A Tape-free AAD Tool for C++, and a Finite Volume Method for SLV Calibration

January 2017

Jacques du Toit
Johannes Lotz
Klaus Leppkes
Maarten Wyns

nag®

Experts in numerical software and High Performance Computing

The Numerical Algorithms Group

- Not for profit company that is 43 years old
- Products
  - Compiled numerical library accessible from many environments
  - Algorithmic Differentiation tools
- Services
  - Bespoke code development
  - HPC code development, profiling and tuning
  - HPC mentoring and procurement
  - AD consulting

Introduction to AD

Introduction to AD

- What is Algorithmic Differentiation?

Introduction to AD

- What is Algorithmic Differentiation?
- Why is it useful in finance?

Introduction to AD

- What is Algorithmic Differentiation?
- Why is it useful in finance?
- What is an adjoint?

Forward Mode AD by Example

Suppose we have a function

```
void foo(double x1, double x2, double &y)
{
    double a = x1*x2;
    double b = sin(a) * x1;
    y = b * exp(a);
}
```

Forward Mode AD by Example

Suppose we have a function

```
void foo(double x1, double x2, double &y)
{
   double a = x1*x2;
   double b = sin(a) * x1;
   y = b * exp(a);
}
```

We can compute derivative with respect to x_1 like so

```
void d_foo_d_x1(double x1, double x2, double &y, double &dy_dx1)
{
   double a = x1*x2;
   double da_dx1 = x2;
   double b = sin(a) * x1;
   double db_dx1 = cos(a)*x1*da_dx1 + sin(a);
   y = b*exp(a);
   dy_dx1 = exp(a)*db_dx1 + b*exp(a)*da_dx1;
}
```

Forward/Tangent Mode AD

Can we make a tool do to this for us? Yes!

- Make a new class `adouble` which holds value and derivative
- Overload operators $+, \times, -, /$ and math functions $\sin, \cos, \ldots$ to compute value and update derivative via chain rule
- Instead of computing with `double` we compute with `adouble`

This tool will in fact compute the *forward/tangent model of AD*

$$y = F(x)$$

$$y^{(1)} = \left[\frac{\partial F}{\partial x} \right] x^{(1)} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} x^{(1)}$$

Same computational cost as finite differences, but more accurate

Adjoint Mode AD

Adjoint mode AD computes

$$y = F(x)$$

$$x_{(1)} = \left[\frac{\partial F}{\partial x} \right]^T y_{(1)} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial F_1}{\partial x_n} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} y_{(1)}$$

- Note this maps *output space* to *input space*
- If $y \in \mathbb{R}$, get entire gradient in one call of adjoint model regardless of input dimension
- Can prove will require at most 5 times more flops than F
- But there's a catch

Adjoint Mode AD

If $x \xrightarrow{F} x_1 \xrightarrow{G} x_2 \xrightarrow{H} y$ then the adjoint model computes

$$\left[\frac{\partial y}{\partial x}\right]^T y_{(1)} = \left[\frac{\partial H}{\partial x_2}\frac{\partial G}{\partial x_1}\frac{\partial F}{\partial x}\right]^T y_{(1)}$$

$$= \left[\frac{\partial F}{\partial x}\right]^T \left[\frac{\partial G}{\partial x_1}\right]^T \left[\frac{\partial H}{\partial x_2}\right]^T y_{(1)}$$

- Most efficient way to accumulate this is *backwards*
- Means need to run program forwards, then backwards
- Can we make a tool to do this?
- Yes! Use same ideas as before, but now need a *tape*

Tape Based Adjoint AD

- As before, compute with `adouble`
- As code runs, `adouble` writes all intermediate calculations to tape
- When reaches end of code, tape played backwards

Tape Based Adjoint AD

- As before, compute with `adouble`
- As code runs, `adouble` writes all intermediate calculations to tape
- When reaches end of code, tape played backwards

Memory

Tape Based AAD and GPUs

Idea of tape doesn't really fit well with GPUs

- GPUs have about 16GB memory, CPUs easily have 64GB
- GPUs have 50,000 threads (0.32MB per thread), CPUs have 32 threads (2,000MB per thread)
- Each thread needs its own tape

Tape on GPU likely to lead to inefficient memory accesses anyway

So what can be done?

C++11 to the Rescue

C++ is not one, but three langauges

- Procedural language of C
- Object oriented language
- Meta-programming language (templates)

Template meta-programming language is Turing complete

C++11 to the Rescue

- Any C++ program consists of *staight line code* plus control flow
- We saw that tangent AD of simple straight line code is completely algorithmic
- Same is true for adjoint AD of simple straight line code

C++11 to the Rescue

- Any C++ program consists of *staight line code* plus control flow
- We saw that tangent AD of simple straight line code is completely algorithmic
- Same is true for adjoint AD of simple straight line code

Can we not write a meta-program which would create the adjoint for a block of straight line code?

C++11 to the Rescue

- Any C++ program consists of *staight line code* plus control flow
- We saw that tangent AD of simple straight line code is completely algorithmic
- Same is true for adjoint AD of simple straight line code

Can we not write a meta-program which would create the adjoint for a block of straight line code?

Turing complete $\neq$ straightforward or **practical**

C++11 to the Rescue

Before C++11 basically not possible

C++11 introduces the keyword **auto**

```
auto n = 10;          // n is int
auto x = 10.0;        // x is double
auto y = bar(n, x);   // y is whatever bar returns
```

Tells the compiler to figure out the type

This is a big deal!

- C++ types are incredibly powerful
- Fundamental blocks in template meta-programming language

Implemented this `dco/map` (Meta Adjoint Programming)

dco/map

Our previous function re-written to use dco/map

```
template<class Active>
void foo(const Active &x1, const Active &x2, Active &y)
{
   const auto a = x1*x2;
   const auto b = sin(a) * x1;
   y = b * exp(a);
}
```

This code can now be run in tangent or adjoint mode

dco/map

Our previous function re-written to use dco/map

```
template<class Active>
void foo(const Active &x1, const Active &x2, Active &y)
{
   const auto a = x1*x2;
   const auto b = sin(a) * x1;
   y = b * exp(a);
}
```

This code can now be run in tangent or adjoint mode

```
usign Active = dco_map::ga1s<double>::type;
Active x1 = 3, x2 = 17;
Active y1;
// Set y_(1) = 1
dco_map::derivative(y1) = 1;

foo(x1, x2, y1);
```

```
template<class Active>
void foo(int n, const Active x[], Active &y)
{
   const auto a = sin(x[0])*x[1];

   Active a, b(0), c, d;
   MAP_FOR(Active, i, 2, n-1, 1) {
      b += x[i];
   } MAP_FOR_END;

   // Call function bar to compute c
   MAP_CALL(Active, bar(a, b, c));

   // Use an if-statement to compute d
   MAP_IF(Active, b < c) {
      d = exp(a)*b;
   } MAP_ELSE {
      d = c*cos(b-a);
   } MAP_IF_END;

   y = d*c*b*a;
}
```

dco/map is C++11

dco/map works with any C++11 compiler, not just GPUs

- Adjoint runtimes are better than tape-based tools
- With some compilers we approach speed of handwritten adjoints

Synthetic test: Harmonic function from computer vision application

Linux			
	gcc5.4	clang3.6	nvcc
pass.	113	40	1,030
hand	244 (2x)	100 (2.5x)	3,407 (3.3x)
dco/map	563 (5x)	103 (2.5x)	1,270 (1.2x)
tape	1,180 (11x)	340 (8.5x)	N/A

dco/map on Local Volatility Monte Carlo Kernel

Linux				
	gcc	clang	icc	nvcc
pass.	1,406	1,461	1,530	18
hand	2,800 (2x)	2,997 (2x)	2,580 (1.7x)	89 (4.9x)
dco/map	3,025 (2.2x)	3,031 (2x)	7,560 (5x)	83 (4.6x)
tape	11,011 (7x)	13,579 (9.3x)	12,520 (8x)	N/A
Windows				
	VS2013	clang	Intel2015	nvcc
pass.	1,510	1,172	1,421	20
hand	1,992 (1.3x)	1,906 (1.6x)	1,876 (1.3x)	90 (4.5x)
dco/map	10,241 (7x)	4,384 (3.7x)	11,671 (8x)	85 (4.2x)
tape	24,000 (16x)	16,125 (16x)	18,833 (13x)	N/A

dco/map on Prototype XVA Application

Not an XVA code, but is very close

- Netting set of one swap
- G2++ two factor interest rate model
- Time-dependent parameters calibrated to yield curve
- Complex analytic formulae for bond prices
- Added spread to forward LIBOR/EURIBOR rates
- No handwritten adjoints (possible, but not fun)

