Date

How is OpenMP used

OpenMP is usually used to parallelize loops
- Find your most time consuming loops.
- Split them up between threads.

Sequential program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++){
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[n], B[n], C[n];
#pragma omp parallel for
  for (int i=0; i<N; i++){
    A[i] = B[i] + k*C[i]
  }
}
```

↖ Single Program Multiple Data (SPMD)

OpenMP Constructs

1) Parallel Regions
   # pragma omp parallel
2) Work Sharing
   # pragma omp for # pragma omp sections
   Data Environment
   # pragma
3) Runtime functions/environment variables
   int my_thread_id = omp_get_num_threads();
   omp_set_num_threads(8);
4) Data Environments
   # pragma omp parallel shared / private
5) Synchronization
   # pragma omp barrier

} explained in next pages.

When you enter a # pragma omp parallel region there is a team of threads.
Variable in program can be visible to all threads (shared) or each thread can get own copy (private).
(Choosing right scope is the core of correct parallel code.)

- Variables declared outside a parallel region are shared by default (unless you specify otherwise)
- Local automatic variables declared inside the parallel region are private by default

Clauses: shared(...), private(...), firstprivate(...), lastprivate(...), reduction(...), default(none).

Date

Example 1: RACE condition

```
#include <iostream>
#include <omp.h>
int main(){
    int counter = 0;          // shared by default
    #pragma omp parallel      // multiple threads execute this block
    {
        // race: multiple threads increment the same counter concurrently.
        counter++;
    }
    cout << "counter = " << counter << "\n";
    return 0;
}
```

You'll often see counter < number_of_threads because of races.

Fix A (Atomic)

```
#pragma omp parallel
{
    #pragma omp atomic
    counter++;
}
```

Fix B (Reduction)

```
int N = omp_get_max_threads();
int counter = 0;
#pragma omp parallel for reduction(+:counter)
for (int i=0; i<N; i++) counter++;
```

reduction(+:counter) gives each thread a private partial sum and combines them safely at the end (fast & lock-free).

private vs firstprivate

private(var) → each thread has an unitialized private copy.

firstprivate(var) → each thread's private copy is initialized from the original variable.

Talkpoint: private is used when threads should compute their own independent value; firstprivate when they need the original initialization.

lastprivate (retrieve last-iteration value) Use when you want the final iteration's private values copied back to original value.

default(none) + explicit clauses forces you to declare the scope of each variable (good practice)

Talkpoint: This default(none) forces explicit thinking about each variable's scope - reduces mistakes

Short notes on other constructs

- reduction (+ : var) - great for sums/products: much faster than locking.
- threadprivate - makes a global variable private per thread. (advanced, less common)
- Always prefer local private variables where possible, minimize shared writable state.

What is OpenMP

Open source Multiprocessing

Multiprocessing → Multithreading - formed by many companies like AMD, Intel, IBM, HP, NVIDIA, ORACLE etc.

OpenMP is available for languages C++, C & Fortran.

Released in 1997 for fortan language in next year it released for C/C++ langerage.

Each process start with one main thread. This thread is call master thread in OpenMP. for a particular block of code, we create multiple threads along with master thread are called slave threads. OpenMP is called Fork-Join model. It is because, all slave threads after execution get joined to the master thread.

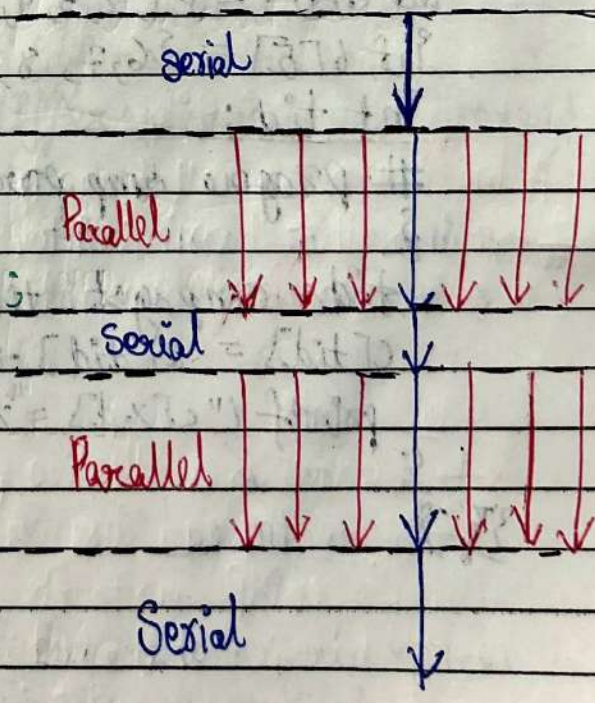
i.e Process starts with single master thread & ends with single master thread.

For C++ we have to include "omp.h" header file.

→ fork because creating multiple threads
Join because all slave threads are giving result to master thread

Open MP Fork and Join Model

```
printf("program begin \n")
N = 1000;
#pragma omp parallel for
for(i=0; i<N; i++)
    A[i] = B[i] + C[i];
M = 500;
#pragma omp parallel for
for(j=0; j<M; j++)
    p[j] = q[j] - r[j];
printf("program done \n")
```



OpenMP is called as Shared Memory model as openMP is used to create multiple threads & these multiple threads share the memory of main Process.

Date

A simple example to create threads.

```
# pragma omp parallel
{
printf ("Hello World")
}
```

By default, number of threads is equal to no. of processor cores. (In i3, i5, i7 there are generally 4 cores so 4 threads)

To create required no. of threads.

```
# pragma omp parallel num_threads(7)
{
printf ("Hello world")
}
```

// 7 threads will be created.

OpenMP program for Array Addition.

```
#include <stdio.h>
#include <omp.h>
void main()
{
int a[5] = {1,2,3,4,5};
int b[5] = {6,7,8,9,10};
int tid;
# pragma omp parallel num_threads (5)
{
tid = omp_get_thread_num();
c[tid] = a[tid] + b[tid]
printf ("c[%d] = %d\n", tid, c[tid]);
}
}
```

compile cmd -
gcc -fopenmp arrayaddition.c

Date

Outputs of array sum program,

c[1] = 9	c[2] = 11	c[1] = 9
c[2] = 11	c[0] = 7	c[0] = 7
c[3] = 13	c[4] = 15	c[3] = 13
c[4] = 15	c[1] = 9	c[4] = 15
c[0] = 7	c[3] = 13	c[2] = 11

We are getting result in different order reason is that all threads are running simantaneously so whichever thread completes its computation first that will get printed first.

We can create multiple for "for" block.

```
# pragma omp parallel for num_threads (5)
for (i=0; i<5; i++)
{
    printf ("Hello World");
}
```

Allocating different work to different thread

In openMP, we can allocate different work to different thread by using "sections".

```
# pragma omp parallel sections num_thread (3)
{
    # pragma omp section
    {
        printf ("Hello World One")
    }
    # pragma omp section
    {
        printf ("Hello World Two")
    }
    # pragma omp section
    {
        printf ("Hello World Three")
    }
}
```

Spiral

Date

Synchronizing Threads

In OpenMP, we are creating multiple threads. By default, variables are shared by all the threads. A block of code where resources (variables) are shared by multiple threads is called "critical section".

There is race condition i.e competition among threads to use these resources (variables).

Can avoid such race condition by using processor directive "# pragma omp critical"

Eg:

```
# pragma omp parallel num_threads(5)
{
  # pragma omp critical
  {
    x = x+1;
  }
}
```

As we are mentioning "# pragma omp critical", only one thread can do the operation of "x = x+1" at one instance.

Synchronization

A barrier makes all threads in the team wait until every thread reaches that barrier. After the barrier, memory updates made before the barrier are guaranteed to be visible to other threads (barrier implies a flush/consistency point).

Key points

- A barrier is a collective synchronization - every thread must reach it. If some thread doesn't, your program deadlocks
- Many OpenMP constructs insert implicit barriers (eg # pragma omp for has a barrier at the end, # pragma omp parallel has a barrier at the end of region unless nowait is used in some constructs.)
- Use barrier when you need an explicit phase seperation eg. Phase1 - all threads fill part of an array
  Phase2 - all threads read the full array.

Date _____

Implicit barriers vs explicit barriers

Common constructs include implicit barriers:

- #pragma omp for – implicit barrier at the end (unless nowait)
- #pragma omp single – implicit barrier at end (unless nowait)
- #pragma omp sections – implicit barrier.

So you often don't need an explicit #pragma omp barrier.

Use barrier when the structure of your code requires it.

```
#pragma omp parallel
{
    if(omp_get_thread_num() == 0){
        #pragma omp barrier  // WRONG: only thread 0 hits barrier
                             // others don't → deadlock.
    }
}
```

All threads must reach the barrier, placing it inside a branch taken by only some threads is illegal.

- #pragma omp critical – serializes a block (only one thread executes at a time). useful for small updates (printing, pushing to shared container)
- #pragma omp atomic – lightweight atomic update for simple operations (eg. x += v)
- #pragma omp taskwait – waits for child tasks of the concurrent task (not the same as barrier).
- flush – force memory visibility without synchronization. (rarely needed explicitly, barrier implies flush).

Talkpoint: prefer default(none), prefer reduction for aggregates, avoid unnecessary barriers – they cost time.

Spiral

Date

Structured Blocks.

Most OpenMP constructs apply to structured blocks.

- Structured block: one point of entry at the top & one point of exit at bottom.
- the only "branches" allowed are STOP statements in C/C++ `exit()`.

A structured block

```
#pragma omp parallel
{
more: do_big_job(id);
    if(++count>1)goto more;
}
printf("All done\n");
```

Not a structured block.

