Problem 1

(a) We add in code to simulate trajectories using Euler-Maruyama method, and count how many times do we get trajectories with ending point greater than 0.5.

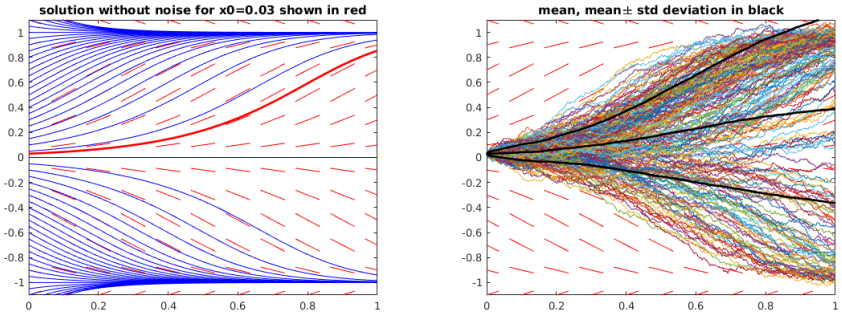

Figure 1: The Deterministic Solution and Simulated Stochastic Solution

Out of 200 Monte Carlo runs, 129 of them has $X(1) > 0.5$. Therefore, we estimate $P[X(1) > 0.5] = \frac{129}{200} = 0.645$

(b) Using forward Komogorov Equation, we can fill in the code

- $F_0(x) = 4(-3x^2 + 1)$
- $F_1(x) = 4(-x^3 + x)$
- $F_2(x) = \frac{b^2}{2}$

The code will give a numerical solution of the density function, as depicted in figure

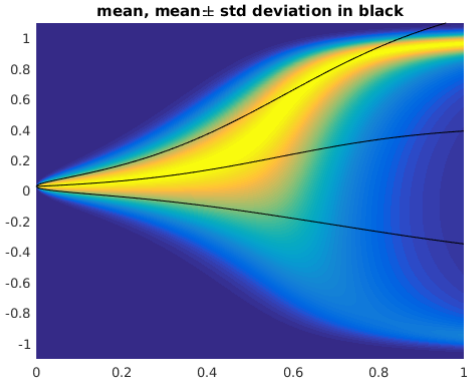

Figure 2: Solution of Density Function from FKE

We add another line of code to calculate $P(x(1) > 0.5)$ from the solution of q

```
disp(sum(q(:,N+1).*(q(:,N+1)>0.5))*dx);
```

This is using a summation to approximate the interval .The result is 0.6720. Which is pretty close to our Monte Carlo solution in part(a).

Problem 2

Approximation 1 The KL expansion gives

$$B_1(t) = \sum_{k=1}^{\infty} \lambda_k^{1/2} Y^{(k)}(\omega) v^{(k)}(t)$$

Then, it is easy to see that

$$E[||B_1||^2_{L^2[0,1]}] = \sum_{k=1}^{\infty} \lambda_k^2 = \sum_{k=1}^{\infty} \frac{1}{k^2\pi^2} = \frac{1}{\pi^2}\frac{\pi^2}{6} = \frac{1}{6} = \frac{1}{6} * \frac{1}{1}$$

The truncation error can be calculated as

$$E[||B_1(t) - X_N||^2_{L^2[0,1]}] = \sum_{k=N}^{\infty} \lambda_k = \sum_{k=N}^{\infty} \frac{1}{\pi^2 k^2}$$

We only need find C, $\alpha = 1$ to be a sharp bound.For $N > 1$, assuming $C, \alpha = 1$ is sharp.

$$\begin{aligned}
&\sum_{k=N}^{\infty} \frac{1}{\pi^2 k^2} \le C\frac{1}{N} \\
\Rightarrow &\frac{1}{N^2\pi^2} + \sum_{k=N+1}^{\infty} \frac{1}{k^2\pi^2} \le C\frac{1}{N+1}\frac{N+1}{N} \\
\Rightarrow &\sum_{k=N+1}^{\infty} \frac{1}{k^2\pi^2} \le C\frac{1}{N+1}(\frac{N+1}{N} - \frac{N+1}{CN^2\pi^2}) \\
\Leftrightarrow &\frac{1+N}{N} - \frac{1+N}{CN^2\pi^2} \ge 1 \\
\Leftrightarrow &C \ge \frac{1}{\pi^2}
\end{aligned}$$