Times-to-solution for full (heterogeneous) application

	gcc	clang	cl	nvcc
pass.	1,650	2,070	900	184 (GPU=15)
dco/map	5,024 (3x)	5,370 (2.6x)	24,590 (27x)	370 (2x) (GPU=111)
tape	12,440 (7.5x)	9,310 (4.5x)	23,600 (26x)	N/A

SLV Calibration

Stochastic Local Volatility Models

- Consider SLV models of the form

$$dX_\tau = \left(r_d - r_f - \frac{1}{2}\sigma_{SLV}^2(X_\tau,\tau)\psi^2(V_\tau)\right)d\tau + \sigma_{SLV}(X_\tau,\tau)\psi(V_\tau)dW_\tau^{(1)},$$

$$dV_\tau = \kappa(\eta - V_\tau)d\tau + \xi\sqrt{V_\tau}dW_\tau^{(2)}$$

- This CIR-type variance process is challenging (can also handle other processes, e.g E-OU)
- Corresponding local volatility model is

$$dX_{LV,\tau} = \left(r_d - r_f - \frac{1}{2}\sigma_{LV}^2(X_{LV,\tau},\tau)\right)d\tau + \sigma_{LV}(X_{LV,\tau},\tau)dW_\tau$$

Need this when we consider accuracy of the method

Kolmogorov Forward Equation

The forward equation for the SLV model is

$$\frac{\partial}{\partial\tau}p = \frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma_{SLV}^2\psi^2(v)p\right) + \frac{\partial^2}{\partial x\partial v}\left(\rho\xi\sigma_{SLV}\psi(v)\sqrt{v}p\right) + \frac{\partial^2}{\partial v^2}\left(\frac{1}{2}\xi^2 vp\right) - \frac{\partial}{\partial x}\left((r_d - r_f - \frac{1}{2}\sigma_{SLV}^2\psi^2(v))p\right) - \frac{\partial}{\partial v}\left(\kappa(\eta - v)p\right)$$

for a (smooth) density $p(x, v, \tau; x_0, v_0)$. We know from Gyongy's theorem that

$$\begin{aligned}\sigma_{LV}^2(x,\tau) &= \mathbb{E}\left[\sigma_{SLV}^2(X_\tau,\tau)\psi^2(V_\tau) \mid X_\tau = x\right] \\ &= \sigma_{SLV}^2(x,\tau)\mathbb{E}\left[\psi^2(V_\tau) \mid X_\tau = x\right] \\ &= \sigma_{SLV}^2(x,\tau)\frac{\int_0^\infty \psi^2(v)p(x,v,\tau;x_0,v_0)\,dv}{\int_0^\infty p(x,v,\tau;x_0,v_0)\,dv}\end{aligned}$$

SLV Calibration

This suggests the following calibration procedure

1. Input stochastic parameters and local vol surface $\sigma^2_{LV}(x,\tau)$
2. At $t_i = i\Delta t$ guess an initial form for leverage surface $\sigma^2_{SLV}(x,t_i)$
   - 2.1 Solve the Kolmogorov forward equation for $p(x,v,t_i;x_0,v_0)$
   - 2.2 Update the leverage surface via Gyongy's theorem
   - 2.3 Iterate until convergence
3. Move to $t_{i+1} = t_i + \Delta t$ and repeat

Idea is simple and inner loop shouldn't need many iterations

Hard part, of course, is to solve the forward equation

Solving Forward Equation

- Have been many attempts to solve forward equation with finite differences
- Boundary condition known for $x = 0$
- No boundary condition known for $v = 0$
- Papers seldom give results for reference problems (e.g. Heston)
- Our understanding from practitioners is solving forward equation is still a problem under certain market conditions

Solving Forward Equation

Forward equation describes a probability density

- Has various properties, e.g. has to integrate to 1 at each time point
- Conversations with practitioners suggest they're not always concerned about this

Solving Forward Equation

Forward equation describes a probability density

- Has various properties, e.g. has to integrate to 1 at each time point
- Conversations with practitioners suggest they're not always concerned about this

Suppose we enforce unit integrability when computing solution

Solving Forward Equation

Forward equation describes a probability density

- Has various properties, e.g. has to integrate to 1 at each time point
- Conversations with practitioners suggest they're not always concerned about this

Suppose we enforce unit integrability when computing solution

- What effect might that have on the numerics?
- We know density must decay when x and v become large ...
- Might it be enough to "stabilise" the procedure?

Finite Volume Scheme

To test this we need to apply a finite volume discretisation

- Finite volume schemes used on equations in conservative form:

$$\frac{\partial}{\partial \tau} p + \frac{\partial}{\partial x}\left(a(p,x,\tau)p\right) = \frac{\partial}{\partial x}\left(b(p,x,\tau)\frac{\partial}{\partial x}p\right)$$

- Forward equation is not in conservative form:

$$\frac{\partial}{\partial \tau} p + \frac{\partial}{\partial x}\left(\mu_1 p\right) + \frac{\partial}{\partial y}\left(\mu_2 p\right) = \frac{\partial^2}{\partial x^2}\left(\frac{1}{2}\sigma_1^2 p\right) + \frac{\partial^2}{\partial x \partial y}\left(\rho\sigma_1\sigma_2 p\right) + \frac{\partial^2}{\partial y^2}\left(\frac{1}{2}\sigma_2^2 p\right)$$

- Putting in conservative form means differentiating σ_1 and σ_2, in particular would need to differentiate the leverage surface $\sigma^2_{SLV}(x,\tau)$
- Want to avoid this since surface is not known analytically

Finite Volume Scheme

Turns out can still make 2nd order f.v. scheme for forward equation

- No transformations, no differentiation, no mesh assumptions
- Discretise the equation directly with "classical" f.v. approach
  - Treat $\frac{\partial^2}{\partial x^2}\left(\sigma^2 p\right)$ as $\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x}\sigma^2 p\right)$
  - Find simple second order approximation to $\frac{\partial}{\partial x}\sigma^2 p$ on left and right cell boundaries
- Impose zero-flux conditions at all domain boundaries
- Dirac delta initial condition: use 4 Rannacher half-steps
- Use Hunsdorfer-Verwer ADI time stepping thereafter
- Solve Rannacher equations using suitable multigrid
- Non-uniform grid (static) to refine around points of interest