```
if(count==1) goto more;
#pragma omp parallel
{
more: do_big_job(id);
    if(++count>1) goto done;
}
done: if(!really_done()) goto more;
```

- In C++/C a block is a single statement or a group of statements between brackets {}.

Book Introduction to parallel Computing Date OpenMP

The whole point of parallel processing = faster programs (performance) and better cost-performance ratio.

Some programs are so demanding (e.g., weather simulation, AI-training, scientific modeling) that even the fastest single CPU can't finish them in reasonable time.

Solution: run parts of the program at the same time on multiple processors.

Why cost performance

~~Some programs are so demanding (eg. weather~~

even if you can run your program on very high-end CPU, you might get the same (or better) performance by using several cheap processors together.

→ Parallel processing can give more speed with for less money.

Rewards	Costs.
Raw performance → finish competition faster.	Increased design Complexity (need to think how tasks are split among processors)
Price performance → same or better using cheaper hardware.	Increased programming complexity (writing correct & efficient parallel code is harder than writing serial code)
	(you have to avoid race conditions, handle synchronization, & manage data-sharing)

Why OpenMP

- writing parallel programs is complex.
- OpenMP's goal: make it easier to add parallelism to existing programs

It does this by:

Providing a standard API (work across platforms)

Using compiler directives / pragmas so you can add parallelism without rewriting whole program.

Shared-memory multiprocessors

multiple CPUs / cores that all share the same main memory.

Ex: My laptop with 4 cores - all cores access the same RAM.

This is different from distributed-memory systems (eg. a cluster with many separate computers) which usually require MPI.

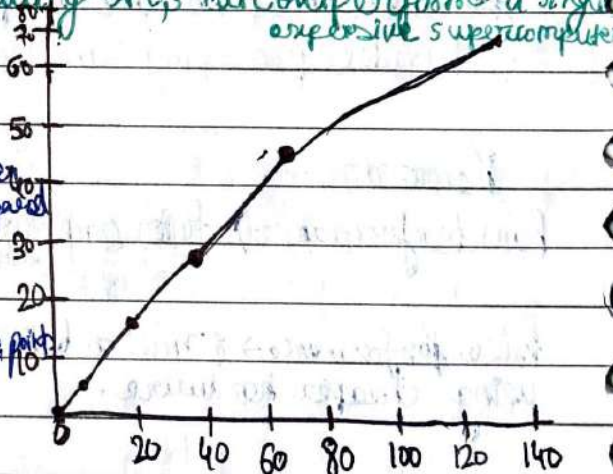
Spiral

Date

Talkpoint:

- Parallel processing is about speed & cost-efficiency, running a program across many processor instead one.
- The trade off is increased coding complexity, that's when OpenMP comes in, it makes it easier to parallelize existing code, especially for shared memory systems.

The MM5 weather model share shows how OpenMP can take a computation from impossible to practical, turning days of simulation into hours, by scaling across many processors.
This speedup directly translates to timely, accurate forecasts.
It's not just about speed, it's also about speed it's also about cost-efficiency since multiple commodity CPUs can outperform a single expensive supercomputer.



X-axis = Number of processors used
Y-axis = Speedup (how many times faster the program runs compared to a single program).

- Near linear scaling at first
  Doubling processors ≈ doubling speed-up to a point.
- As no. of processors increases:
  Speedup still improves but with diminishing returns (due to overheads & parallelization limits - Amdahl's law)

1 CPU = baseline speed 64 CPU's = ~60x faster 128 CPU's = close to 80x faster.
Even fastest single processors can't match this combined performance.
Cray YMP supercomputer → 512 MFLOPS per CPU (fast at the time) still not enough for real-time forecasting.
Origin 2000 with many cheaper processors beats it in total throughput.
This is more cost-effective than building a single huge super-fast processor.
CFD (Computational Fluid Dynamics)
Solving equations that model fluid flow (air, water, etc.)
Computation-heavy Aerospace (aircraft design) Automotive (aerodynamics) Turbine design

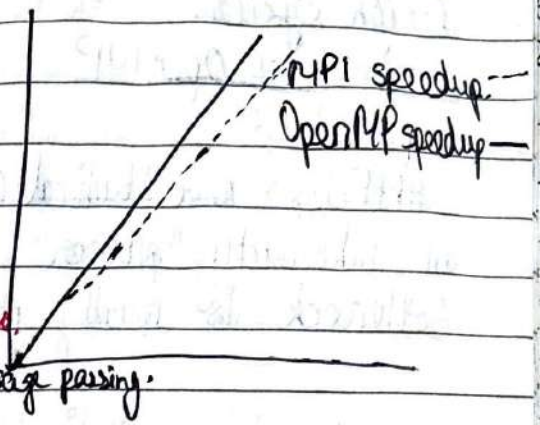
Spiral

Date

NAS Parallel Benchmarks, APPFLU.

APPFLU is one of the NAS parallel Benchmarks (NPB) used to measure how well a program scales on multiple processors.

In this benchmark, OpenMP's speedup is almost identical to MPI's up to the tested processor count.



This proves that OpenMP can deliver parallel performance comparable to MPI for shared-memory systems, without the extra complexity of message passing.

Real World point:

If your problem fits into shared memory (like a single powerful server node), OpenMP can give you near-MPI performance with less development overhead.

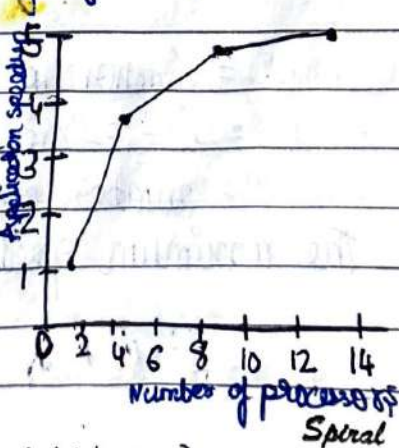
Parallel computing enables huge performance boosts but only if:

1) The problem is parallelizable (large independent chunks of work).
2) Overhead (thread management, memory creation) is small compared to actual computation.

example application — Crash simulation (full scale car crash analysis)

- Extremely expensive if done physically.
- Parallel computing makes high-detail simulations economically feasible.
- Used in auto industry & govt. safety agencies.

Performance of a leading crash code



The curve rises quickly at first, but flattens as processors increase - classic diminishing returns.

Early gains: going from 1 → 4 → 8 processors gives large speedups.

Later gains: adding processors beyond that yields smaller improvements due to:

- Parallel overhead
- Non-parallelizable code sections (Amdahl's law).

Spiral

Date

Talkpoint: Even with diminishing returns parallel computing greatly reduces simulation time for such complex engineering problems, making them feasible for regular use in product design cycles.

Benefits of OpenMP

Scalability

APPFLU's near identical OpenMP/MPI scaling shows that openMP is not inherently "slower" than MPI in shared memory systems, the bottleneck is usually memory bandwidth or algorithms limits.

Incremental parallelism

OpenMP lets you parallelize your code in stages, ideal when you can't dedicate months of to a full rewrite.

Economic feasibility

Real crash tests or large fluids simulations are very expensive in the real world. Simulating them on parallel systems is for cheaper, faster and safer.

Amdahl's law connection

Speedup is limited by the serial fraction of code. For eg: if 10% of the code is serial, the max speedup is 10x no matter how many processors you throw at it.

Amdahl Law Formula (additional info from internet)

f_s = fraction of time spent in serial code
$p = 1 - f_s$ = fraction that can be parallelized
N = number of processors

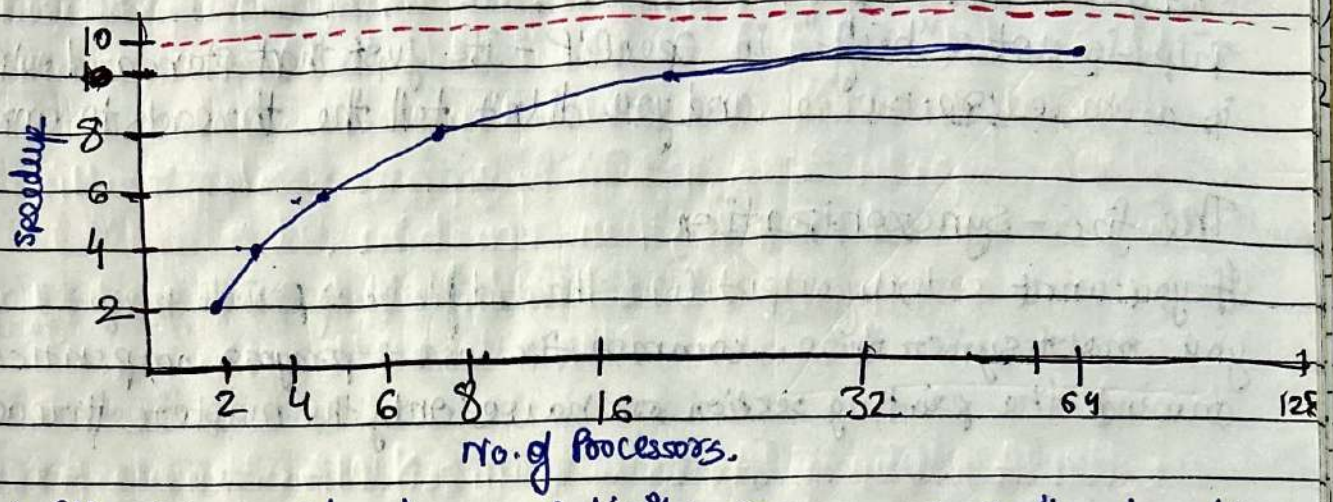
The maximum speedup is:

$$S(N) = \frac{1}{f_s + \frac{p}{N}}$$

Spiral

MPI = group of people in seperate rooms talking via phone calls (message passing).
OpenMP = group of people at the same table sharing one notebook. (shared memory).
For APPFLU, both groups work equally fast because the problem is well suited to shared memory. Data

Amdahl's law 10% serial fraction



> With 10% serial code even infinite processors can't go beyond 10x speedup.
> The curve rises quickly at first (1→8 processors) but flatten - diminishing returns.

OpenMP basics

> Controlled via directives (# pragma omp ...), runtime functions (omp_get_thread_num()) & environment variables (OMP_NUM_THREADS)

> Uses fork-join model: master thread → fork team → parallel work → join back.

Directives # pragma omp parallel {/* work */}
Key functions: omp_get_thread_num() → returns thread id
omp_get_num_threads() → returns total threads in the team.
omp_set_num_threads(n) → set threads programmatically.
Environment variable
OMP_NUM_THREADS → sets default threads without changing code.
Ex: set OMP_NUM_THREADS=4 (windows)
Output can get jumbled.
> When you run # Pragma omp parallel & each thread prints its ID, all threads share the same output screen.

> They run independently and don't wait for each other.

Spiral

Date

That means one thread might print while another imidiately starts printing before the first thread finishes - so the text appears interleaved.

This is not a "bug" in OpenMP - its just that standard output (stdout) is a shared resource, and you didn't tell the threads to coordinate.

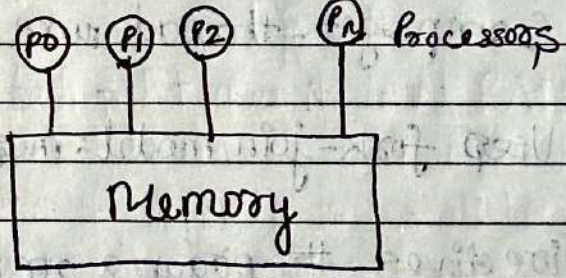
The fix - Syncronization.

If you want orderly output (one thread finishes printing before another starts) you must syncronize. common fix use #pragma omp critical around the printing section or have only the master thread print results.

Syncronization in OpenMP is broad topic & includes:
Barriers (wait until all threads arrive) Critical sections Atomic Updates Ordered execution.

THE OPENMP PARALLEL COMPUTER

Several CPUs / cores connected to one, globally addressable memory.
(all threads see the same addresses)



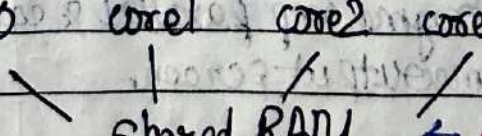
WHAT IT LOOKS LIKE

Shared Memory multi Processors

Two common flavors
1) UMA (SMP). 2) ccNUMA.

UMA (a.k.a SMP) - uniform memory access.

All cores see one shared RAM and the same latency/ bandwidth to any address.



← ~same access cost for all cores.

- Easiest way case for OpenMP: performance mostly depends on total memory bandwidth & algorithm.

Spiral

Date _____

ccNUMA – cache-coherent Non-Uniform Memory Access.

- One address space (still "shared memory") but RAM is split into per-socket banks.

Socket 0: cores + local RAM 0
Socket 1: cores + local RAM 1
} Interconnect (keeps caches coherent)

Core on S0 → RAM 0 = fast (local)
Core on S0 → RAM 1 = slower (remote)

- Good OpenMP code must place data near threads that use it (first-touch). Otherwise many accesses become remote → slower.

"first-touch" matters on ccNUMA
- The OS maps physical pages for a big array when they're first written.
- If thread 0 alone initializes a 20 GB array, every page lands in socket 0's memory.
- Later, when all sockets read that array, sockets 1...N do remote access → big slowdown.

Fix: initialize in parallel so each thread "touches (writes) its own chunk first, pages locally for that thread's socket.

Program which shows bad (serial first touch) vs good (parallel first touch) and measures difference in the compute phase with result uploaded in latex file

Talkpoint:

UMA: Any core → any memory with roughly equal cost. OpenMP is straightforward memory placement rarely bites you.

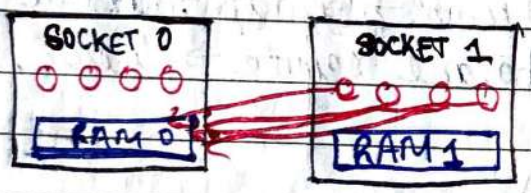