Since for $N = 1$, we get $C = \frac{1}{6} > \frac{1}{\pi^2}$. Then from above derivation, we know that $C = \frac{1}{6}, \alpha = 1$ holds for $N > 1$

Approximation 2 We know that, on each subinterval $[\frac{K-1}{N}, \frac{K}{N}]$, we got a Brownian Bridge. We first look at this small Brownian Bridge

$$\begin{aligned} C_h &= E[||B_h||^2_{L^2[0,h]}] \\ &= E[\int_0^h ||B_h(t)||^2 dt] \\ &= E[\int_0^1 ||h^{1/2}B_1(y)||^2 h dy], h^{1/2}y := t \\ &= h^2 E[||B_1(t)||^2_{L^2[0,1]}] \\ &= \frac{h^2}{6} \end{aligned}$$

Therefore, combining all the "small bridge", we would get

$$C_2 = N * C_h = N * h^2 * \frac{1}{6} = N * (1/N)^2 * \frac{1}{6} = \frac{1}{6}\frac{1}{N}$$

Therefore, again we would get $C = \frac{1}{6}, \alpha = 1$, for the Brownian Bridge approximation.

Problem 3

If we get the basis function of the form

$$\phi_i(x)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)$$

Then, our approximate solution from the SG method is of the form

$$u(x, y_1, y_2) = \sum_{i,j_1,j_2} U_{i,j_1,j_2}[\phi_i(x)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)]$$

The random component comes from y_1, y_2 in our case, therefore, we can get

$$\begin{aligned} E[u(x_1)] &= E[\sum_{i,j_1,j_2} U_{i,j_1,j_2}[\phi_i(x_1)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)]] \\ &= \sum_{i,j_1,j_2} U_{i,j_1,j_2}\phi_1(x_1)E[\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)] \end{aligned}$$

Since the basis functions ϕ and ψ are pre-chosen, those integrals can be evaluated in closed form.

Similarly, we get

$$\begin{aligned}\mathrm{Var}(u(x_1)) &= E[u(x_1)^2] - E[u(x_1)]^2 \\ &= E[\sum_{i,j_1,j_2} U_{i,j_1,j_2}[\phi_i(x_1)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)]]^2 - (E[u(x_1)])^2 \\ \mathrm{Cov(u(x_1), u(x_2))} &= E[\sum_{i,j_1,j_2} U_{i,j_1,j_2}[\phi_i(x_1)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)] \sum_{i,j_1,j_2} U_{i,j_1,j_2}[\phi_i(x_2)\psi_{j_1-1}(y_1)\psi_{j_2-1}(y_2)]] \\ &\quad - E(u(x_1))E(u(x_2))\end{aligned}$$

Those expectations can be distributed, and computed using integrals of ψ, and functional values of $\phi(x_1), \phi(x_2)$. In the code, those integrals are known constants, and the summation is calculated using vector inner products.

```
% find expectation
Eu = Ua(:,1)*sqrt(2)^n*2^-n;
% sqrt(2)^n since sqrt(2) = integral(psi_0(y_j),y_j=-1..1)
% 2^(-n)    since E[f(y)] = integral f(y) 2^(-n) dy_1 ... dy_n

x = (0:N)'/N;

% find covariance
V = Ua(:,2:end);
Cu = V*V'*2^-n;
```

This is the code for the calculation.