One Dimensional Test Cases

1D Test Case: Black-Scholes

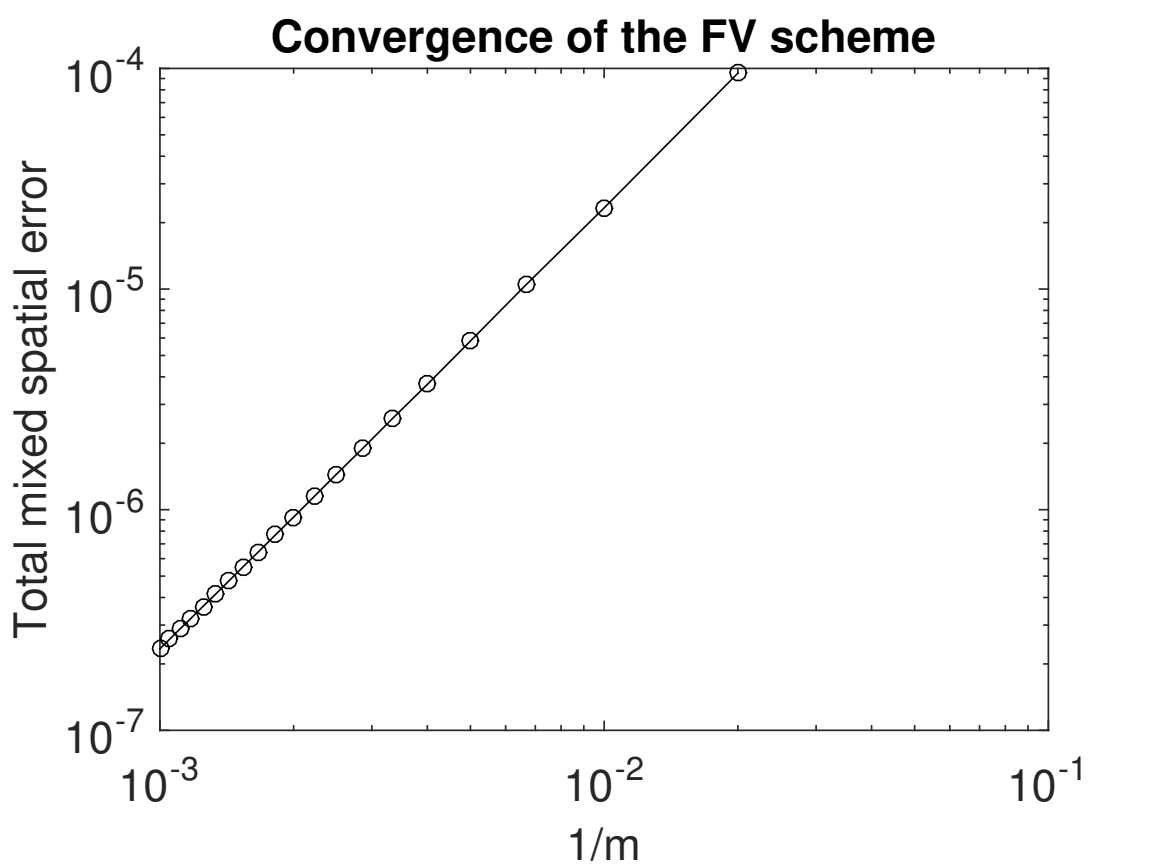


Figure: Second order convergence of numerical solution to theoretical solution

1D Test Case: Black-Scholes

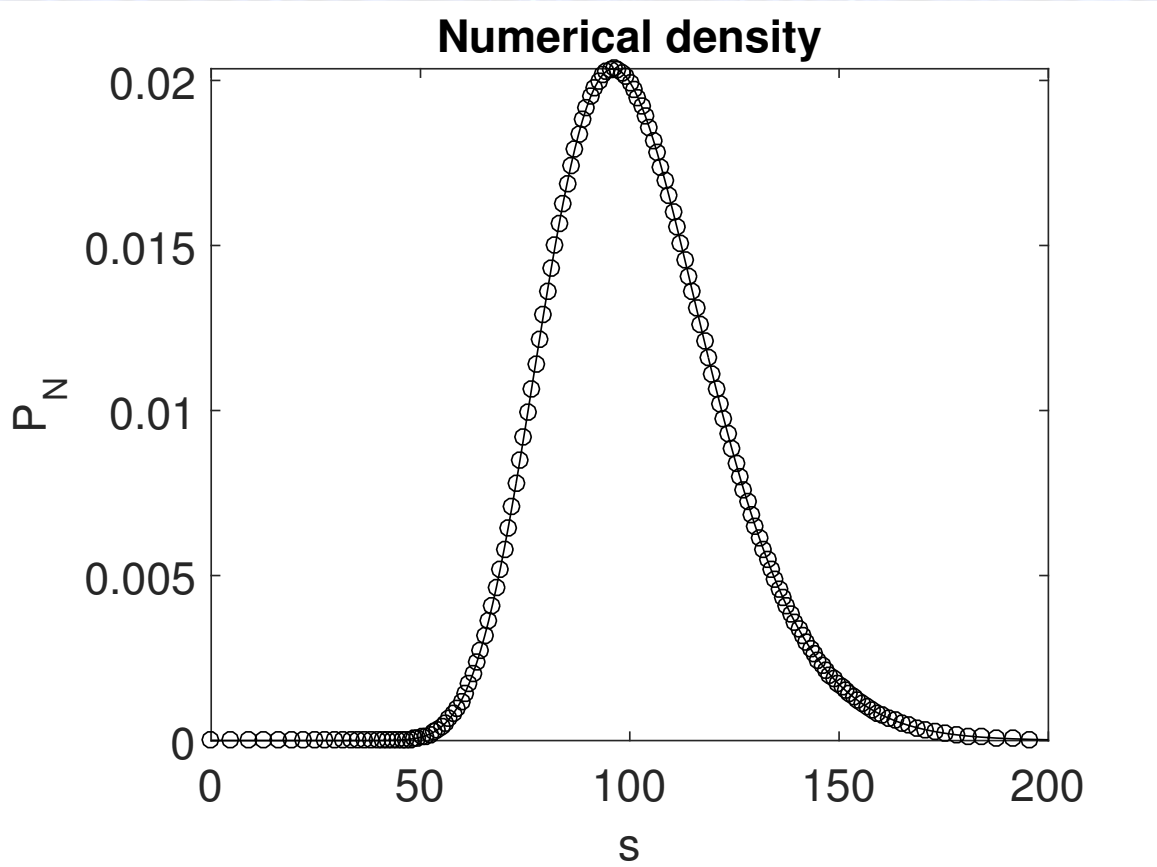


Figure: Numerical density at $T = 1$

1D Test Case: CIR (Feller Satisfied)

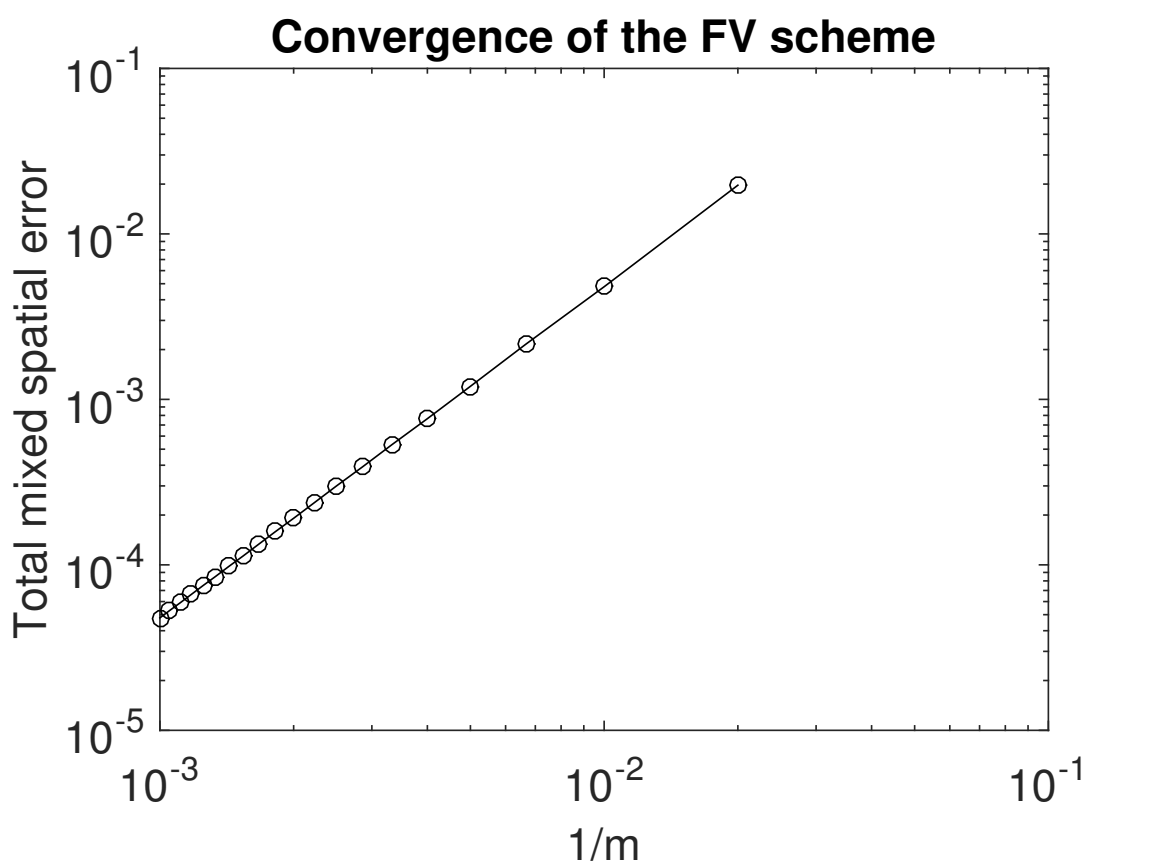


Figure: Second order convergence of numerical solution to theoretical solution

1D Test Case: CIR (Feller Satisfied)