ccNUMA: Still one address space, but placement matters. Use parallel first-touch, stick to static chunking when data is block-partitioned, & consider OMP_PROC_BIND to keep threads near their data Spiral

Date

Rules of Thumb

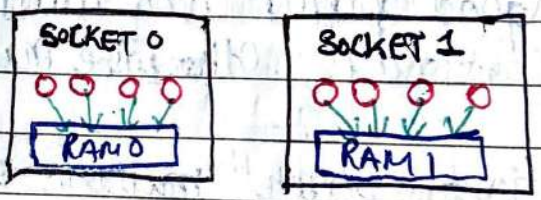
- Initialize big arrays in parallel (first-touch)
- Keep each thread working mostly on its own chunk (avoid ping-pong)
- Use schedule(static) for predictable locality; use dynamic only if you truly need load balance.
- For I/O or reductions, minimize cross-socket traffic; do per-thread partials, then combine.
- If performance drops as you add threads past one socket's core count, it's often NUMA bandwidth/latency.

ccNUMA Memory Access – Bad First Touch



Bad first touch
Socket 1 cores access RAM 0 (remote)

ccNUMA Memory Access – Good First Touch



Good first touch
Each socket uses its local RAM.

Pitfalls to know: (especially on NUMA)

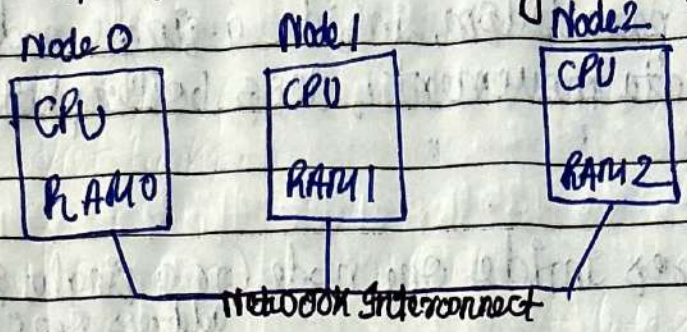
- Memory bandwidth can bottleneck at high thread counts.
- NUMA placement matters: if one initializes a huge array, all its pages may land on one socket's memory → other sockets read remotely (slower).

Pro tip: "first-touch" initialization
Initialize big arrays in parallel so pages are near the threads that will use them.

HPC → High Performance Computing.

Date

What is distributed-memory (message-passing) architecture



- Each node/process has its own private memory. There is no single globally addressable RAM.
- If process A needs data that lives in process B's, A must ask B i.e send a message.
- Examples of systems: clusters, networked workstations, IBM SP2 (CM-5), modern HPC clusters.

Implication: programmer must design where data lives & explicitly move (send/receive). This gives scalability (can use many nodes) but increases complexity.

Trade offs: MPI (distributed) Vs OpenMP (shared)

Pros (MPI / distributed Memory)
- Scales across many nodes - can use huge aggregate memory (& compute)
- Good when dataset exceeds single-node RAM.

Cons (MPI)
- You must explicitly send/recieve data → more code, possible bugs.
- Network latency/bandwidth matters - communication cost can dominate if granularity is small.

Spiral

Date

Pros (OpenMP / shared memory)
- Easier to add parallelism inside a single node via pragmas
- No explicit data movement, less boilerplate.

Cons (OpenMP):
- Limited to cores inside one node (or a single shared address space).
- Must manage NUMA locality (first-touch covered).

Rule:
- If program will run on a cluster, across nodes → use MPI (or hybrid)
- If you only target a single multi-core machine → OpenMP is simpler.

Hybrid MPI between nodes + OpenMP inside a Node
(common in High Performance Computing (HPC))

Why hybrid?
- Use MPI to partition global domain across nodes.
- Within each node, use OpenMP threads to use all cores & reduce MPI ranks (less MPI communication).

Important: MPI and threads interact → use MPI_Init_thread to request support level.

Simple hybrid (code of 1st product using this on Intern File made)

- MPI_Init_thread lets you control thread level support. For many hybrid programs MPI_THREAD_FUNNELED or MPI_THREAD_SERIALIZED suffices. MPI_THREAD_MULTIPLE allows concurrent MPI calls from multiple threads but depends on MPI implementation performance.
- Hybrid reduces number of MPI ranks (lower memory per rank, fewer messages) but increases complexity (thread + message interactions).

Date

WHY OPENMP

- Shared memory multiprocessor systems have become much more common, powerful, and affordable in recent years.
- But, existing programming models were often designed for distributed memory systems, not shared memory.
- This caused a gap b/w hardware capabilities.

Goal of OpenMP

Provide a standard & portable API for programming on shared memory parallel systems.

Hardware Trends.

In recent years

- Quantity of processors in shared memory systems has increased rapidly
- Scalability (ability to handle more processors effectively) is improving.

- First multiprocessor systems had only 2 processors → quickly moved to 4 & beyond.
- Medium to high-end systems now support 2 to 32 RISC processors (examples: Compaq AlphaServer, SunEnterprise)
- Falling prices → high-end systems used in smaller organisations too.

Software Gap

- Different vendors had their own APIs for shared memory programming.
- Distributed memory models (like MPI) became common, but
  - They're harder to use for shared memory because they need explicit message passing.
- Without a standard API, developers often:
  - Used vendor-specific solutions.
  - Relied on message passing even when all processors share memory, which is inefficient.
- OpenMP emerge as the standard, making shared parallel programming more portable and easier.

Spiral

Date

Shared vs Distributed Memory Programming

Feature	Shared Memory	Distributed Memory
Ease of parallelizing an app.	Easy to do; less effort	Harder; often all or nothing work.
Scaling to many processors	Limited by hardware vendor's design (eg, up to few dozen CPUs)	Scales to large number of processors
Extra code needed	Minimal, compiler help	More extra code, must explicitly send/receive messages
Impact on code quality	Slight complexity increase; debugging easier.	Code more complex; harder to debug.
Portability	Requires shared memory hardware & compiler support	Runs anywhere, even without shared memory

Why Not Just Use MPI for Shared Memory?

MPI is standardized, widely available & portable. But

- Requires explicit message passing.
- Programmer must manually partition data.
- Minimal incremental parallelization - you often have to rewrite a lot of program.
- Many modern processors already share memory caches, so message passing can be unnecessary overhead.

Spiral

Date

Other Alternatives

- Pthreads: Good for shared memory, but
  - Lower-level → more complex & awkward for large-scale Scientific Computing
  - Portability issues b/w platforms.
- Proprietary vendor APIs: Not portable, locks you to one system
- New languages (eg: Java): Adoption is slow, porting existing codebases is hard.

Why OpenMP Wins

- High level, easy to learn (compiler directives like #pragma omp parallel)
- Portable across vendors and systems.
- Allows incremental parallelism:
  You can parallelize parts of your code gradually.
- Works well with existing serial code - don't have to rewrite everything.

History of OpenMP

- OpenMP: Standard API for shared memory parallel programming (est. 1997)
- Predecessors: X3H5, HPF - failed due to hardware variety & portability issues.
- Pre-OpenMP Problem: Hardware capable, but no portable model.
- Motivation: Developer demand for a single standard across vendors.
- Development: Led by Industry group (hardware + software vendors)
- ARB: Governing body for OpenMP standard.
- First Standard: Released July 1997.

ARB → Architectural Review Board (ARB).

Spiral

Chapter 2 Getting Started with OpenMP Date

A parallel programming language must provide three things:

1. Specifying parallel execution.
   → How to tell computer "this program to run parallel"
2. Communication between threads.
   → Threads may need to share or pass values
   - In OpenMP (shared memory), communication = just reading/writing shared variables.
   - OpenMP uses #pragma omp parallel for to say split this loop among threads.
   - In MPI (distributed memory) communication = explicit send/receive messages.
3. Synchronization between threads
   → We must coordinate so that don't step on each other toes.

   Examples in OpenMP

   #pragma omp barrier (all wait here until everyone arrives)

   #pragma omp critical (only one thread enters this block at a time)

Different approaches to parallelism.

The book mentions 3 approaches that languages/libraries take care:

(a) New language constructs
   - Example: Fortran 95 has forall keyword,
   - The language itself add syntax for parallelism.

(b) Directive-based (OpenMP's way)
   - We use compiler directives (#pragma omp)
   - Directives looks like comments, so if the compiler doesn't support OpenMP, it just ignores them → code runs as normal serial.

   Advantage: one code works for both serial and parallel.

Spiral

Date _____

(c) library calls (MPI, pthreads)
- No new language syntax, but special functions calls (MPI_Send, pthread_create)
- More manual, but also more flexible.

Two Big benefits of OpenMP compare directives

1. Same code works everywhere.
   - With OpenMP pragmas → if OpenMP is off, they're ignored, program runs serially.
   - So you don't to maintain two versions of the code.

2. Incremental parallelism:
   - can start with a working sequential program,
   - Then gradually add # pragma omp lines to parallelize more parts.
   - Safer and easier to debug.

At the highest level, OpenMP is:
- A set of compiler directives (# pragma omp...)
- Runtime library routines (like omp_get_thread_num())
- Environment variables (like OMP_NUM_THREADS).

→ Parallel languages must handle execution, communication, synchronization.
- Approaches: new constructs (Fortran forall), directive-based (OpenMP), libraries (MPI, pthreads)
- OpenMP is directive based, so:
  1. Same code runs serially if pragmas are ignored.
  2. Parallelism can be added incrementally.

In C/C++, we use # pragma omp pragmas + OpenMP runtime functions

Spiral

Date

OpenMP Directives (#pragma omp)

In C++, OpenMP relies on the #pragma keyword.

#pragma: tells the compiler to do something special, but if the compiler doesn't understand it, it just ignores it.

- If you compile with OpenMP (g++ -fopenmp file.cpp)
  - Multiple threads run.
- If you compile without OpenMP (g++ file.cpp)
  - The directive is ignored → only one thread prints the line.

Talkpoint: You don't need two versions of code. One program works for both serial and parallel.

Conditional Compilation with _OPENMP

Sometimes you want your program to behave differently depending on whether OpenMP is enabled or not.

