Monte Carlo simulation of Heston model

Aurélien Perez & Tina Truong, 2025

Heston model

$$dS_t = rS_t dt + \sqrt{v_t} S_t d\hat{Z}_t \tag{1}$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t \tag{2}$$
$$\hat{Z}_t = \rho W_t + \sqrt{1-\rho^2} Z_t \tag{3}$$

where:

- the spot values $S_0 = 1$ and $v_0 = 0.1$,
- r is the risk-free interest rate, we assume $r = 0$,
- κ is the mean reversion rate of the volatility,
- θ is the long-term volatility,
- σ is the volatility of volatility,
- W_t and Z_t are independent Brownian motions

Contents

I. GPU methods

Parallel Execution 👬👬👬👬

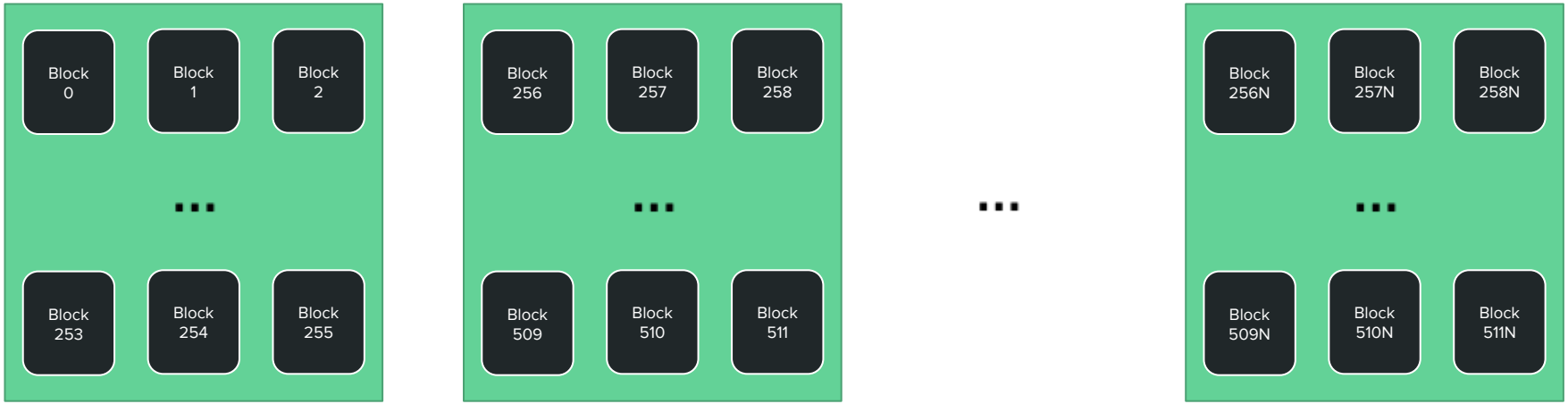
For each parameter set **κ, θ, σ,** we simulate 2^18 trajectories
1024 Threads per block

Each thread:
- Simulates 1 trajectory (Euler, Exact or Almost Exact scheme)
- Computes discounted payoff
- Stores result in shared memory (R1s, R2s)

Shared memory reduction per block (sum of 1024 payoffs)
Thread 0 of block → atomicAdd to:
- d_sum[triplet_id]
- d_sum2[triplet_id]

Final payoff mean and variance (computed outside the kernel)

$(\kappa_0, \theta_0, \sigma_0)$ $(\kappa_1, \theta_1, \sigma_1)$ ... $(\kappa_N, \theta_N, \sigma_N)$



atomicAdd to d_sum[0], d_sum2[0]

atomicAdd to d_sum[1], d_sum2[1]

atomicAdd to d_sum[N], d_sum2[N]