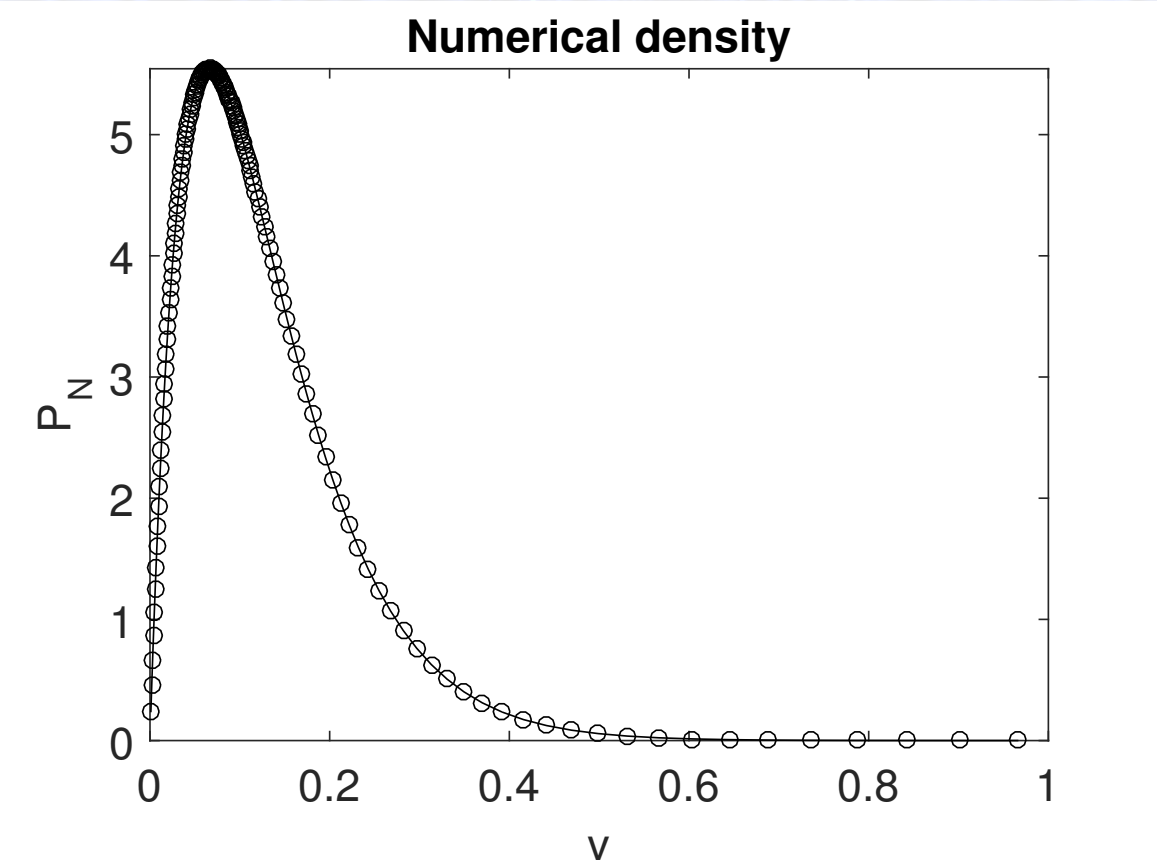


Figure: Numerical density at $T = 0.25$

1D Test Case: CIR (Feller Violated)

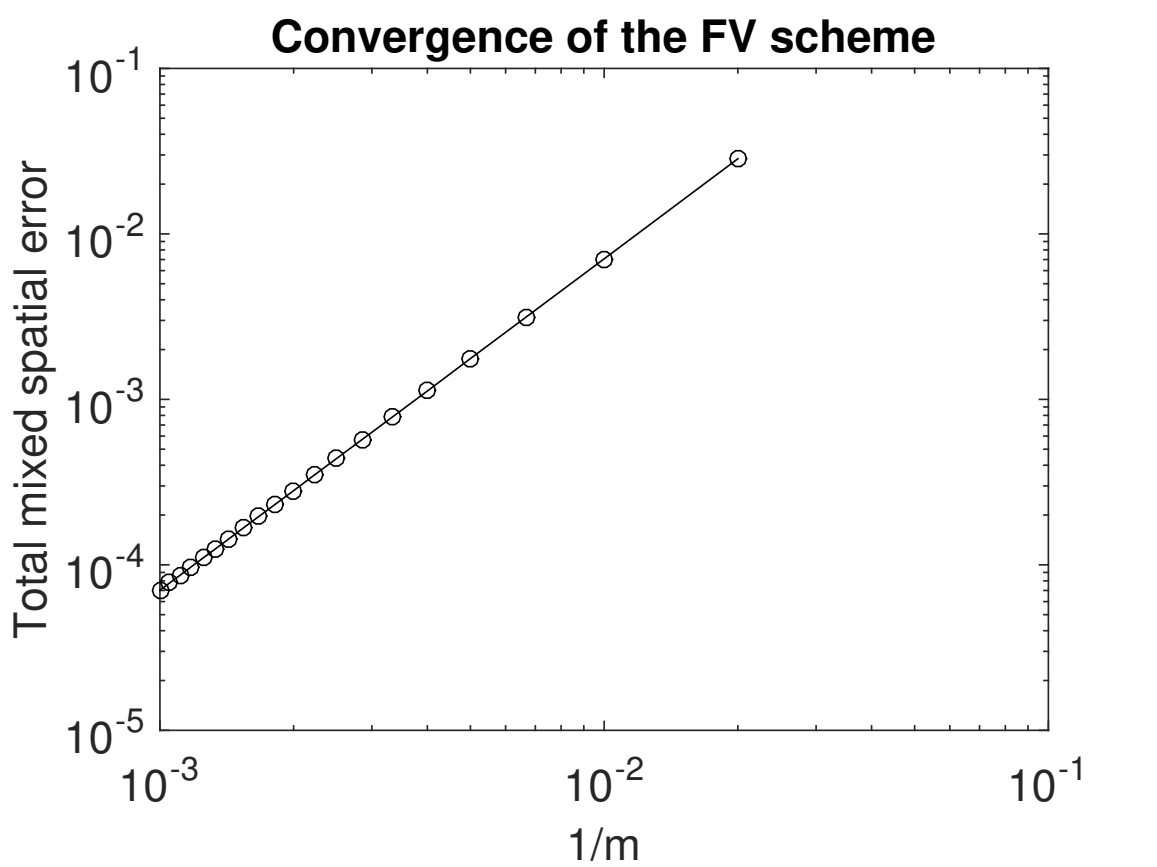


Figure: Between first and second order convergence of numerical solution to theoretical solution

1D Test Case: CIR (Feller Violated)