```
#include <iostream>
using namespace std;
int main(){
    #ifdef _OPENMP
        cout << "Compiled with OpenMP!" << endl;
    #else
        cout << "Compiled without OpenMP" << endl;
```

- If compiled with -fopenmp → prints the OpenMP message.
- Without -fopenmp → prints sequential message.

This is super useful for debugging, or for writing portable software that should run anywhere.

Spiral

Date

Combining Directives & Normal Code.

```
#include <iostream>
#include <omp.h>
using namespace std;
int main(){
    int sum=0;
    #pragma omp parallel for reduction (+:sum)
    for(int i=1; i<=100; i++){
        sum+=i; // Each threads works on a piece of loop
    }
    cout << "Sum from 1 to 100=" <<sum<< endl;
}
```

// with openMP → loop is split across threads, then results are combined (reduction)

// Without OpenMP → loop runs normally still (code it.)

If you're working on a large scientific simulation
- You want it to run fast on a supercomputer (many threads)
- But you also want to test it on your laptop (single processor)

With OpenMP directives:
- On laptop → directives are ignored → program still runs.
- On supercomputer → directives activate → massive speedup.

- If OpenMP not enabled → treated as comment → safe fallback
- _OPENMP macro → lets you check if code is compiled with OpenMP.
- Benefit: single codebase, portable, incremental development
Example: parallel loops, conditional messages.

Using # pragma omp critical

```
Shared total_iters = 0
Thread 1:
    compute depth (1)
    ---- Enter critical ----
    total_iters = total_iters + depth(1)
    ---- leaves critical ----
Thread 2:
    compute depth (2)
    ---- Wait (if inside critical) ----
    ---- Enter critical ----
    total_iters = total_iters + depth(2)
    ---- leaves critical ----
Thread 3:
```

Using reduction (+: total_iters)

```
Thread 1: local_sum=0 → local_sum+=depth(1)
Thread 2: local_sum=0 → local_sum+=depth(2)
Thread 3: ...
```

Each thread keeps its own private copy - no waiting!

critical = safe but slower (thread wait-ish)

at the end:
total_iters = sum of all local sum
total_iters = depth(1) + depth(2) + depth(3)

Date

critical:

Only one thread at a time updates total_iters. Others must wait (causing delays)

Safe, but slower (threads wait in line)

Reduction:

No waiting. Each thread works independently. At the end, OpenMP combines results automatically.

Safer + faster (no waiting, automatic combination)

Conditional compilation - done

Parallel ~~structs~~ control structures - the "fork-join" model

Communication & the data environment (shared vs private vs reduction)

OpenMP is shared-memory: all threads can see the same global / heap variables.

But each thread has its own stack (automatic variable inside a function are private to thread unless shared by clause.

A. shared - one copy visible to all threads.

```
int hits = 0;
#pragma omp parallel shared(hits)
{
    // WARNING: writing 'hits++' from many threads races!
}
```

Spiral

Date _____

B private - each thread gets its own copy (uninitialized)

```
#pragma omp parallel private(tmp)
{
    int tmp;                    // private search
    tmp = compute_local();
    // tmp is invisible to other threads
}
```

C firstprivate - private but initialized from the original value.

```
int seed = 123;
#pragma omp parallel firstprivate(seed)
{
    // each thread starts with seed = 123, then can modify its own copy
}
```

D lastprivate - after a worksharing loop, keep the last iteration's value.

```
int last_i = -1;
#pragma omp parallel for lastprivate(last_i)
for(int i=0; i<N; ++i){
    // ...
    last_i = i;
}
// here, last_i == N-1;
```

E reduction - private during the region, combined at end.

```
double sum=0.0;
#pragma omp parallel for reduction(+:sum)
for(int i=0; i<N; ++i){
    sum += a[i];    // each thread has a private sum, they're added at end.
}
```

Spiral

Date

Without reduction, there would be a race.

Best practice: force yourself to be explicit.