Parallel Execution 👬👬👬👬

```
HestonParam *d_params;
testCUDA(cudaMalloc(&d_params, n_triplets * sizeof(HestonParam)));
testCUDA(cudaMemcpy(d_params, h_params, n_triplets * sizeof(HestonParam), cudaMemcpyHostToDevice));
free(h_params);


float *d_sum, *d_sum2;
testCUDA(cudaMallocManaged(&d_sum, n_triplets * sizeof(float)));
testCUDA(cudaMallocManaged(&d_sum2, n_triplets * sizeof(float)));
for (int i = 0; i < n_triplets; i++){
    d_sum[i] = 0.0f;
    d_sum2[i] = 0.0f;
}
```

Local Randomness 🎲

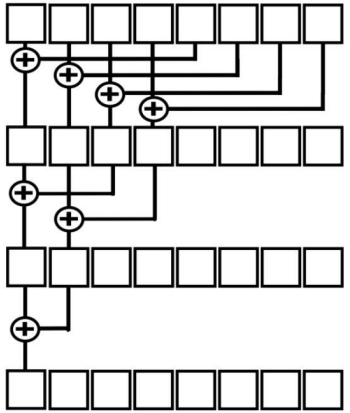
```
// Set the state for each thread
__global__ void init_curand_state_k(curandState* state) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    curand_init(0, idx, 0, &state[idx]);
}
```

thread	0	1	…	2^18-1	2^18	…	2*(2^18)-1
seed	0	0	…	0	0	…	0
sequence	0	1	…	2^18-1	2^18	…	2*(2^18)-1
params	$(\kappa_0, \theta_0, \sigma_0)$	$(\kappa_0, \theta_0, \sigma_0)$	…	$(\kappa_0, \theta_0, \sigma_0)$	$(\kappa_1, \theta_1, \sigma_1)$	…	$(\kappa_1, \theta_1, \sigma_1)$

Block-Level Reduction 🧊🗡️

```
// reduction
__syncthreads();
int i = blockDim.x / 2;
while (i!=0){
    if (threadIdx.x < i){
        R1s[threadIdx.x] += R1s[threadIdx.x + i];
        R2s[threadIdx.x] += R2s[threadIdx.x + i];
    }
    __syncthreads();
    i /= 2;
}

if (threadIdx.x == 0){
    atomicAdd(&d_sum[triplet_idx], R1s[0]);
    atomicAdd(&d_sum2[triplet_idx], R2s[0]);
}
```



1024 threads

512 threads

...

2 threads

1 thread

Atomic add to pass from GPU to CPU and avoid collision

II. Euler method

I. Euler method

$$S_{t+\Delta t} = S_t + rS_t\Delta t + \sqrt{v_t}\, S_t \sqrt{\Delta t}\left(\rho G_1 + \sqrt{1-\rho^2} G_2\right)$$

$$v_{t+\Delta t} = g\left(v_t + \kappa(\theta - v_t)\Delta \mathrm{t} + \sigma \sqrt{v_t \Delta t} G_1\right)$$

```
// Euler Scheme kernel
__global__ void MC_Heston_Euler_kernel(float S_0, float v_0, float r, float rho, float sqrt_dt,
    float K, int N, curandState *state, float *d_sum, float *d_sum2, int n_triplets, HestonParam *d_params) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_threads = n_triplets * N_TRAJ;
    if (idx >= total_threads) return;
    int triplet_idx = idx >> POW_TRAJ;
    if (triplet_idx >= n_triplets) return;

    __shared__ HestonParam p_shared;
    if (threadIdx.x == 0) p_shared = d_params[triplet_idx];
    __syncthreads();

    HestonParam p = p_shared;
    float kappa = p.kappa, theta = p.theta, sigma = p.sigma;
    float dt = sqrt_dt * sqrt_dt;
    float S = S_0;
    float v = v_0, v_next;
    curandState localState = state[idx];

    extern __shared__ float A[];
    float* R1s, * R2s;
    R1s = A;
    R2s = R1s + blockDim.x;

    for (int i = 0; i < N; i++){
        float2 G = curand_normal2(&localState);
        v_next = fmaxf(v + kappa*(theta - v)*dt + sigma * sqrtf(v) * sqrt_dt * G.x, 0.0f);
        S += r * S * dt + sqrtf(v) * S * sqrt_dt * (rho * G.x + sqrtf(1.0f - rho*rho) * G.y);
        v = v_next;
    }
    R1s[threadIdx.x] = expf(-r * dt * N) * fmaxf(0.0f, S-K)/N_TRAJ ;
    R2s[threadIdx.x] = R1s[threadIdx.x] * R1s[threadIdx.x] * N_TRAJ;
```