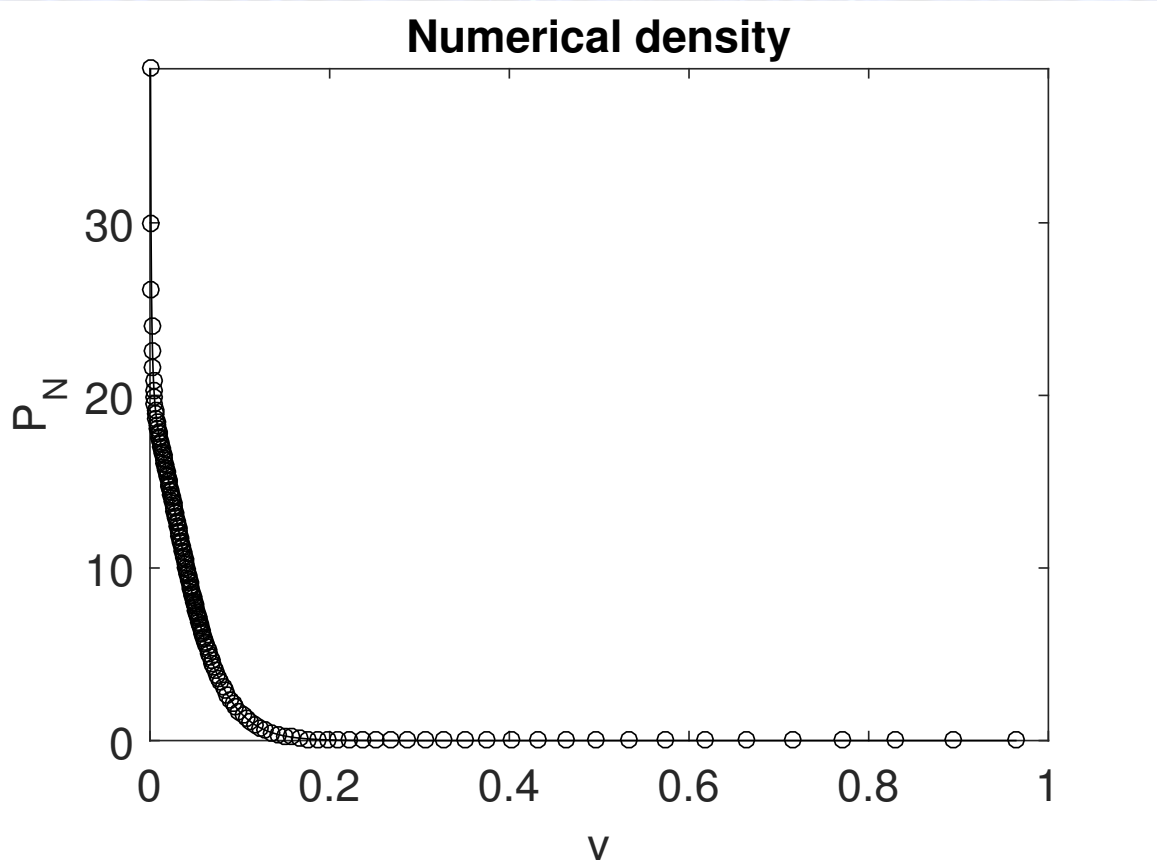


Figure: Numerical density at $T = 0.25$

Two Dimensional Test Cases

2D Test Case: Black-Scholes

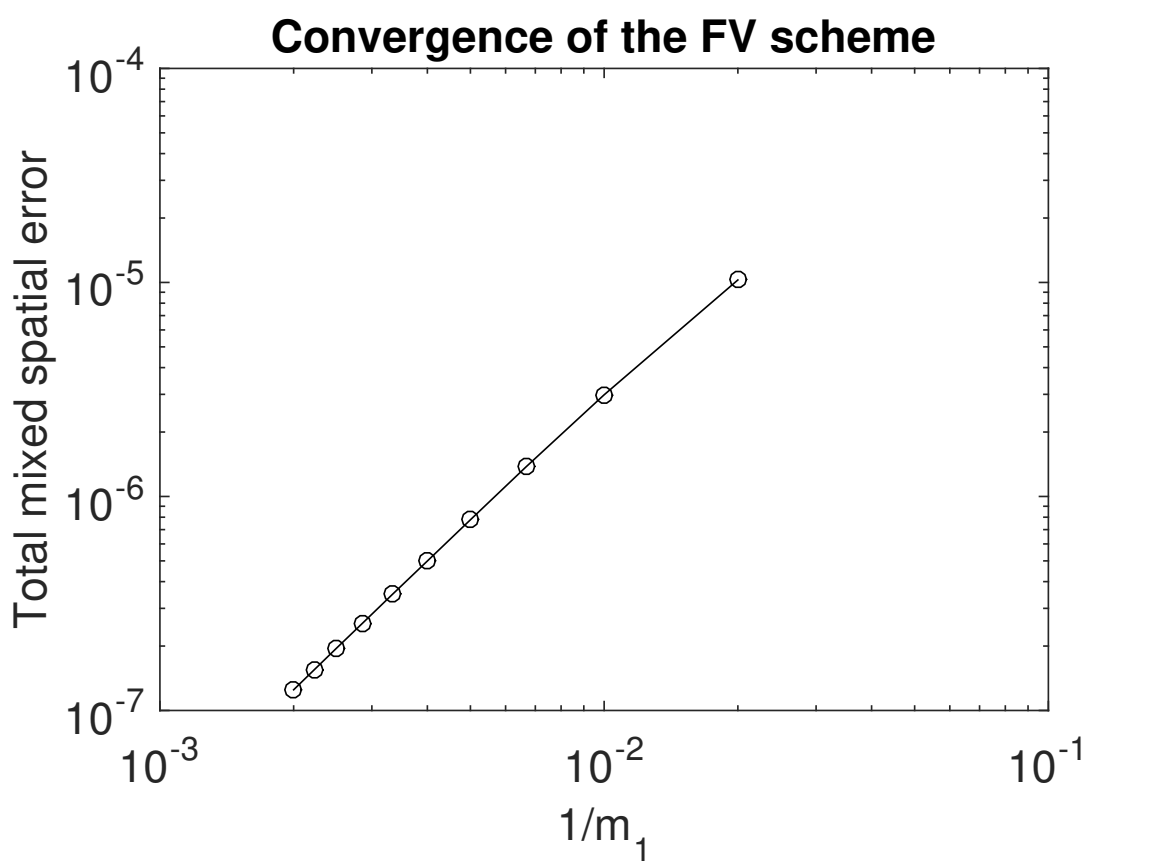


Figure: Second order convergence of numerical solution to theoretical solution

2D Test Case: Black-Scholes