```
#pragma omp parallel for default(none) shared(a,N) reduction(+:sum)
for (int i=0; i<N; ++i) sum += a[i];
```

default(none) makes the compiler warn if you forgot to declare scope.

Communication rule of thumb: communication is just reads/writes to shared variables, but correctness/visibility is guaranteed at synchronization points (eg., end of parallel for, barrier, critical, atomic). Don't assume another thread sees your write immidiately unless there's a sync.

Date _____

Black Scholes Problem.

> An option is a contract giving the right to buy an asset at strike price K at expiry T. A European call pays $\max(S_T - K, 0)$ at time T.

Goal: given current price S and time t, find the option price $V(S, t)$.

Black Scholes derive a partial differential equation (PDE) for option price $V(S, t)$:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

with final condition $V(S, T) = \max(S - K, 0)$.

boundary conditions
$V(0, t) = 0$
$V(S \to \infty, t) \approx S$

S = the underlying asset (stock) price.
- It's a variable that can range 0 to ∞
- On numerical grid, this is space axis.

t = time (current time b/w now and expiry).
- In the PDE form, we usually solve backward in time from the final condition $t = T$ back to $t = 0$.

$V(S, t)$ = the option price when the asset is at price S at time t.
- This is the unknown we are solving for.

Ex: If today ($t = 0$) the stock price is $S = 70$, then $V(70, 0)$ is the fair value of option today
- At expiry ($t = T$), we already know $V(S, T) = \max(S - K, 0)$

Spiral

Date

Closed-form Black-Scholes solution.

$$C(S,t) = S\phi(d_1) - Ke^{-r(T-t)}\phi(d_2)$$

where

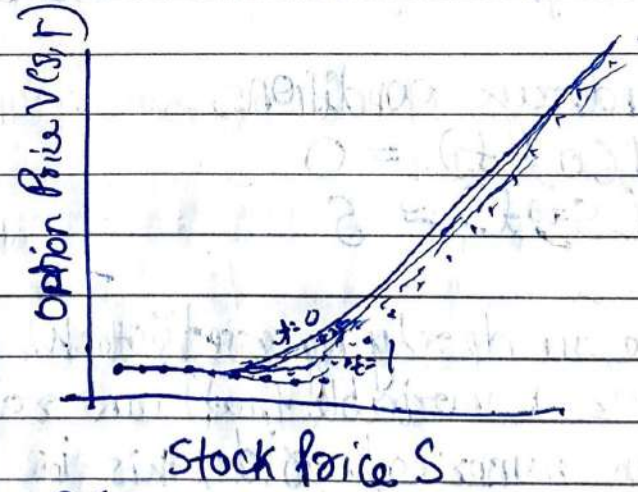
$$d_1 = \frac{\ln(S/K) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}},$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

σ → volatility
r → rate.
T → expiry (time to expiry).

Ex:

$K = 85$
$r = 0.1$
$\sigma = 0.2$
$T = 1$



Truncation $S_{max} = 3K = 255$.

Spatial steps $M = 51 \rightarrow \Delta S = S_{max}/M = 255/5 = 50$
Time steps $N = 100 \Rightarrow \Delta t = T/N = 0.01$.

Spiral

Date _____

For small S (far below strike) option ≈ 0

For large S option ≈ $S - Ke^{-r(T-t)}$
(non-linear for deep in money).

As time decreases toward expiry (in t toward T in our indexing), the curve moves toward the payoff line.

At $t=0$ & $S = S_{max} = 255$, the analytic price $\approx 255 - Ke^{-rT}$.

Compute $e^{-rT} = e^{-0.1} \approx 0.904837418$.

$Ke^{-rT} = 85 \times 0.904837418 = 76.911805$.

$V(S_{max}, 0) \approx 255 - 76.911805 = 178.088$
1947.

Matches top range of curve

first derivative

$$\left.\frac{\partial V}{\partial S}\right|_j = \frac{V_{j+1} - V_{j-1}}{2\Delta S}$$

second derivative

$$\left.\frac{\partial^2 V}{\partial S^2}\right|_j = \frac{V_{j+1} - 2V_j + V_{j-1}}{\Delta S^2}$$

Ex: $\Delta S = 5$, $\Delta t = 0.01$

$S_{center} = 100$ $(j = 20)$, $t = 0.99$ $(n = 99)$

$V(95) = 10.084958$

$V(100) = 15.08458$

$V(105) = 20.084958$

$\frac{\partial V}{\partial S} = \frac{10}{10} = 1.0$

$\frac{\partial^2 V}{\partial S^2} = \frac{20.084958 - 2(15.08458) + 10.084958}{5^2}$

Date

$$\Delta t \leq \frac{\Delta S^2}{\sigma^2 S_{max}^2}$$

$\Delta S = 5 \Rightarrow \Delta S^2 = 25$.

$\sigma = 0.2 \Rightarrow \sigma^2 = 0.04$,

$S_{max} = 255 \Rightarrow S_{max}^2 = 255^2 = 65025$

$\sigma^2 S_{max}^2 = 0.04 \times 65025 = 2601.0$

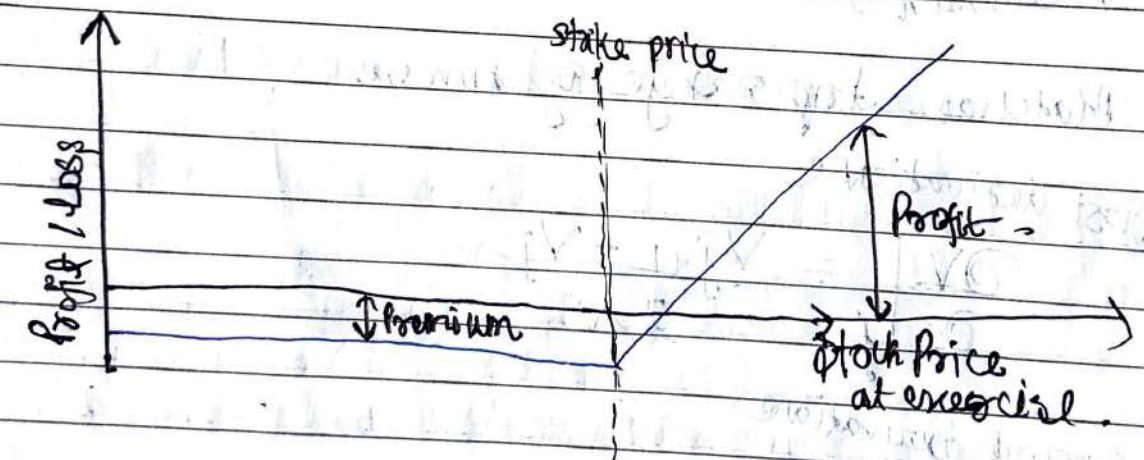
$\Delta t \leq \frac{25}{2601.0} \approx 0.00961168$

Compare

$\Delta t = 0.01$

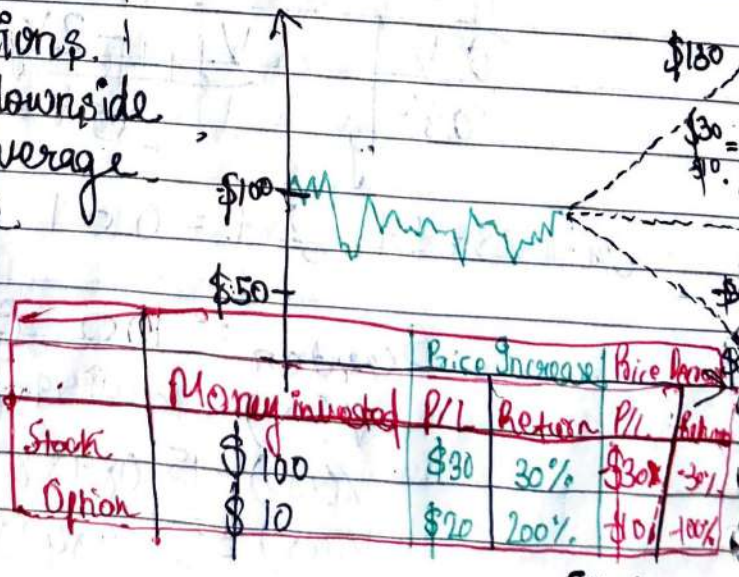
$0.01 > 0.00961168$.

Profit and loss on an option.



3 advantages of options.

1. Limits your downside.
2. Provide leverage.
3. Use as a hedge.



	Money invested	Price Increase P/L	Return	Price decrease P/L	Return
Stock	$100	$30	30%	-$30	-30%
Option	$10	$20	200%	-$10	-100%

Date

~~Implicit~~

We want to solve Black Scholes PDE for $V(S,t)$

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0,$$

with final condition $V(S,T) = \max(S-K, 0)$

We discretize $S_j = j\Delta S$ and $t^n = n\Delta t$.

Let $V_j^n \approx V(S_j, t^n)$.

Spatial central difference approximations:

$$\left.\frac{\partial V}{\partial S}\right|_j \approx \frac{V_{j+1} - V_{j-1}}{2\Delta S} \qquad \left.\frac{\partial^2 V}{\partial S^2}\right|_j \approx \frac{V_{j+1} - 2V_j + V_{j-1}}{\Delta S^2}$$

PDE can be written as

$$\frac{\partial V}{\partial t} = \mathcal{L}[V] \quad \text{where} \quad \mathcal{L}[V] = -\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - rS\frac{\partial V}{\partial S} + rV.$$

We moved spatial terms to RHS so we march backward in time.

Explicit (forward time)

Replace time derivative by forward (in marching direction) explicit difference using known values at the late time level:

$$\frac{V_j^n - V_j^{n+1}}{\Delta t} = \mathcal{L}[V]^{n+1}$$

Spiral

Conditionally stable — requires very small Δt, esp if σ or δ_{max} are large. Date

or more commonly written stepping backwards

$$V_j^n = V_j^{n+1} + \Delta t \, \mathcal{L}[V]^{n+1}.$$

(when we march from $n+1$ (expiry) to n (earlier time), the right hand side uses the known values V^{n+1}.)

Plugging central differences:

$$V_j^n = V_j^{n+1} + \Delta t \left(A_j \left(V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1} \right) + B_j \left(V_{j+1}^{n+1} - V_{j-1}^{n+1} \right) - r V_j^{n+1} \right).$$

where $A_j = \frac{1}{2}\sigma^2 S_j^2$ $\quad B_j = \frac{r S_j}{2\Delta S}$.

Parallel across spatial nodes.

Implicit (backward time)

Use the spatial at the unknown new time level:

$$\frac{V_j^n - V_j^{n+1}}{\Delta t} = \mathcal{L}[V]^n.$$

Now RHS depends on unknowns V^n at all nearby spatial points. Rearranged this yeilds a linear system.

$$V_j^n - \Delta t \, \mathcal{L}[V]^n = V_j^{n+1}.$$

After assembling spatial discretization, this forms a tridiagonal linear system.

$$\alpha_j V_{j-1}^n + \beta_j V_j^n + \gamma_j V_{j+1}^n = RHS_j$$

$\forall j$. Solving to get V^n.

- Dampen high freq noise

Unconditionally stable for linear PDE like Blackscholes. Can use large Δt.

Date

Crank Nicolson.

Time centered avg of explicit & implicit.

$$\frac{V_j^{n+1} - V_j^n}{\Delta t} = \frac{1}{2}\left(L[V]^n + L[V]^{n+1}\right).$$

This yeild a tridiagonal system too, but is second order accurate in time.

Final Black Scholes presentation:

The Black scholes model (1973) is a PDE Model for option pricing. Before black scholes and merton developed this approach, finding a fair price of a derivative was a "big problem". In simple terms, the model follows a (geometric brownian motion with constant volatility & interest rate, and that one can form a risk free hedge.

