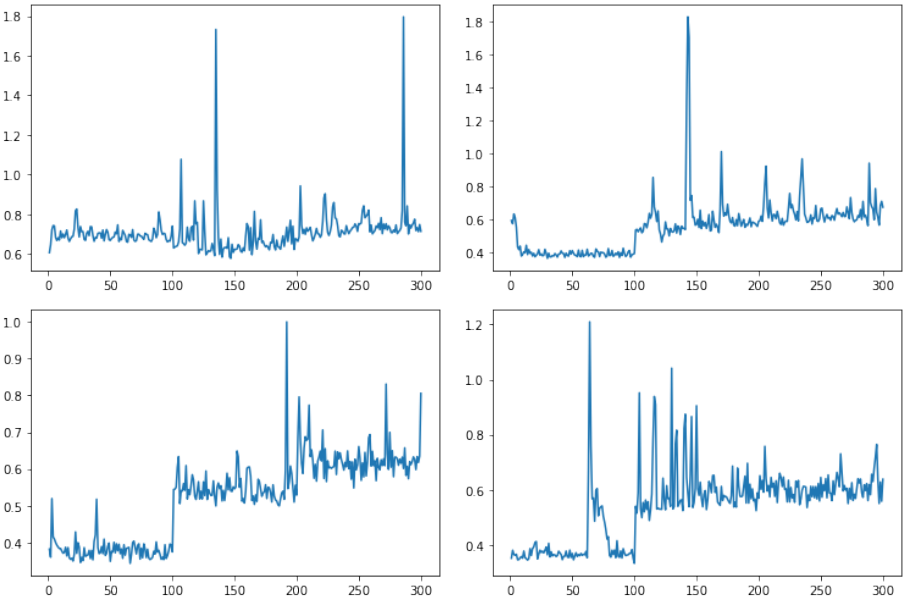

Figure 18: Series of Runtimes versus Iteration Number, increasing in cores left to right, form top to down

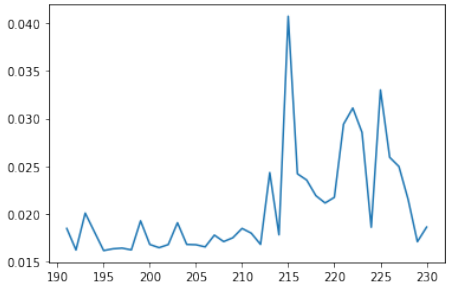

Figure 19: Runtimes versus Iteration Number in Execution for one Core

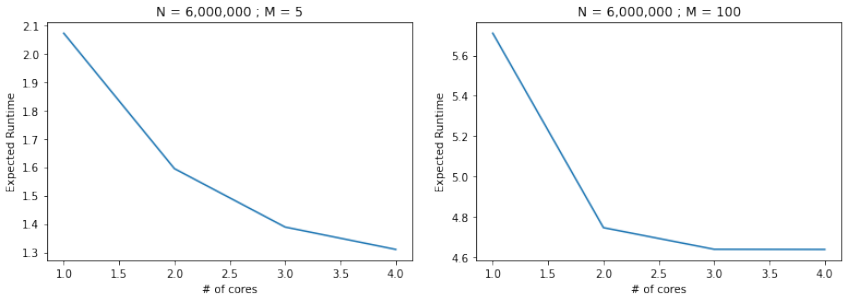

Figure 20: Runtime Expectations versus cores

References

[1] Randall J LeVeque and Randall J Leveque. *Numerical methods for conservation laws*. Vol. 3. Springer, 1992.

[2] Jack J. Dongarra et al. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, 1998. DOI: 10.1137/1.9780898719611. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719611. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719611.

[3] Gilbert Strang. *Computational science and engineering*. 9780961408817. 2007.

[4] Peter S. Pacheco. "Chapter 2 - Parallel Hardware and Parallel Software". In: *An Introduction to Parallel Programming*. Ed. by Peter S. Pacheco. Boston: Morgan Kaufmann, 2011, pp. 15–81. ISBN: 978-0-12-374260-5. DOI: https://doi.org/10.1016/B978-0-12-374260-5.00002-6. URL: http://www.sciencedirect.com/science/article/pii/B9780123742605000026.

[5] Florian Herzog. *Lecture notes in Stochastic Differential Equations*. 2013.

[6] quantpi. *Fokker Planck Equation Derivation: Local Volatility, Ornstein Uhlenbeck, and Geometric Brownian*. Accessed Jun. 12, 2020. Sept. 2019. URL: https://youtu.be/MmcgT6-lBoY.

[7] Kyle Mandli. *Lecture notes on Numerical Methods of PDE's*. https://https://github.com/mandli/numerical-methods-pdes. Accessed: 2020-02-15. 2020.

[8] *A Python Introduction to Parallel Programming with MPI 1.0.2*. URL: https://materials.jeremybejarano.com/MPIwithPython/collectiveCom.html.

[9] Richard Feynman. *Lectrure Notes on The Principle of Least Action*. https://www.feynmanlectures.caltech.edu/II_19.html. Accessed: 2010-08-27.

[10] Richard Fitzpatrick. *Lecture notes in Computational Physics: An introductory course*. https://https://github.com/mandli/numerical-methods-pdes.

[11] mpi4py. *mpi4py/mpi4py*. URL: https://github.com/mpi4py/mpi4py/tree/7db3f9c741d3dfd8dda14ffb537ed251280d2025.

[12] Benjamin Seibold. *Lecture notes in Numerical Methods for Partial Differential Equations*. https://ocw.mit.edu/courses/mathematics/18-336-numerical-methods-for-partial-differential-equations-spring-2009/index.htm.