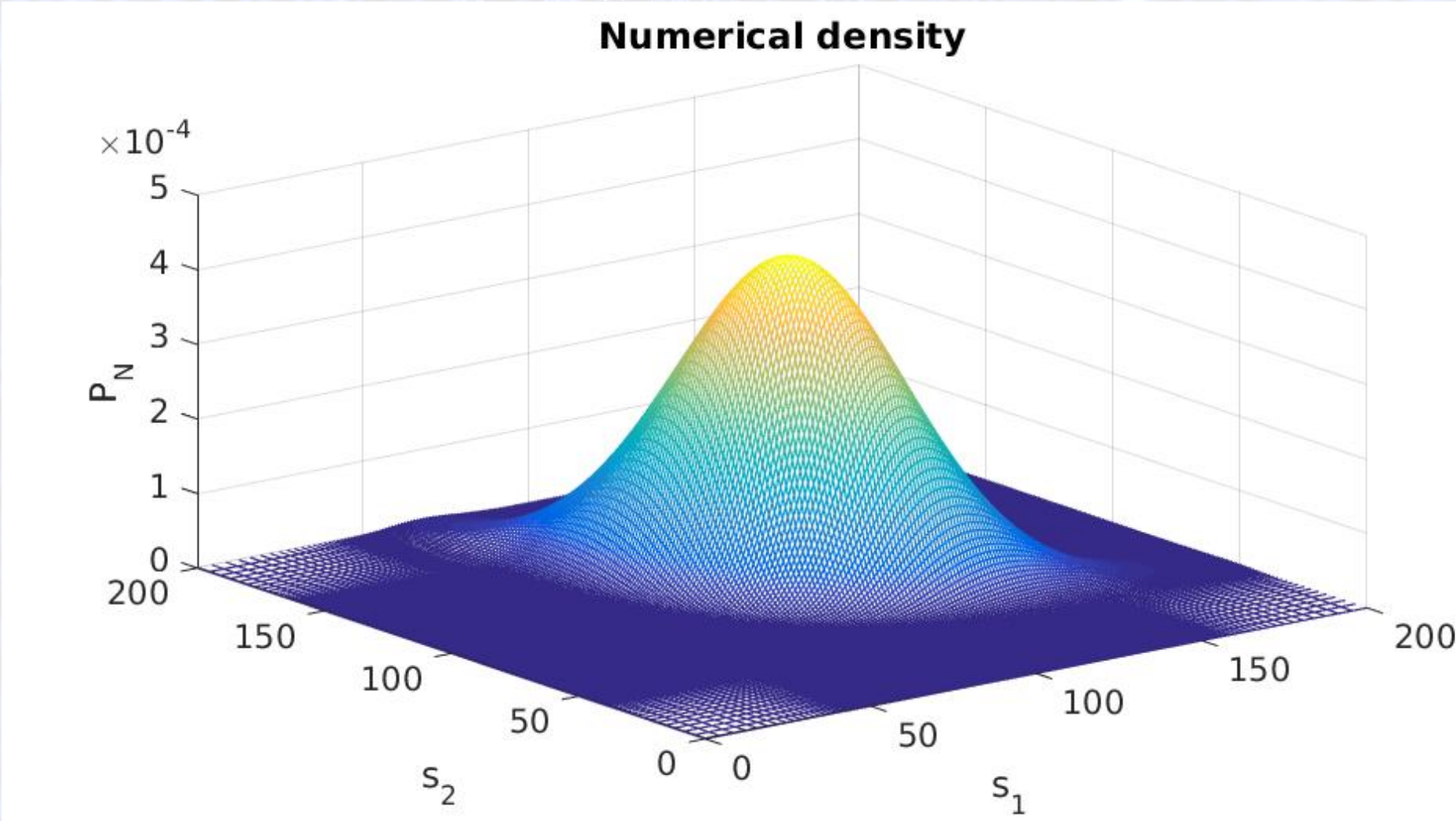


Figure: Numerical density at

1D Test Case: Heston (Feller Satisfied)

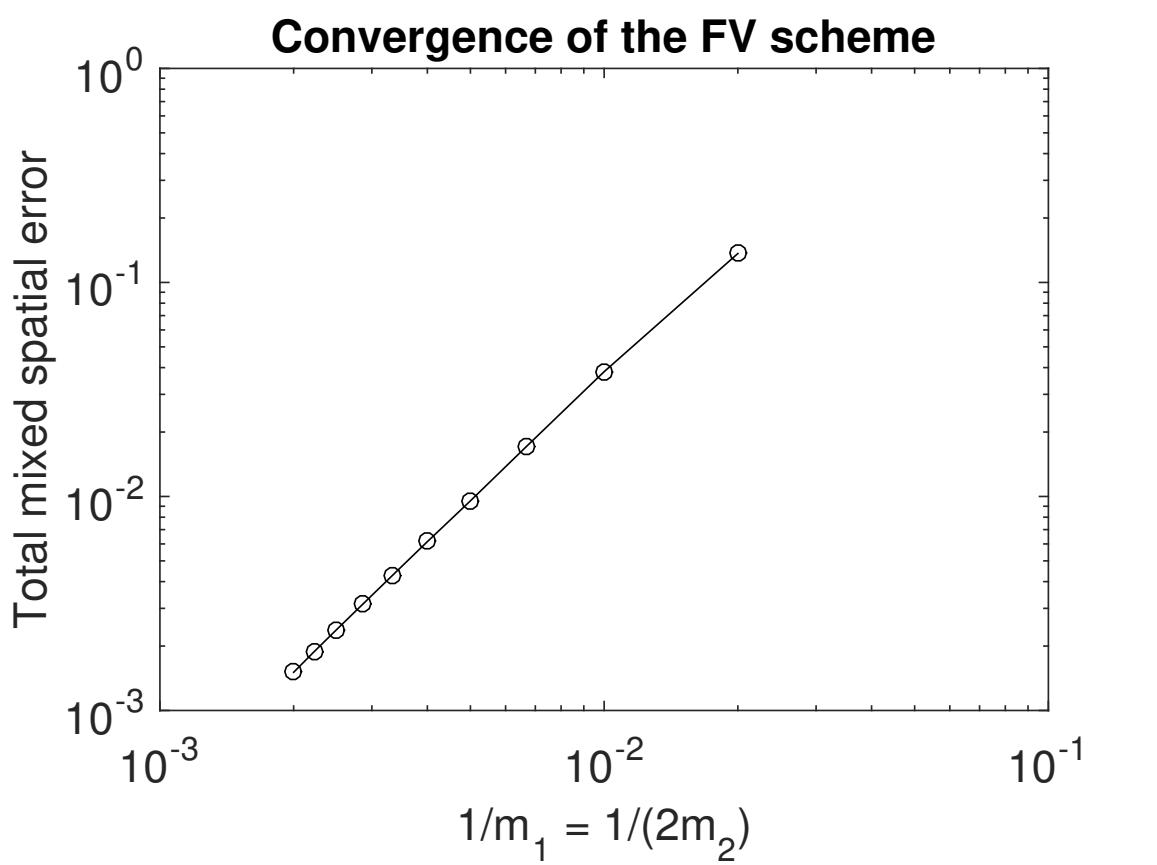


Figure: Second order convergence of numerical solution to theoretical solution

1D Test Case: Heston (Feller Satisfied)