Problem 4

(a) Use sample mean and sample variance to get the estimate for $u(0.3)$ and its confidence interval.
Use sample percentage to estimate $P(u(0.8)) > 0.05$
Use sample variance to estimate $C(0.3, 0.8)$

Code is

```
% print out information for u(0.3)
disp('******* OUTPUT for PART(a)********');
fprintf('u(0.3) =%f \n',Eu(0.3*N));
fprintf('0.95 Confidence Interval for u(0.3): (%f,%f) \n',Eu(0.3*N)-eu(0.3*N),Eu(0.3
fprintf('Convariance c(0.3,0.8) =%f  \n',Cu(0.3*N,0.8*N));
fprintf('P[u(0.8)>0.05] = %f  \n', count8/M );
disp('****** PART(a) COMPLETED *********');
```

By tuning the arguments n, N, M. We found that *maxeu* decrease pretty fast with M. But the result seem to be stable for not too small values of n and N. We choose $n = 10$,$N = 200$,$M = 1000$. The output is

```
******* OUTPUT for PART(a)********
u(0.3) =0.066980
0.95 Confidence Interval for u(0.3): (0.066408,0.067553)
Convariance c(0.3,0.8) =-0.000046
P[u(0.8)>0.05] = 0.440000
****** PART(a) COMPLETED *********
```

(b) Following the hint, we use Monte Carlo method to evaluate the integral

$$P(u(0.8) > .5)$$

The basic procedure is

- Simulate $y_1, y_2, ..., y_n$, independently from $Unif[-1, 1]$
- Evaluate $\psi_1(y), ..., \psi_k(y)$ using the code.
- For this particular simulated sample, count if $u(0.8) = \sum_{i=1}^{N_h} z_i \phi_i(x) > 0.05$
- Repeat the above procedure for M times, and estimate the probability to be n/M. With $n =$ number of times with $u(0.8) > 0.05$

Remark: since we are using Nodal Basis. Therefore $\phi_i(x_j) = \delta_{ij}$. Therefore, we can evaluate $u(0.0.8) = Ua(0.8, :) * z(index(0.8))$.

The code is modified as

```
% display the result for part (b)
disp('******* OUTPUT for PART(b) ********')
fprintf('u(0.3) = %f\n ', Eu(0.3*N) );
fprintf('C(0.3,0.8) = %f\n ', Cu(0.3*N,0.8*N));

% calculate p(u(0.8)>.05) using monte carlo method
count = 0;
for q = 1:1000
    V = zeros(p+1,n);
    % simulate y1,..,yn  add by G.S.
    y = 2*rand(1,n)-1;
    for k=0:p
        a = legendre(k,y,'norm');
        a = a(1,:); % a = [psi_k(y1),...psi_k(yn)];
        V(k+1,:) = a;
    end
```

```
    V = tensp(V);
    V = V(:);
    z = Ua(0.8*N,:)*V;
    count = count + (z>0.05);
end
fprintf('P(u(0.8)>0.05) = %f\n ',count/1000);
disp('******* PART(b) COMPLETED *********');
```

We choose the arguments for return reasonably consistent result in acceptable time. $n = 3$,$N = 100$,$p = 3$. One sample of output would be

```
******* OUTPUT for PART(b) ********
u(0.3) = 0.066735
C(0.3,0.8) = -0.000046
P(u(0.8)>0.05) = 0.446000
******* PART(b) COMPLETED *********
```

We can see that results from SG and Monte Carlo methods are quite consistent.

(c) We experiment with different values of arguments to study this problem. For simplicity, we disable all plotting functionality of the functions.

- $N = 10$,$p = 3$. How will n affect running time for SG-FEM method? As show

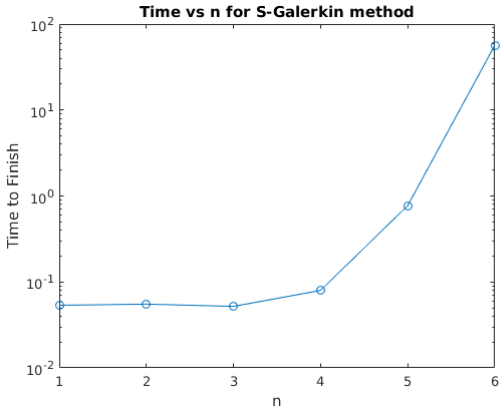

Figure 3: Time vs n semilogy plot

in figure 3. Running time will quickly blow up as n increases. At $n = 7$, the memory exceeds the machine limit. Because we used direct solver in matlab.