Under these assumptions the option price $V(S,t)$ (as a f^n of underlying stock price S and time t) satisfies parabollic partial differential equation. Black scholes derived this Eq^n in 1973, merton gave parallel derivation. Scholes & Merton later won the 1997 Nobel Prize for discovery

The practical problem solved is: given the current stock price S, time to expiration T, volatility σ, and risk free interest rate r, what should be "fair" premium (price) of an option? We consider a European call option, which gives the right but not obligation to buy the stock at a fixed price K at the expiration date T.

The Black Scholes model provides an explicit (closed form) answer for such options, which balances the effects of time delay, volatility and discounting.

Date

To understand PDE,

) European call option: A contract giving its holder the right to purchase one share of underlying stock at price K, but only on the expiration date (not before). The holder would exercise the option to buy cheaply at K and immediately sell for the market price, otherwise option expires worthless.

) Strike price (K): The fixed price at which the option holder can buy (for a call) or sell (for a put) the underlying asset. For ex: if the strike price of a call is $50, the holder can buy the stock for $50 at expiry no matter it's market price.

The strike price is a key determinant of the option intrinsic value (the immediate profit potential) and moneyness.

) Expiration Time (T): The date (often measure from "today") when the option contract expires. For a European option, exercise is allowed at $t = T$.

The time remaining until expiry is often denoted $T = T - t$. All else equal, a longer time to expiry typically increases option's premium, because there is more time to move stock advantageously.

) Volatility (σ): A measure of how erratic the stock price movements are, typically defined as returns per unit time. High volatility means the stock price can swing widely, making the option more valuable (since large favorable becomes more likely). In the Black-Scholes model, the stock's volatility is assumed constant. Volatility is an important variable in the collection of option prices.

Date

- Risk-free interest rate: The continuously compounded rate of return of a risk-free investment (eg a govt bond) for ex: $1 invested today grows to $ $e^{(r \cdot T)}$ after time T. The risk-free rate enters the PDE because option pricing is done under a risk-neutral valuation essentially and assumes all investors accept return r on average, so that expected stock growth under price measure is r.

European call's payoff at expiration T is $\max(S_T - K, 0)$ if the stock price S_T exceeds K, the option pays off $S_T - K$, otherwise it pays 0. An option's premium today must reflect (under risk neutral valuation) the expected discounted payoff under the model.

The Black-Scholes PDE

Under the standard assumptions (no dividends, lognormal stock, constant r, σ) one can show the call price $V(S,t)$ satisfies the Black-Scholes PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

- $\frac{\partial V}{\partial t}$ represents decay of option value.
- $\frac{\partial^2 V}{\partial S^2}$ convexity in S
- $rS\frac{\partial V}{\partial S}$
- $-rV$

** Expressing fast moving prices in terms of slower changing factors like implied volatility & Greeks which allow for easy comparison b/w all the different contracts. Like

Comparing Option Prices and Vega
Comparing option prices in diff stocks would be chaotic and meaningless as prices are influenced by so many different factors: stock price, time to expiration, volatility & each can vary widely b/w stocks.

Spiral

but when you compare implied volatilities you're speaking common that cut through the noise & lets you focus on what really mat

Stock A: Option Price = $10, Option IV = 20% < more expensive
Stock B: Stock price = $3, Option IV = 60%

Professional options traders will figure out option prices in terms of implied volatility.

Delta $\Delta = \frac{\partial C}{\partial S}$ how the call price changes with the change in stock price S.

Gamma $\Gamma = \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta}{\partial S}$ rate of change of delta. how delta changes as stock price changes.

Theta $\Theta = \frac{\partial C}{\partial t}$ how the call price changes as time passes.

$$\Theta + \frac{1}{2}\sigma^2 S^2 \Gamma + rS\Delta - rC = 0$$

σ: volatility
C: Discount call price for present value (remove interest from future price)
S: Price of stock at expiration (due to interest).

Option strike is $100 & stock price is $75.

rS represent the forward price of stock this is the stock value at expiration if only interest affects its price.

negative rC discounts the call options value back to present terms meaning it removes the effects of interest from the option's future value at expiration.

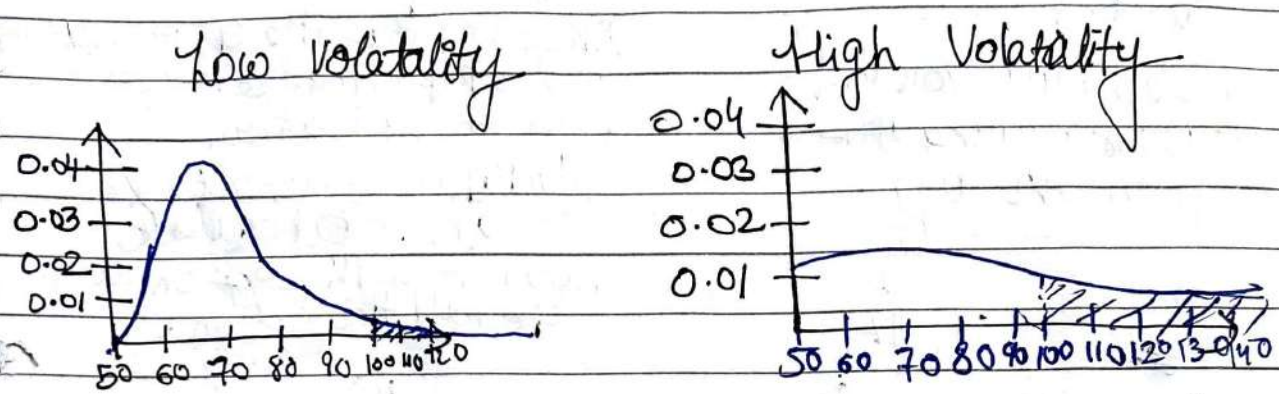
Volatility (σ) is the standard deviation of the assumed log normal distribution showing how the stock prices movements affect the rate of the Delta changes volatility reflects the uncertainty or variability in the stock's price.

Spiral

Date

and the higher it is the greater the potential for options price changes this means there is a higher chance of finishing in the money (ITM) and thus higher volatility will increase options prices.

Option Strike is $100 and stock price is $75.



We can see here that the same option with a strike of 100 and current stock price of 75 will priced higher if volatility is higher since it has greater probability of finishing above 100.

This is represented by larger area under the curve one key assumption in the black scholes model is that volatility stays constant throughout options life. that's not realistic but makes maths simpler while giving v useful framework.

$$C = SN(d_1) - Ke^{-rt}N(d_2)$$

$$d_1 = \frac{\ln(S/K) + (r + \sigma^2/2)t}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

To actually calculate these effects like greeks and call price this equation needs to be solved

The solution has been done for us and known as the famous black scholes model.

Date

$$C = \boxed{SN(d_1)} - \boxed{Ke^{-rt}N(d_2)}$$

average value of the stock above the strike price

in other words

The amount you will earn for owning this option on avg.

likelihood of paying strike price (ie likelihood of finishing ITM)

in other words

likelihood that the option holder will end up paying the exercise price at expiration

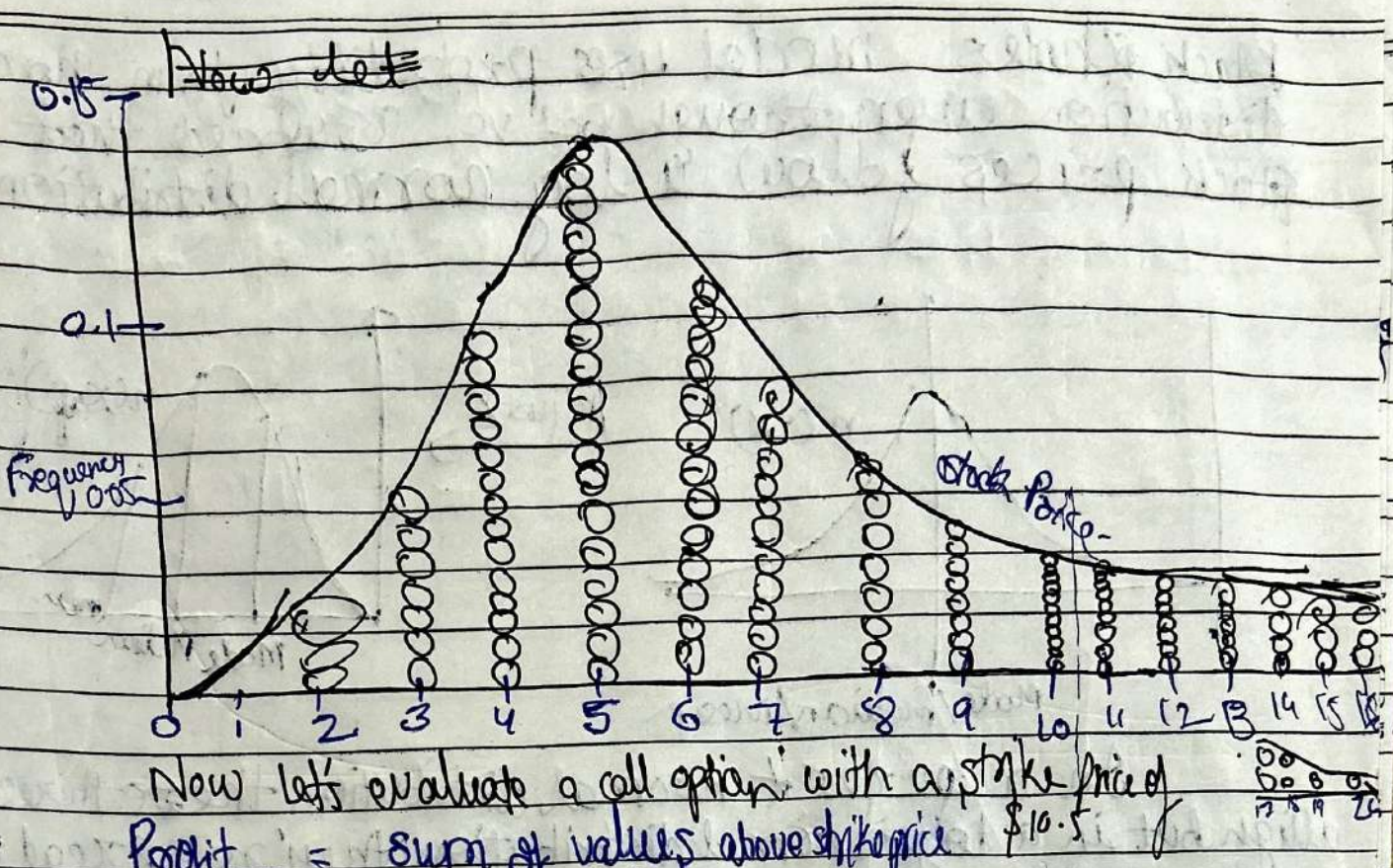
Multiplying by strike we can calculate how it will cost on avg to hold this option.