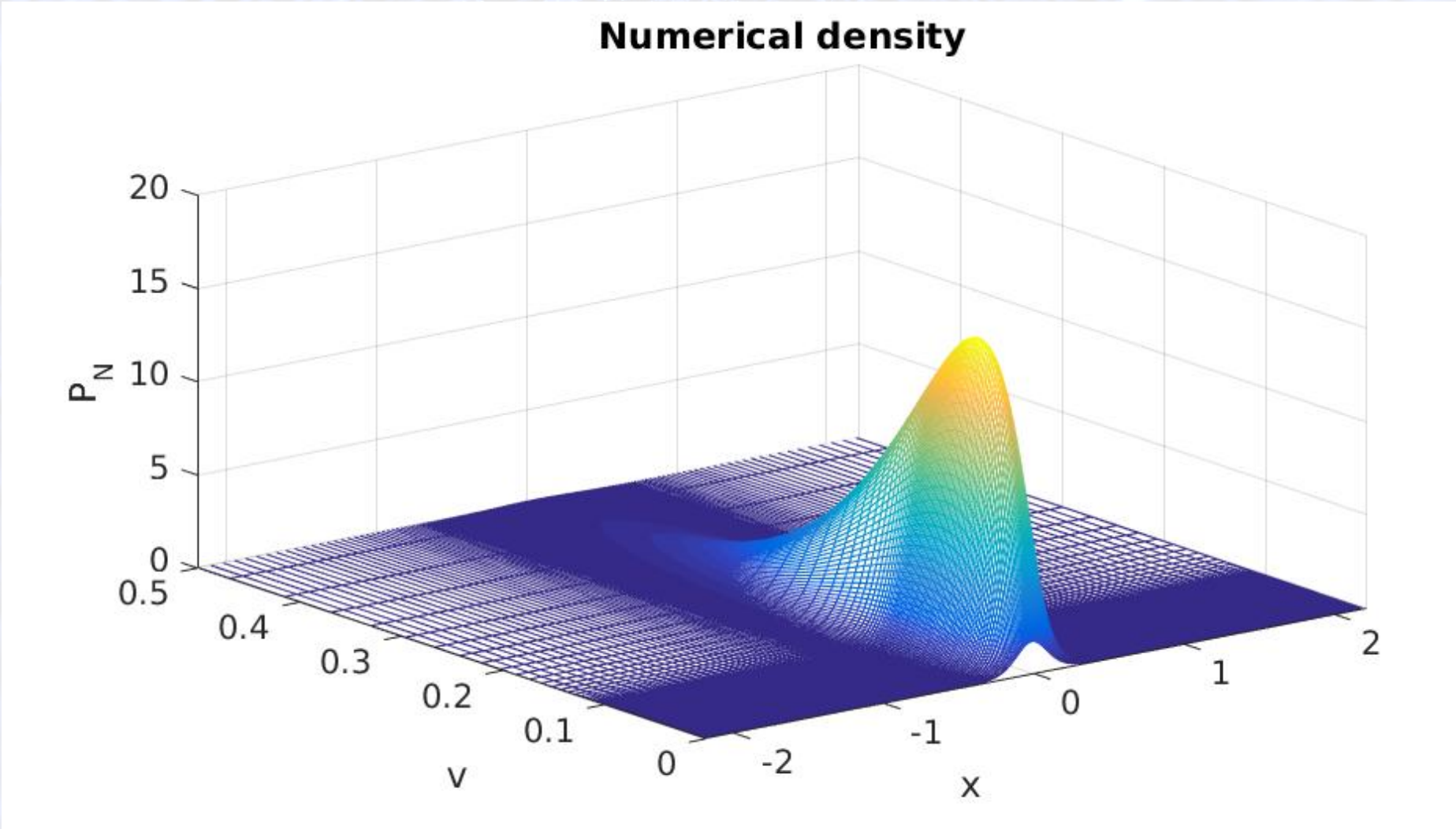


Figure: Numerical density at $T = 0.25$

1D Test Case: Heston (Feller Violated)

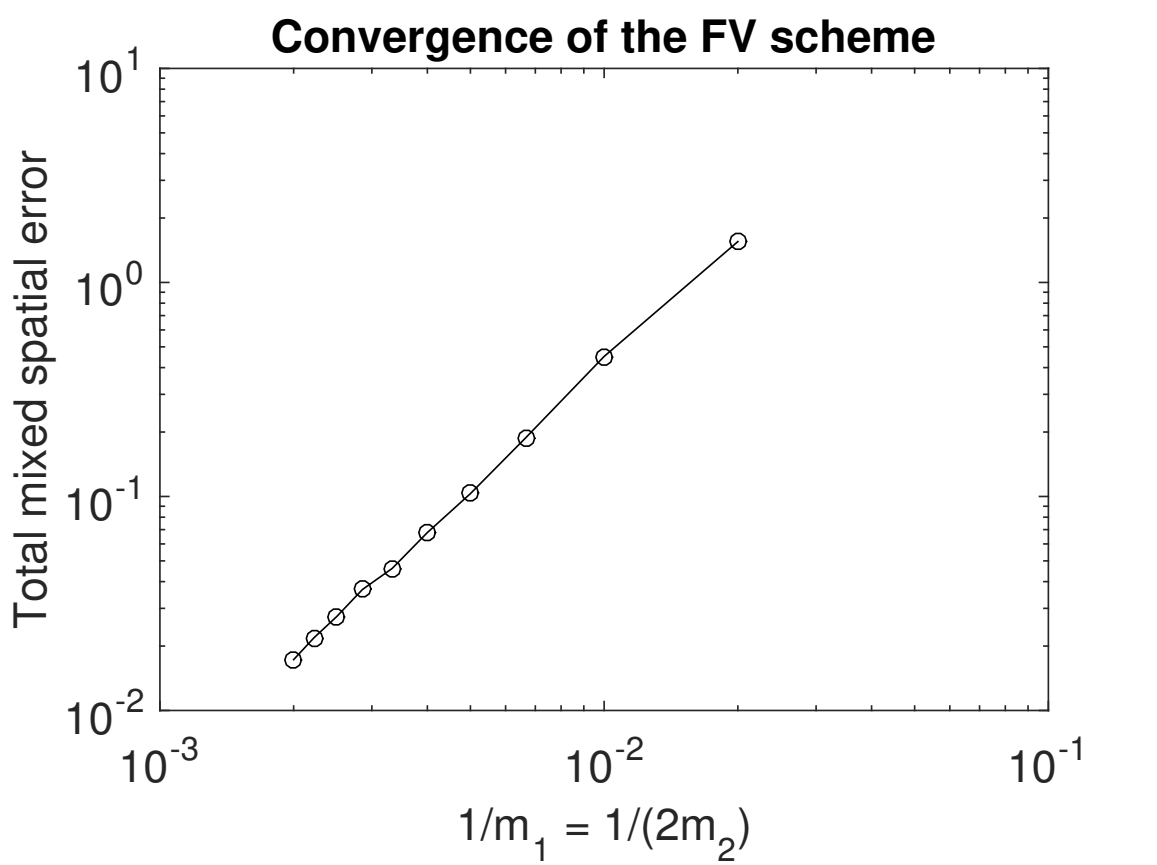


Figure: Between first and second order convergence of numerical solution to theoretical solution

1D Test Case: Heston (Feller Violated)