III. Exact method

II. Exact method

step 1. Within a for loop on time steps, we define

$$d = 2\kappa\theta/\sigma^2, \quad \lambda = \frac{2\kappa e^{-\kappa\Delta t} v_t}{\sigma^2(1-e^{-\kappa\Delta t})}, \quad N = \mathcal{P}(\lambda), \quad \text{with } \mathcal{P} \text{ simulated by } \texttt{curand\_poisson}$$

and denoting $\mathcal{G}(\alpha)$ the standard gamma distribution whose simulation is presented in [6]

$$v_{t+\Delta t} = \frac{\sigma^2(1-e^{-\kappa\Delta t})}{2\kappa}\mathcal{G}(d+N).$$

step 2. The integral $\int_0^1 v_s ds$ is stored in a variable `vI` set to zero before the for loop on time steps. Then, in the for loop we update `vI+=`$0.5 * (v_t + v_{t+\Delta t})\Delta t$.

step 3. Once we finish the for loop, we compute $\int_0^1 \sqrt{v_s} dW_s$, using the expression

$$\int_0^1 \sqrt{v_s} dW_s = \frac{1}{\sigma}\left(v_1 - v_0 - \kappa\theta + \kappa \texttt{vI}\right)$$

Then we compute

$$m = -0.5\texttt{vI} + \rho \int_0^1 \sqrt{v_s} dW_s, \quad \Sigma^2 = (1-\rho^2)\texttt{vI}$$