- $N = 10$,$M = 100$, for Monte Carlo Methods. Monte Carlo methods, on the other hand, scales very well with n. It is a very slow linear growth in calculation time. As we can see from figure 4.

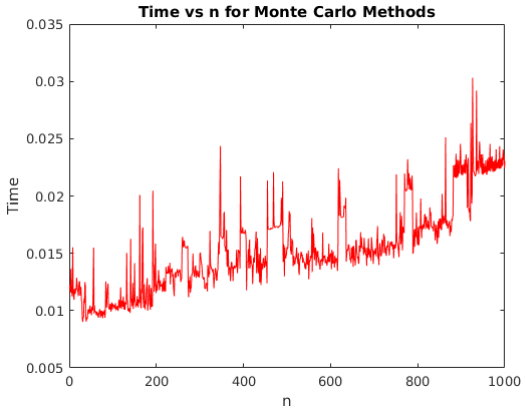

Figure 4: Monte Carlo Methods

– Accuracy Comparison Between the Two Methods: We first establish a baseline for measuring the "accuracy" of the two methods.
Using $N = 10.n = 1000, M = 100000$. We use an extensive Monte Carlo run to estimate $u(0.3)$ as the "correct" probability. And see how well the two methods perform in getting that probability. From figure 5, we can see

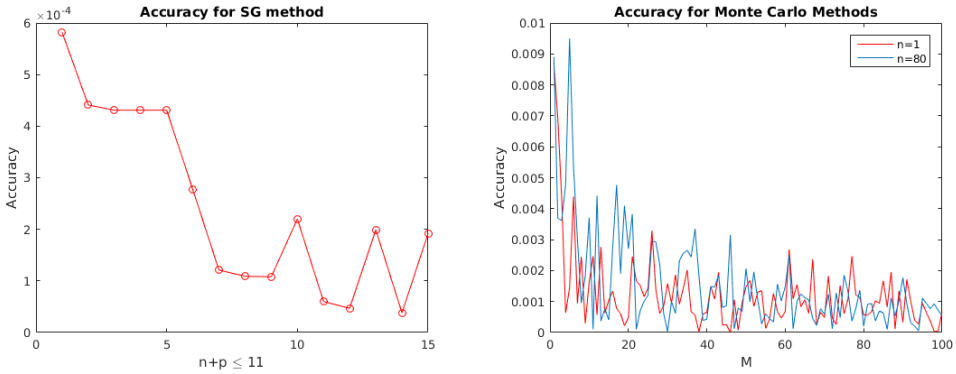

Figure 5: Accuracy of both Methods

that Stochastic Galerkin method is more expensive to run. But the accuracy is orders of magnitude better than Monte Carlo methods. It is much more stable.
Monte Carlo methods, is much faster to run. But the performance do not really increase with p. At least in our 1 100 trial, not significant improvement can be seen. And the performance also improves rather slowly with increasing M.

Because Monte Carlo method is fast to run. And because SG can only process

very small values of $n + N + p$. Therefore, we try altering N only for Monte Carlo method. Again, from

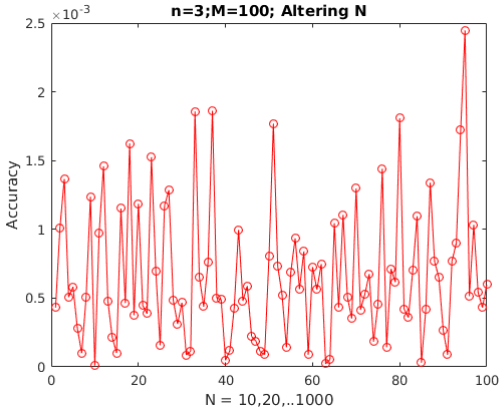

Figure 6: N Tuning for Monte Carlo Methods

We can see from figure 6. Increasing N still do not necessarily increase the performance of Monte Carlo method for our test problem.

Conclusion: Monte Carlo methods is much faster to run, offers larger space for tuning inputs, but the result is generally less accurate, or stable.