Math

1. Given a fifty-seat room where each seat is numbered, people entered the room in ordered. The first one is drunk and he will take a seat randomly. The rest people will take their own seat as long as it is not taken. Calculate the probability that last two people can take their own seats.

The probability that the last two people can take their own seats are 1/3. Let's assume the drunk man takes #1 room by change, then it's clear all the rest of the people will have the correct seats. If he takes #49 or #50, then one of the last two people will not get their seat, one of them will seat at #1. The probabilities that the drunk man seat at #1, #49, #50 are equal. So the probabilities that the last two people can take their own seats are 1/3. Otherwise let's assume the drunk man takes the n-th seat, where n is a number between 2 and 48, Everyone between 2 and (n-1) will get his own seat. That means the n-th person essentially becomes the new "drunk" guy with designated seat #1. We could continue the previous analysis. Since at all jump points there's an 2/3 chance for the "drunk" guy to choose seat #49,#50 1/3 change choose his own seat. The probability that the last two people can take their own seats are 1/3.

4. (sellside) Given a stock, assume the implied volatility to time t1 is σ1 and the implied volatility to time t2 is σ2, calculate the correlation between St1 and St2 .

Assume the stock price follows Brownian motion, the variance of change in stock price Var(S2 – S1) = t2 –t1.

$$Var(S2 - S1) = \sigma_2^2 + \sigma_1^2 - 2\sigma_1\sigma_2\rho_{12} = t_2 - t_1$$

$$\rho_{12} = \frac{t_2 - t_1 - \sigma_1^2\sigma_2^2}{2\sigma_1\sigma_2}$$

5. Given a Brownian motion W_t, when is W_t^N a martingale? Why W_t^3 is not a martingale?

When N=0 or 1, W_t^N is a martingale.
When N=3, for any t>s>=0,

$E(W_t^3 - W_s^3 \mid W_s) = E((W_s+W_t-W_s)^3 - W_s^3 \mid W_s)$
$=E (3*(W_t-W_s)^2*W_s +3*(W_t-W_s)*W_s^2 + 3*(W_t-W_s)^3 \mid W_s)$
$= 3* W_s*(t-s) + 3* (E(W_t\mid W_s)-W_s)*W_s^2+ 3*E(W_t-W_s)^3$
$= 3* W_s*(t-s)$

It is not equal to zero. So $E(W_t^3\mid W_s) = W_s^3$ is not always true. W_t^3 is not a martingale.

6. (sellside) Calculate the price of the option with payoff 1 under traditional geometric Brownnian motion assumption.

$dS_t = S_t * (\text{mu } dt + \text{sigma } dW_t)$

Let f(x) =1/x, then f'(x)=-x^(-2), f"(x)=2x^(-3).

By Ito's Lemma, $d(1/S_t) = -S_t\text{\^{}}(-2) * dS_t + ½* 2* S_t\text{\^{}}(-3) * d\langle S\rangle_t$
$= (1/S_t) * [(-\text{mu} + \text{sigma}\text{\^{}}2) dt – \text{sigma } dWt]$

Use the Black-Scholes model, the Call option price $(1/S_t)* N(d_1) – K e\text{\^{}}(-(T-t)) * N(d_2)$, where d_1= [ln(1/ S_t /K) + (r+ ½ sigma^2) (T-t)] / (sigma* sqrt(T-t)) and $d_2=d_1$-sigma *sqrt(T-t).

7. (sellside) What is the definition of stability in numerical PDE? What is the definition of implicit and explicit scheme?

Stability: Numerical errors which are generated during the solution of discretized equations should not be magnified .

Explicit methods calculate the state of an ordinary or partial differential equations system at a later time using information from the state of the system at current time. While for an implicit method, we need to solve an equation containing information from both current and later time to get the state of system at later time.

8. Consider the basketball shooting game, define success rate as number of successful shoots divided by number of total shoots. Assume the successful rate rising from below 0.5 to above 0.5, is there a moment which has exactly success rate 0.5.

There is a moment which has exactly success rate 0.5.

For example, we use 0 to denote Failure and 1 denote Success. The sequence is 0, 1, 0, 1, 1….