and we set $S_1 = \exp(m + \Sigma G)$ where G is a standard normal random variable independent from (G_1, G_2).

```cpp
__device__ float rgamma_sup1(curandState *state, float alpha) {
    float d = alpha - 1.0f / 3.0f;
    float c = 1.0f / sqrtf(9.0f * d);
    float z, u, x, v;
    while (true) {
        z = curand_normal(state);
        u = curand_uniform(state);
        x = 1.0f + c * z;
        v = x * x * x;
        if (z > -1.0f / c && logf(u) < (0.5f * z * z + d - d * v + d * logf(v)))
            return d * v;
    }
}

__device__ float rgamma_inf1(curandState *state, float alpha) {
    float u = curand_uniform(state);
    return rgamma_sup1(state, alpha + 1.0f) * powf(u, 1.0f / alpha);
}

// Gamma distribution sampling (Marsaglia-Tsang)
__device__ float rgamma(curandState *state, float alpha) {
    if (alpha < 1.0f) {
        return rgamma_inf1(state, alpha);
    }
    else{
        return rgamma_sup1(state, alpha);
    }
}
```

```
// Exact scheme kernel (Broadie-Kaya)
__global__ void MC_Heston_Exact_kernel(float S_0, float v_0, float r, float rho, float sqrt_dt,
    float K, int N, curandState *state, float *d_sum, float *d_sum2, int n_triplets, HestonParam *d_params) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_threads = n_triplets * N_TRAJ;
    if (idx >= total_threads) return;
    int triplet_idx = idx >> POW_TRAJ; // idx / 2^POW_TRAJ
    // idx          0 1 2 3 4 ... 2^18 - 1  2^18 2^18+1 ...
    // triplet_idx  0 0 0 0 0 ...     0       1     1     ...
    if (triplet_idx >= n_triplets) return;

    __shared__ HestonParam p_shared;
    if (threadIdx.x == 0) p_shared = d_params[triplet_idx];
    __syncthreads();

    HestonParam p = p_shared;
    float kappa = p.kappa, theta = p.theta, sigma = p.sigma;
    float sigma2 = sigma * sigma;
    float dt = sqrt_dt * sqrt_dt;
    float d = 2.0f * kappa * theta / sigma2;
    float v = v_0, v_next, vI = 0.0f, S = S_0;
    curandState localState = state[idx];

    for (int i = 0; i < N; i++) {
        float lambda = (2.0f * kappa * expf(-kappa * dt) * v) / (sigma2 * (1.0f - expf(-kappa * dt)));
        int N_pois = curand_poisson(&localState, lambda);
        float gamma = rgamma(&localState, d + N_pois);
        v_next = (sigma2 * (1.0f - expf(-kappa * dt)) / (2.0f * kappa)) * gamma;
        vI += 0.5f * dt * (v_next + v);
        v = v_next;
    }
    float m = -0.5f * vI + (rho / sigma) * (v - v_0 - kappa * theta + kappa * vI);
    float Sigma = sqrtf((1.0f - rho * rho) * vI);
    float2 G = curand_normal2(&localState);
    S = S_0 * expf(m + Sigma * G.x);

    extern __shared__ float A[];
    float* R1s, * R2s;
    R1s = A;
    R2s = R1s + blockDim.x;

    R1s[threadIdx.x] = expf(-r * dt * N) * fmaxf(0.0f, S-K)/N_TRAJ ;
    R2s[threadIdx.x] = R1s[threadIdx.x] * R1s[threadIdx.x] * N_TRAJ;
```

IV. Almost exact method

III. Almost exact method

$$\log(S_{t+\Delta t}) = \log(S_t) + k_0 + k_1 v_t + k_2 v_{t+\Delta t} + \sqrt{(1-\rho^2) v_t \Delta t}\left(\rho G_1 + \sqrt{1-\rho^2} G_2\right)$$

with

$$k_0 = \left(-\frac{\rho}{\sigma}\kappa\theta\right)\Delta t$$

$$k_1 = \left(\frac{\rho\kappa}{\sigma} - 0.5\right)\Delta t - \frac{\rho}{\sigma}$$

$$k_2 = \frac{\rho}{\sigma}$$

where v_t is simulated as in the exact scheme.

```
// Almost Exact Scheme kernel (Haastrecht-Pelsser)
__global__ void MC_Heston_Almost_Exact_kernel(float S_0, float v_0, float r, float rho, float sqrt_dt,
    float K, int N, curandState *state, float *d_sum, float *d_sum2, int n_triplets, HestonParam *d_params) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_threads = n_triplets * N_TRAJ;
    if (idx >= total_threads) return;
    int triplet_idx = idx >> POW_TRAJ;
    if (triplet_idx >= n_triplets) return;

    __shared__ HestonParam p_shared;
    if (threadIdx.x == 0) p_shared = d_params[triplet_idx];
    __syncthreads();

    HestonParam p = p_shared;
    float kappa = p.kappa, theta = p.theta, sigma = p.sigma;
    float sigma2 = sigma * sigma;
    float dt = sqrt_dt * sqrt_dt;
    float d = 2.0f * kappa * theta / sigma2;
    float S = logf(S_0);
    float v = v_0;
    float v_next;
    float rhosigma = rho / sigma;
    float k0 = (-rhosigma * kappa * theta) * dt;
    float k1 = (rhosigma * kappa  - 0.5f) * dt - rhosigma;
    float k2 = rhosigma;
    curandState localState = state[idx];

    for (int i = 0; i < N; i++){
        float2 G = curand_normal2(&localState);
        float lambda = (2.0f * kappa * expf(-kappa * dt) * v) / (sigma2 * (1.0f - expf(-kappa * dt)));
        int N_pois = curand_poisson(&localState, lambda);
        float gamma = rgamma(&localState, d + N_pois);
        v_next = (sigma2 * (1.0f - expf(-kappa * dt)) / (2.0f * kappa)) * gamma;
        S += k0 + k1 * v + k2 * v_next + sqrtf((1.0f - rho*rho) * v) * sqrt_dt * (rho * G.x + sqrtf(1.0f - rho*rho) * G.y);
        v = v_next;
    }
    S = expf(S);

    extern __shared__ float A[];
    float* R1s, * R2s;
    R1s = A;
    R2s = R1s + blockDim.x;
    R1s[threadIdx.x] = expf(-r * dt * N) * fmaxf(0.0f, S-K)/N_TRAJ ;
    R2s[threadIdx.x] = R1s[threadIdx.x] * R1s[threadIdx.x] * N_TRAJ;
```