$$C = \text{Profit}_{avg} - \text{Cost}_{avg}$$

To explain this concept further we will imagine log normal distribution of stock prices at expiration where each possible price is at 50 cent interval.

A log normal distribution is just one where taking the log of values gives you a normal distribution it works for stock prices because stocks can't drop below zero and tend to follow this pattern historically

Date _____



Now let's evaluate a call option with a strike price of $10.5

$$Profit_{avg} = \frac{\text{sum of values above strike price}}{\text{Total \# of values}}$$

$$= \frac{\Sigma \$11 \times 8 + \cdots + 20 \times 1}{\text{Total \# of values}} = \$3.30$$

$$P(ITM) = \frac{\text{\# of values above strike price}}{\text{Total \# of values}} = 24.00\%$$

$P(ITM) = 24\%$.

$$Cost_{avg} = P(ITM) * \text{Strike} = 24\% \times 10.5 = \$2.52$$

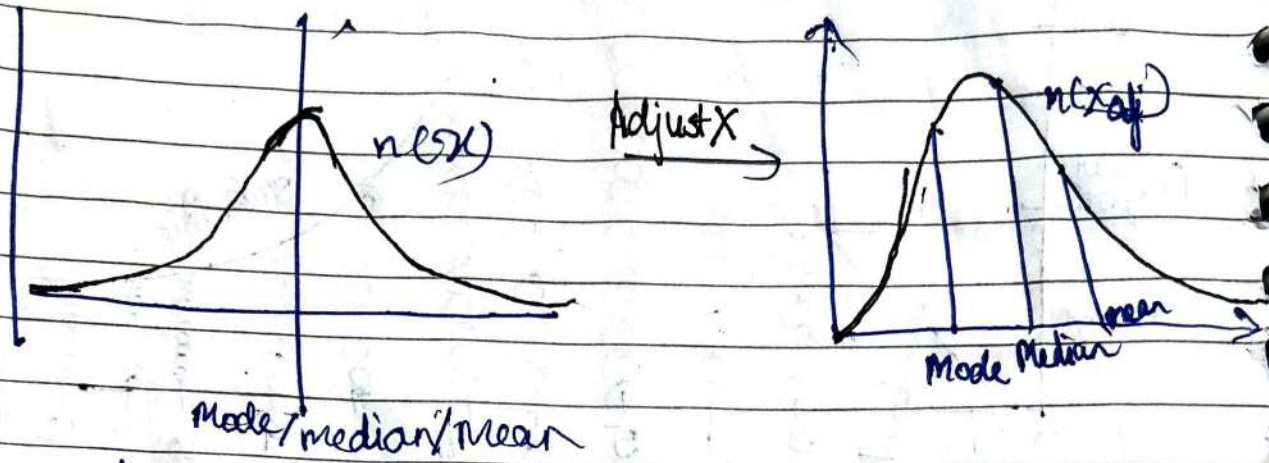
$$C = Profit_{avg} - Cost_{avg} = \$0.78$$

From this we can conclude that the current fair value of the option is 78 cents.

Spiral

Date

Black scholes model uses probability from normal distribution even though we've assumed that stock prices follow a log normal distribution.



In a perfect normal distribution these three align but in a log normal distribution they're spread out understanding these points helps us adjust from a log normal to a normal framework.

In black scholes the rel^n ship b/w strike price & stock prices key for normal dist it's just the stock price S minus strike price X $S - X$

But in log normal dist we have to adjust by taking log giving us $\ln(S/X)$ $\ln\left(\frac{S}{X}\right)$

If $S > X$ value is +'ve & call is in the money (ITM)

If $S < X$ it's −'ve & call is out of money (OTM).

Now let's further adjust for two key components that affect stock prices.

- interest rate.
- volatality.

Date

Since option are valued based on fwd price & fwd price depend upon interest rate. we need to adjust for interest rates over the life of option this adjustment ensures that pricing reflects the present value of the stocks expected price at expiration.

$$\ln\left(\frac{S}{X}\right) + rt$$

Volatality
In a log normal dist stock price are skewed with a longer right tail this results the mean of dist being shifted right side to the mode.
The shift is mathematically eqvt to $\frac{\sigma^2 t}{2}$.

Combining adjustments interest rate & volatility Give num^r for d_1

$$\ln\left(\frac{S}{X}\right) + \left(r + \frac{\sigma^2}{2}\right)t$$

to wrap things up we need to convert this value into number of standard deviations because S.D help measure how far the strike price is from the mean in a log normal distribution.

$$d_1 = \frac{\ln\left(\frac{S}{X}\right) + \left(r + \frac{\sigma^2}{2}\right)t}{\sigma\sqrt{t}}$$

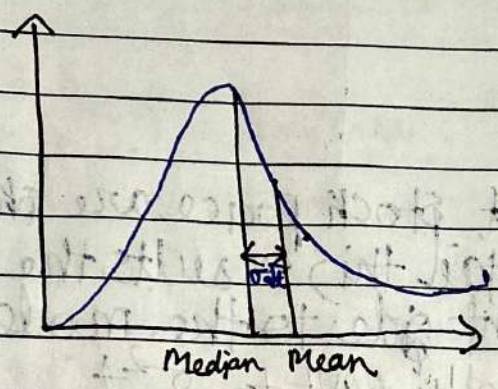
- how much option is in or out of the money
- Forward price adjustment (from interest)
- Adjustment for mean lognormal distribution
- Normalization factor (or a standard deviation) to determine the total amount of standard deviation.

Date

Essentially D_1 tells us how many std deviations the strike price is from the mean.

Combining this with the fwd price of stock gives us the expected avg profit of the option at expiration

$$\text{Average Profit at Expiration} = Se^{rt} N(d_1)$$



To figure out likelihood the option will be exercised we use the median of the log normal distribution this is the point that splits the total occurences in half and it usually falls to the left of the mean.

$$d_2 = d_1 - \sigma\sqrt{t}$$

The value d_2 uses the median to estimate the probability of the being in the money at expiration multiplying this probability by the exercise price gives us the avg amount we'll pay if we exercise the option

$$\text{Average Price Paid at Expiration} = XN(d_2)$$

Combining this with the first term will give us the expected value of call option at expiration ~~but because we're paying for option today~~

$$C_{\text{expiration}} = Se^{rt}N(d_1) - XN(d_2)$$

Spiral

Date

but because we are paying for the option today we have to adjust for present value multiplying both terms by e^{-rt} gives us the familiar black scholes model.

$$C_{today} = SN(d_1) - Xe^{-rt}(d_2) \quad ****$$

$$C_t + rSC_S + \frac{1}{2}\sigma^2 S^2 C_{SS} = rC$$

$$C(S,T) = \max(S-K, 0)$$

$$S = Ke^x \Rightarrow X = \ln(S/K)$$

$$\frac{dC}{dS} = \frac{dC}{dX}\frac{1}{S} \Rightarrow S\frac{dC}{dS} = \frac{dC}{dX}$$

$$\frac{\partial^2 C}{\partial S^2} = \frac{\partial^2 C}{\partial X^2}\left(\frac{1}{S}\right)^2 = \frac{\partial^2 C}{\partial X^2} - \frac{\partial C}{\partial X}$$

~~$\frac{\sigma^2}{2}$~~

Let $t = T - T/(\sigma^2/2)$

$$C_t = \frac{\partial C}{\partial t}\cdot\frac{\partial T}{\partial t} = C_T\left(\frac{\sigma^2}{2}\right)$$

$$-\frac{\sigma^2}{2}C_T + rC_x + \frac{1}{2}\sigma^2[C_{xx} - C_x] = rC$$

$$-C_T + \lambda C_x + C_{xx} - C_x = \lambda C$$

$$C_T = (\lambda - 1)C_x + C_{xx} - \lambda C$$

$C = KV$ $\quad V_t = (\lambda-1)V_x + V_{xx} - \lambda V$

Spiral

Date / /

Black Scholes Equation (Explicit)

Definition

Black Scholes Equation (BSE) is used to calculate the fair market value of a European option

The PDE → $$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \quad \text{— (1)}$$

- K = strike price (pre determined price at which holder of the option can buy the stock)
- V = price of the option → what we're trying to calculate
- S = current price of the stock / asset (constantly changing)
- t = time, T = expiration time of the option
- r = risk-free interest rate (like the interest rate on a Govt Bond)
- σ = Volatility of the asset
  - high σ (startup) → high value → expensive
  - low σ (utility company) → low " → cheap

$\frac{\partial V}{\partial t}$ = rate of change of option's price wrt time

$\frac{\partial V}{\partial S}$ = " " " " " " asset's price

$\frac{\partial^2 V}{\partial S^2}$ = " " " " $\frac{\partial V}{\partial S}$ " " "

Derivation

In order to computerize this equation, we take approximations of the derivatives according to FDM.

We will use grid V_{ij} → option value at time i, stock price j

for $\frac{\partial^2 V}{\partial S^2}$, we take the central difference formula for second derivative

i.e $$u''(x) = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2}$$

in the grid V_{ij}, $u(x+h) \Rightarrow V_{i,j+1}$, $u(x-h) \Rightarrow V_{i,j-1}$, $u(x) = V_{ij}$

$h \rightarrow \Delta S = \frac{S_{max} - S_{min}}{M}$ ↳ no. of steps

Date

$$\therefore \frac{\partial^2 V}{\partial S^2} = \frac{V_{i,j+1} + V_{i,j-1} - 2V_{ij}}{(\Delta S)^2} \quad \text{—(2)}$$

We take $S_{max} = 2K$ for a wide enough grid