The first three shooting rate is 1/3, the fourth rate is 1/2, and the fifth rate is 3/5.

Proof: Denote success by S and Failure by F.

When S < F, the successful rate < ½

S = F, the successful rate = ½

S > F, the successful rate > ½

If at the nth shoot the successful rate first > ½, the last shooting must be a successful shoot. It means S > F. Since S and F are both integers, we have S >= F + 1 and S -1 >= F.

S – 1 >= F means if we remove the last successful shoot at time n, we must have S = F at time n-1. Otherwise at time n-1 we have S > F, and it won't be the first time successful rate > ½.

It would be the same if the shooting rate is 80%. Then S = 4F.

Programming

10. Design an algorithm to check is a point is inside the triangle or not.

```
import numpy as np

def intriangle(p1, p2, p3, p):
    """p1, p2, p3 are 2 dimensional vectors
    aranged clock-wise on the plane"""
    t1 = np.cross(p1-p, p2-p)
    t2 = np.cross(p2-p, p3-p)
    t3 = np.cross(p3-p, p1-p)

    if ((t1>0 and t2>0 and t3>0)
           or (t1<0 and t2<0 and t3<0)):
      return True
    else:
      return False

if __name__ == "__main__":
    p1 = np.array([0,0])
    p2 = np.array([2,0])
    p3 = np.array([0,4])
    p = np.array([1,1])
    q = np.array([2,2])

    print intriangle(p1,p2,p3,p)
    print intriangle(p1,p2,p3,q)
```

12. Given an array, design an algorithm to return the longest decreasing sub-array.

```
#include<iostream>
#include<vector>

std::vector<int> findLongestDecreasingSubarray_v2(const int* ptr, int
len);
void printVector(const std::vector<int>& v2);
void printArray(const int *ptr, int len);

int main()
{
    int seq[] = { 7, 2, 4, 8, 7, 6, 5, 3, 6, 5};

    int length = sizeof seq / sizeof seq[0];
    std::cout << "The Original array is " << std::endl;
    printArray(seq, length);

    std::vector<int> largestDescendingSubarray =
findLongestDecreasingSubarray_v2(seq, length);

    std::cout << "The largest Desceding Subarray is" << std::endl;
    printVector(largestDescendingSubarray);
    system("pause");
}


void printVector(const std::vector<int>& v2)
{
    for (unsigned int i = 0; i < v2.size(); ++i)
    {
     std::cout << v2[i] << ",";
    }
    std::cout << std::endl;
}

void printArray(const int * ptr, int len)
{
    for (int i = 0; i < len; ++i)
    {
     std::cout << ptr[i] << "," ;
    }
    std::cout << std::endl;
}


std::vector<int> findLongestDecreasingSubarray_v2(const int* ptr, int
len)
{
    int start = 0;
    int end = 0;
    int largestStart = 0;  // The start of the previous largest
subarray
    int largestEnd = 0;    // The end of the previous largest subarray
    int largestLength = largestEnd - largestStart + 1;  // The length
of the previous largest subarray
    int currLength = end - start + 1;  // The length of the current
subarray

    // The original array is { 7, 2, 4, 8, 7, 6, 5, 3, 6, 5}

    for (int i = 1; i < len; i++)  // use i to loop through the array
```

```
	{
	 if (ptr[i] < ptr[i - 1])
	 {
		end = i;
		currLength = end - start + 1;
	 }
	 else
	 {
		if (currLength > largestLength)
		{
			largestStart = start;
			largestEnd = end;
			largestLength = currLength;
		}
		start = i;
		end = i;
		currLength = end - start + 1;
	 }

	 if (i == len - 1)
	 {
		if (currLength > largestLength)
		{
			largestStart = start;
			largestEnd = end;
			largestLength = currLength;
		}
	 }

	}


	std::vector<int> largestDescendingSubarray;

	for (int i = largestStart; i <= largestEnd; ++i)
	{
	 largestDescendingSubarray.push_back(ptr[i]);
	}

	return largestDescendingSubarray;
}
```