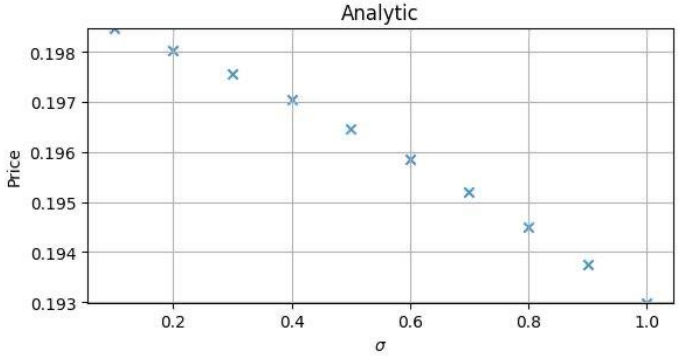
V. Results comparison

Execution speed / triplet

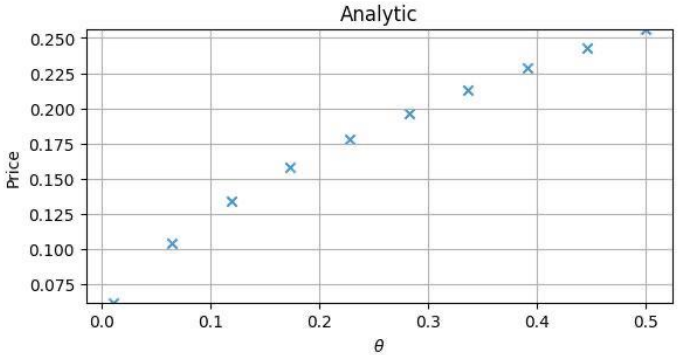
Exact	436.7 ms
Almost Exact	437.01 ms
Euler	5.4 ms (87x)

Analytical prices by inverting Fourier transform

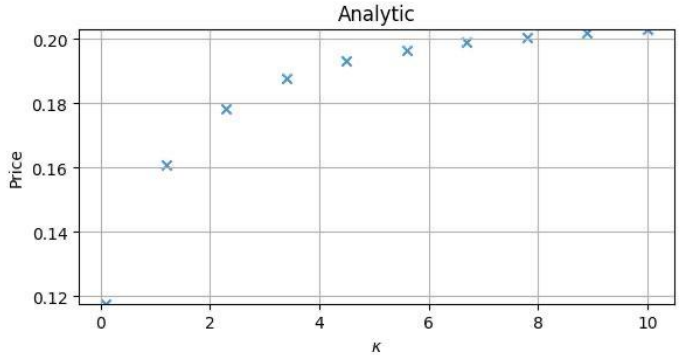
Call Price Sensitivity to σ (fixed $\kappa \approx 5.60$, $\theta \approx 0.28$)