S_{min} is taken as 0

M is the no. of steps we take in the price

N . . . . . time

so $\Delta S = \frac{S_{max} - S_{min}}{M} = \frac{2K}{M}$

$\Delta t = T/N$

for $\frac{\partial V}{\partial S}$, we take central difference formula for first derivative

i.e $u'(x) = \frac{u(x+h) - u(x-h)}{2h}$

$$\therefore \frac{\partial V}{\partial S} = \frac{V_{i,j+1} - V_{i,j-1}}{2\Delta S} \quad \text{—(3)}$$

for $\frac{\partial V}{\partial t}$, we take backward difference formula for first derivative

If we were to take central difference formula, it would be more accurate, but then we would have 3 different time steps i-1, i and i+1

So at the cost of some accuracy, we use the backward difference formula so as to have only i and i-1 s.t. i-1 can be taken as the only unknown variable

$$\therefore \frac{\partial V}{\partial t} = \frac{V_{ij} - V_{i-1,j}}{\Delta t} \quad \text{—(4)}$$

Date / /

from ①, ②, ③ and ④, we get

$$\underbrace{\left(\frac{V_{ij} - V_{i+1,j}}{\Delta t}\right)}_{\frac{\partial V}{\partial t}} + \frac{1}{2}\sigma^2 S_j^2 \underbrace{\left(\frac{V_{i,j+1} + V_{i,j-1} - 2V_{ij}}{(\Delta S)^2}\right)}_{\frac{\partial^2 V}{\partial S^2}} + rS_j \underbrace{\left(\frac{V_{i,j+1} - V_{i,j-1}}{2\Delta S}\right)}_{\frac{\partial V}{\partial S}} - rV_{ij} = 0$$

Transformed a PDE to an algebraic formula

$S_j = j\Delta S$

$$\Rightarrow \frac{V_{ij} - V_{i+1,j}}{\Delta t} = -\frac{1}{2}\sigma^2 (j\Delta S)^2 \left(\frac{V_{i,j+1} + V_{i,j-1} - 2V_{ij}}{(\Delta S)^2}\right) - r(j\Delta S)\left(\frac{V_{i,j+1} - V_{i,j-1}}{2\Delta S}\right) + rV_{ij}$$

$$= -\frac{1}{2}\sigma^2 j^2 (V_{i,j+1} + V_{i,j-1} - 2V_{ij}) - \frac{1}{2} rj (V_{i,j+1} - V_{i,j-1}) + rV_{ij}$$

$$V_{i+1,j} = V_{ij} + \Delta t \left[\frac{1}{2}\sigma^2 j^2 (V_{i,j+1} + V_{i,j-1} - 2V_{ij}) + \frac{1}{2} rj (V_{i,j+1} - V_{i,j-1}) - rV_{ij}\right]$$

Coeff of $V_{i,j-1} \rightarrow \frac{1}{2}\sigma^2 j^2 \Delta t - \frac{1}{2} rj\Delta t = \frac{1}{2}\Delta t j (\sigma^2 j - r) = a$

Coeff of $V_{ij} \rightarrow 1 - \sigma^2 j^2 \Delta t - r\Delta t = 1 - \Delta t(\sigma^2 j^2 + r) = b$

Coeff of $V_{i,j+1} \rightarrow \frac{1}{2}\sigma^2 j^2 \Delta t + \frac{1}{2} rj\Delta t = \frac{1}{2}\Delta t j (\sigma^2 j + r) = c$

$V_{i+1,j} = a_j V_{i,j-1} + b_j V_{ij} + c_j V_{i,j+1}$

$$\therefore \boxed{V_{i+1,j} = a_j V_{i,j-1} + b_j V_{ij} + c_j V_{i,j+1}} \quad — (5)$$

This is the equation we will use to code this up

Stability

This method only gives reliable solution if

$$\Delta t \leq \frac{1}{\sigma^2 M^2}$$

to get this, increase N to a suitable number.

↳ No. of time steps.

Date

- Boundary conditions

In order to use eq 5, we will need some boundary conditions i.e the values at the edges of our grid V_{ij}

1) At Expiration ($t = T$)

At the time of expiry, the value of the option its immediate cash value. The holder will only exercise their right to buy the stock if $S > K$ (current price of stock is more than the agreed upon price) and their profit is $S - K$. if $S \leq K$, the option is worthless at value 0.

$\therefore V_{T,j} = \max(S_j - K, 0) \quad \forall j$

2) At zero stock price ($S = 0$)

At any time if S reaches 0, the value of the option is 0 as it is the right to buy a stock at rate K. $\therefore$ 0 is an absorbing barrier, once a stock's S hits 0, it cant get any higher given any amount of time (Bankrupt)

$\therefore V_{t,0} = 0 \quad \forall t$

3) At a Very high stock price ($S = S_{max}$)

if S hits very high, almost certainly the option will be exercised at expiration. $\therefore$ holder will pay K for S. But the payment of K will happen in future at expiration so to get its present value by discounting it back to t using risk free rate r,

$V_{t,S_{max}} = S_{max} - K e^{-r(T-t)}$

With these three conditions in place, we can write the code

Later on, I learned that the convention is to write $V(S, T)$ instead of $V(T, S)$ so I've made the changes in the code.

Date / /

Black Scholes Equation (Implicit)

Reason

The explicit method is fast but suffers with conditional stability. Implicit method is used to overcome this weakness. It requires more calculation but is unconditionally stable. Here, no matter the choice of M and N, the solution is stable.

Derivation

$$\frac{\partial V}{\partial t} = \frac{V_{ij} - V_{i-1j}}{\Delta t}$$ This is the same as in explicit

$$\frac{\partial^2 V}{\partial S^2} = \frac{V_{i-1j+1} - 2V_{i-1j} + V_{i-1j-1}}{(\Delta S)^2}$$

$$\frac{\partial V}{\partial S} = \frac{V_{i-1j+1} - V_{i-1j-1}}{2\Delta S}$$

We are taking i-1 instead of i. This creates 3 unknowns for one known (V_{ij})

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

$$\Rightarrow \frac{V_{ij} - V_{i-1j}}{\Delta t} = -\frac{1}{2}\sigma^2 S_j^2 \frac{V_{i-1j+1} - 2V_{i-1j} + V_{i-1j-1}}{(\Delta S)^2} - rS_j \frac{V_{i-1j+1} - V_{i-1j-1}}{2\Delta S} + rV_{i-1j}$$

$S_j = j\Delta S$

$$\Rightarrow V_{ij} - V_{i-1j} = -\Delta t\left[\frac{1}{2}\sigma^2 (j\Delta S)^2 \frac{V_{i-1j+1} - 2V_{i-1j} + V_{i-1j-1}}{(\Delta S)^2} + rj\Delta S \frac{V_{i-1j+1} - V_{i-1j-1}}{2\Delta S} - rV_{i-1j}\right]$$

$$V_{ij} = V_{i-1j} - \Delta t\left[\frac{1}{2}\sigma^2 j^2 (V_{i-1j+1} - 2V_{i-1j} + V_{i-1j-1}) + \frac{1}{2}rj(V_{i-1j+1} - V_{i-1j-1}) - rV_{i-1j}\right]$$

Coeff of $V_{i-1j-1} \rightarrow -\frac{1}{2}\sigma^2 j^2 \Delta t + \frac{1}{2}rj\Delta t = \frac{1}{2}j\Delta t(\sigma^2 j - r) = a_j$

Coeff of $V_{i-1,j} \rightarrow 1 + \sigma^2 j^2 \Delta t + r\Delta t = 1 + \Delta t(\sigma^2 j^2 + r) = b_j$

Coeff of $V_{i-1,j+1} \rightarrow -\frac{1}{2}\sigma^2 j^2 \Delta t - \frac{1}{2}rj\Delta t = -\frac{1}{2}j\Delta t(\sigma^2 j + r) = c_j$

Date

$V_{ij} = a_j V_{i-1,j-1} + b_j V_{i-1,j} + c_j V_{i-1,j+1}$

Known — unknowns

∴ There are 3 unknowns, we must use $Ax = b$

$x \rightarrow$ Unknowns vector $\rightarrow [V_{i-1,1}, V_{i-1,2}, \dots, V_{i-1,M-1}]^T$

$b \rightarrow$ Knowns vector $\rightarrow [V_{i,1}, V_{i,2}, \dots, V_{i,M-1}]^T$

$A \rightarrow$ Coeff matrix $\rightarrow$ Tridiagonal $\rightarrow$ main diagonal $\Rightarrow b_j$
- lower " $\Rightarrow a_j$
- upper " $\Rightarrow c_j$
- All other entries are 0

Coding Logic

$V \rightarrow M+1 \times N+1$ matrix

Boundary conditions same as in explicit ($t = T$, $S \geq 0$, $S = S_{max}$)

The change:

- Loop from $i = N$ to 1
  At each step, i is the known col, $i-1$ is the unknown col (b, x) — Happens outside loop
- Assemble A as the tridiagonal Matrix
- Assemble b using i^{th} col of V
- Solve $Ax = b$
- solved x contains the new $i-1^{th}$ col of V

Once the loop finishes, the option prices at $t = 0$ are in the 0^{th} col of V