13. Implement the Sudoko algorithm.

```
#include<iostream>
#include<ostream>
#include<istream>
```

```cpp
using namespace std;

class Sudoku{
public:
  //constructor;
  Sudoku();
  void Print();
  bool Solve();
  void setBoardValue(int nx, int ny, int value );

private:
  int board[9][9];
  bool Solve(int nx, int ny);
  bool verify(int nx, int ny);
};

Sudoku::Sudoku(){
  for(int i=0; i<9; i++)
    for(int j=0; j<9; j++){
      board[i][j]=0;
    }
}
bool Sudoku::Solve(){
  return Solve(0,0);
}

void Sudoku::setBoardValue(int nx, int ny, int value){
  board[nx][ny]=value;
}

void Sudoku::Print(){
  for(int j=0; j<9; j++){
    if( j%3 == 0 ){
      cout<< "-------------------------------" << endl;
    }
    for(int i=0; i<9; i++){
      if( i%3==0 ){
    cout << "|";
      }
      if(board[i][j] !=0){
    cout<<" "<<board[i][j]<<" ";
      } else {
    cout<<" * ";
      }
    }
    cout<<"|"<<endl;
  }
  cout << "-------------------------------" << endl;
}
```

```
bool Sudoku::Solve(int nx, int ny){
  if( board[nx][ny] != 0 ){
    if(verify(nx, ny)){
      if( nx==8 && ny== 8){
    return true;
      }
      int next_x = nx+1;
      int next_y = ny;

      if(next_x >=9 ){
    next_x = 0;
    next_y++;
      }
      return Solve(next_x, next_y);
    } else{
      return false;
    }
  }

  for (int val=1; val<10; val++){
    setBoardValue(nx, ny, val);
    if(verify(nx,ny)){
      if( nx == 8 && ny==8 ){
    return true;
      }
      int next_x = nx+1;
      int next_y = ny;
      if(next_x>=9){
    next_x = 0;
    next_y++;
      }
      if(Solve(next_x,next_y)){
    return true;
      }
    }
  }

  board[nx][ny] = 0;
  return false;
}


bool Sudoku::verify(int nx , int ny ){
  int i, j;

  for(i=nx - (nx%3) ; i< nx-(nx%3) + 3;i++ )
    for(j=ny - (ny%3) ; j<ny-(ny%3)+ 3; j++){
      if ( i==nx && j==ny ) continue;
      if (board[i][j] == board[nx][ny] ) return false;
    }
  for(i=0; i<9; i++){
    if( i == nx ) continue;
    if( board[i][ny] == board[nx][ny] ) return false;
  }
  for(j=0; j<9; j++){
    if( j==ny ) continue;
    if( board[nx][j] == board[nx][ny] ) return false;
  }
  return true;
}


int main(){

  Sudoku puzzle;

  puzzle.setBoardValue(0,0,2);
  puzzle.setBoardValue(1,1,5);
  puzzle.setBoardValue(2,2,9);

  puzzle.setBoardValue(3,2,6);
  puzzle.setBoardValue(4,1,2);
  puzzle.setBoardValue(5,0,7);

  puzzle.setBoardValue(6,2,5);
  puzzle.setBoardValue(7,0,9);
  puzzle.setBoardValue(8,1,8);

  puzzle.setBoardValue(0,5,6);
  puzzle.setBoardValue(1,4,7);
  puzzle.setBoardValue(2,3,5);

  puzzle.setBoardValue(3,3,3);
  puzzle.setBoardValue(4,4,8);
  puzzle.setBoardValue(5,5,4);

  puzzle.setBoardValue(6,3,9);
  puzzle.setBoardValue(8,4,2);

  puzzle.setBoardValue(0,6,3);
  puzzle.setBoardValue(1,7,4);
  puzzle.setBoardValue(2,8,7);

  puzzle.setBoardValue(6,8,3);
  puzzle.setBoardValue(7,6,1);
  puzzle.setBoardValue(8,7,7);

  puzzle.Print();
  cout << endl;

  if(puzzle.Solve()){
    cout<<"Found solution:"<<endl;
    puzzle.Print();
  } else {
    cout <<"Puzzle is ill posed!";
  }
  cout<<endl;
  return 0;
}
```