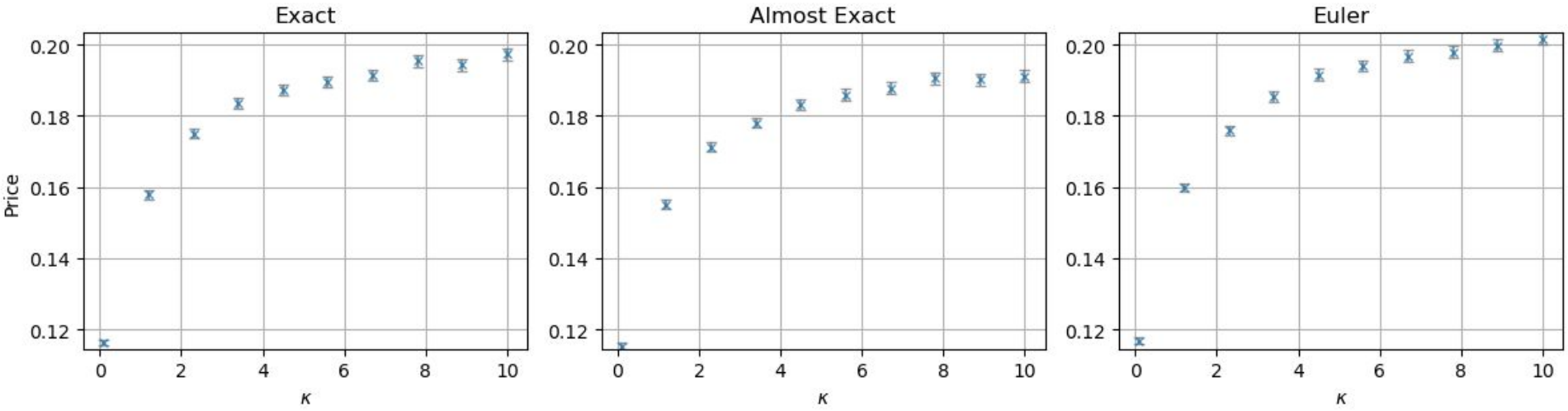


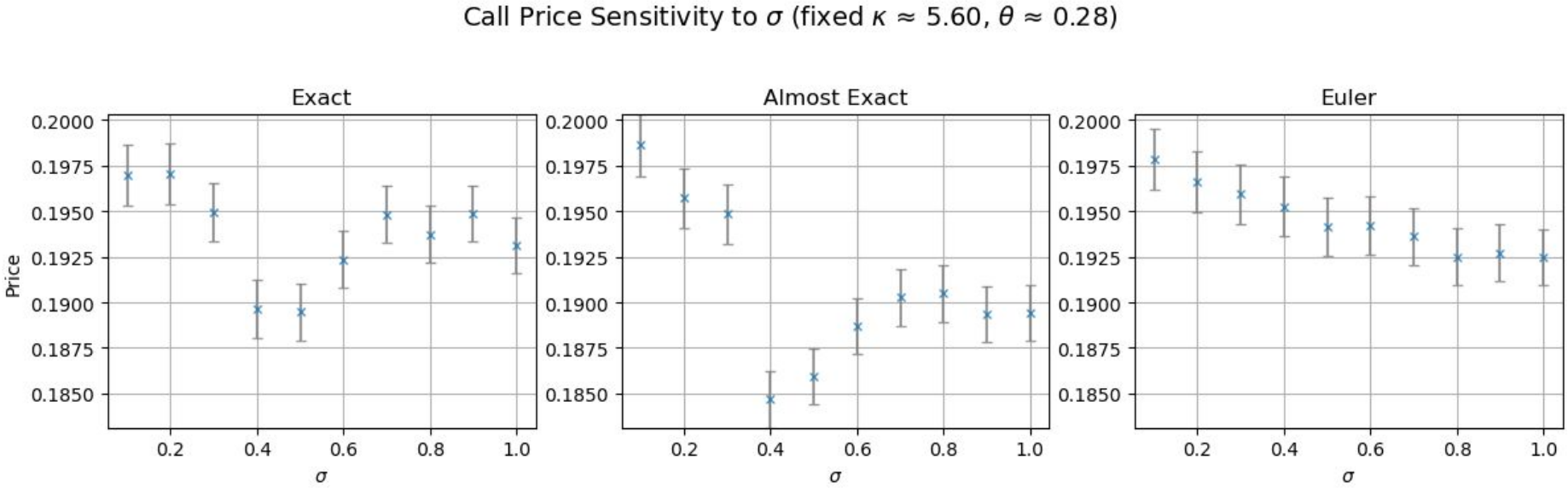
Call Price Sensitivityto θ (fixed $\kappa \approx 5.60$, $\sigma \approx 0.50$)

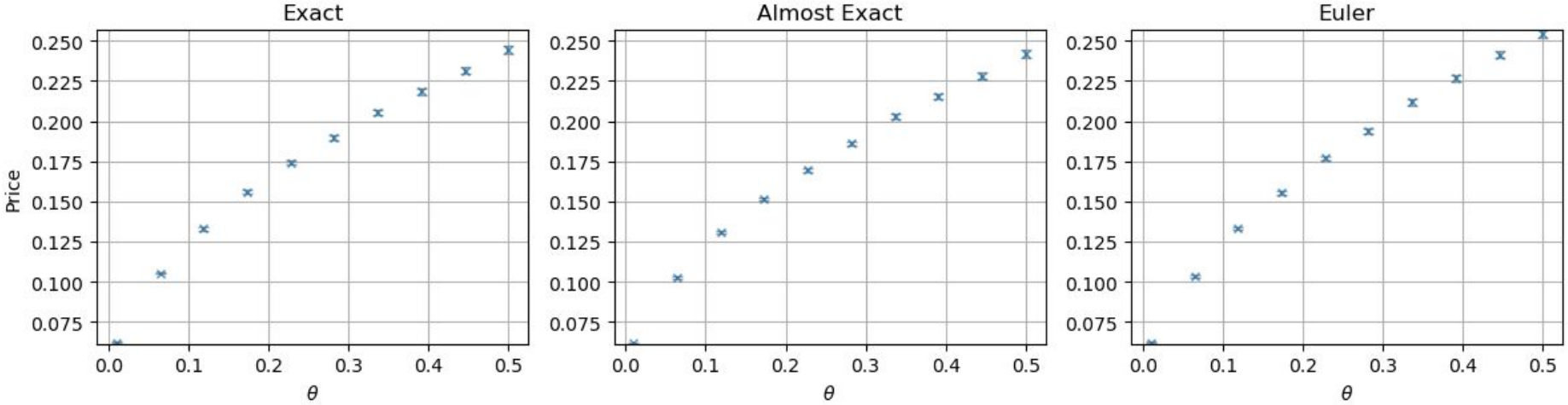


Call Price Sensitivity to κ (fixed $\theta \approx 0.28$, $\sigma \approx 0.50$)




Call Price Sensitivity to κ (fixed $\theta \approx 0.28$, $\sigma \approx 0.50$)
Exact
Almost Exact
Euler
Price
0.20
0.18
0.16
0.14
0.12
0
2
4
6
8
10
κ


Call Price Sensitivity to σ (fixed κ ≈ 5.60, θ ≈ 0.28)
Exact
Almost Exact
Euler
Price
σ
0.2000
0.1975
0.1950
0.1925
0.1900
0.1875
0.1850
0.2
0.4
0.6
0.8
1.0


Call Price Sensitivity to θ (fixed $\kappa \approx 5.60$, $\sigma \approx 0.50$)
Exact
Almost Exact
Euler
Price
θ
0.250
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.0
0.1
0.2
0.3
0.4
0.5

Conclusion

Euler is fast and accurate.