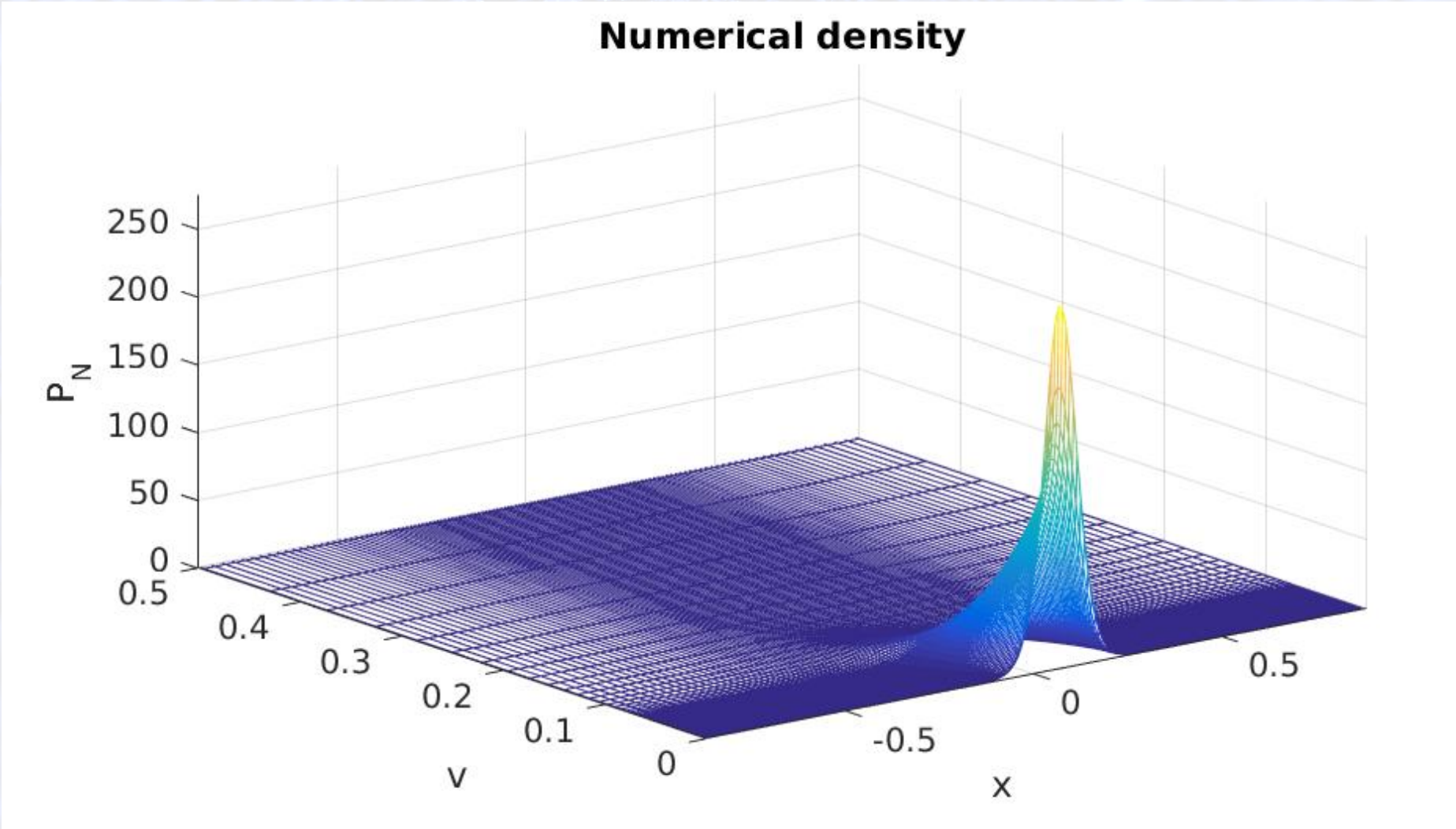


Figure: Numerical density at $T = 0.25$

SLV Calibration

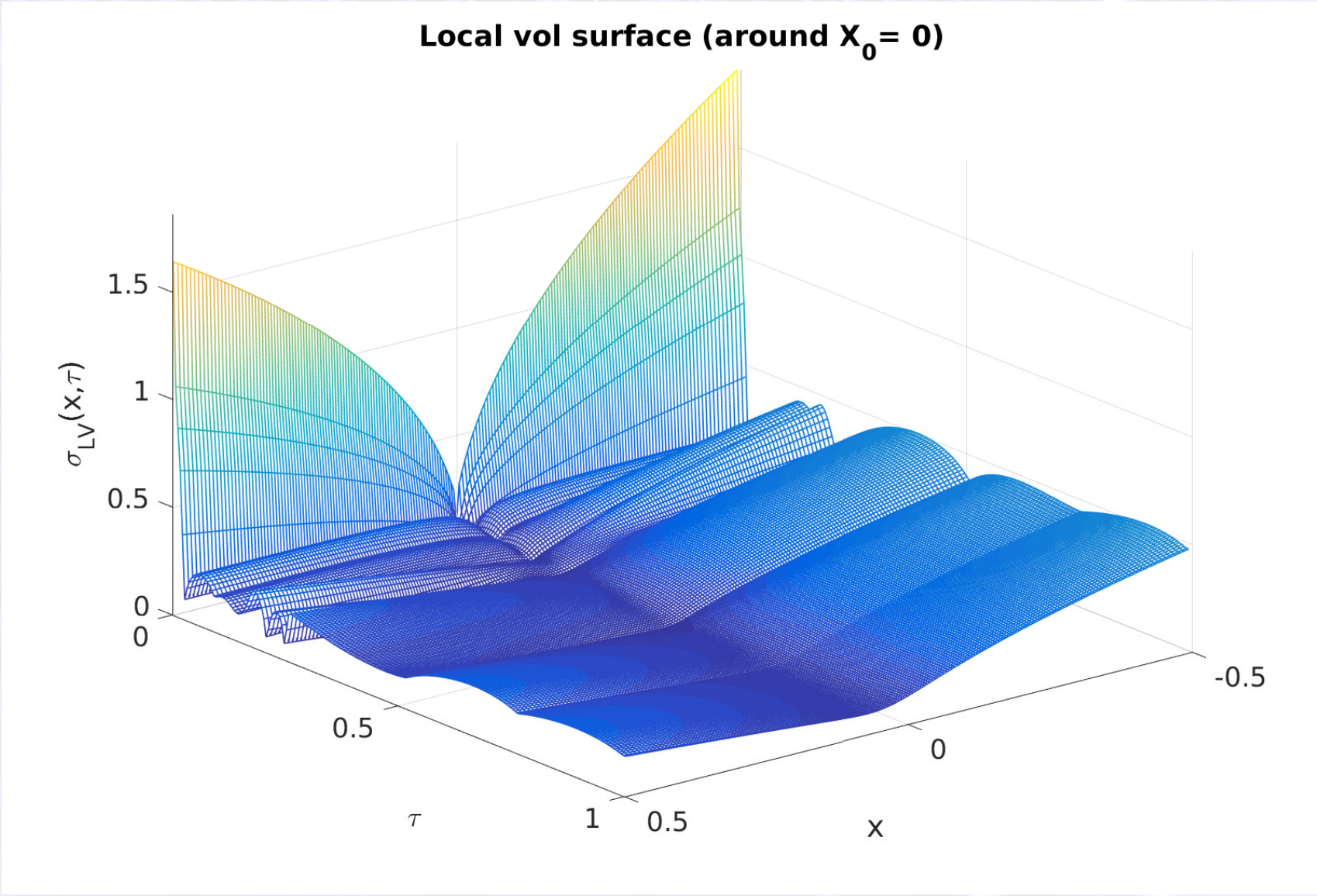
Calibration Data

- EUR/USD vanilla option data from 2 March 2016 ($S_0 = 1.088$)
- Local volatility surface computed from data via SSVI-type interpolation
- Three sets of stochastic parameters taken from book by Iain Clark

How to judge accuracy?

- Compare LV density and SLV marginal density, all computed via our finite volume scheme
- Compare implied vol under LV and SLV models

Input Local Volatility Surface



SLV Calibration: Feller Satisfied

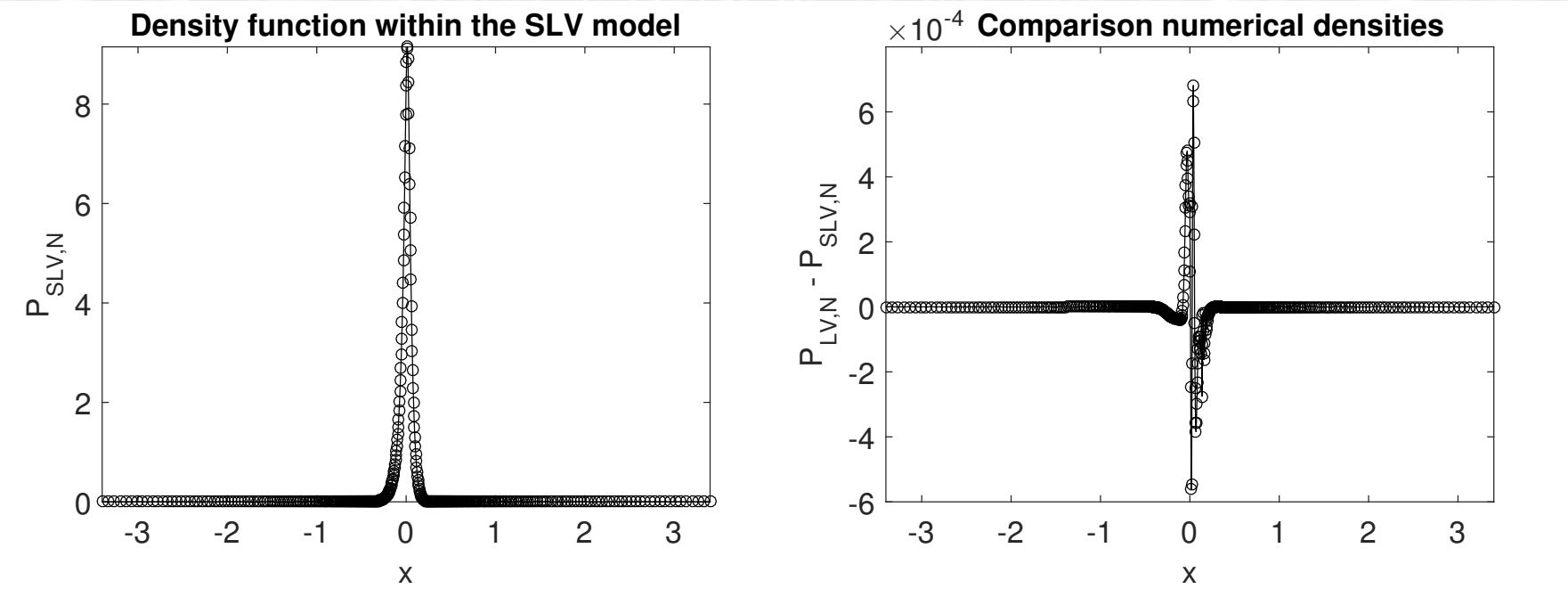


Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 0.25$

SLV Calibration: Feller Violated

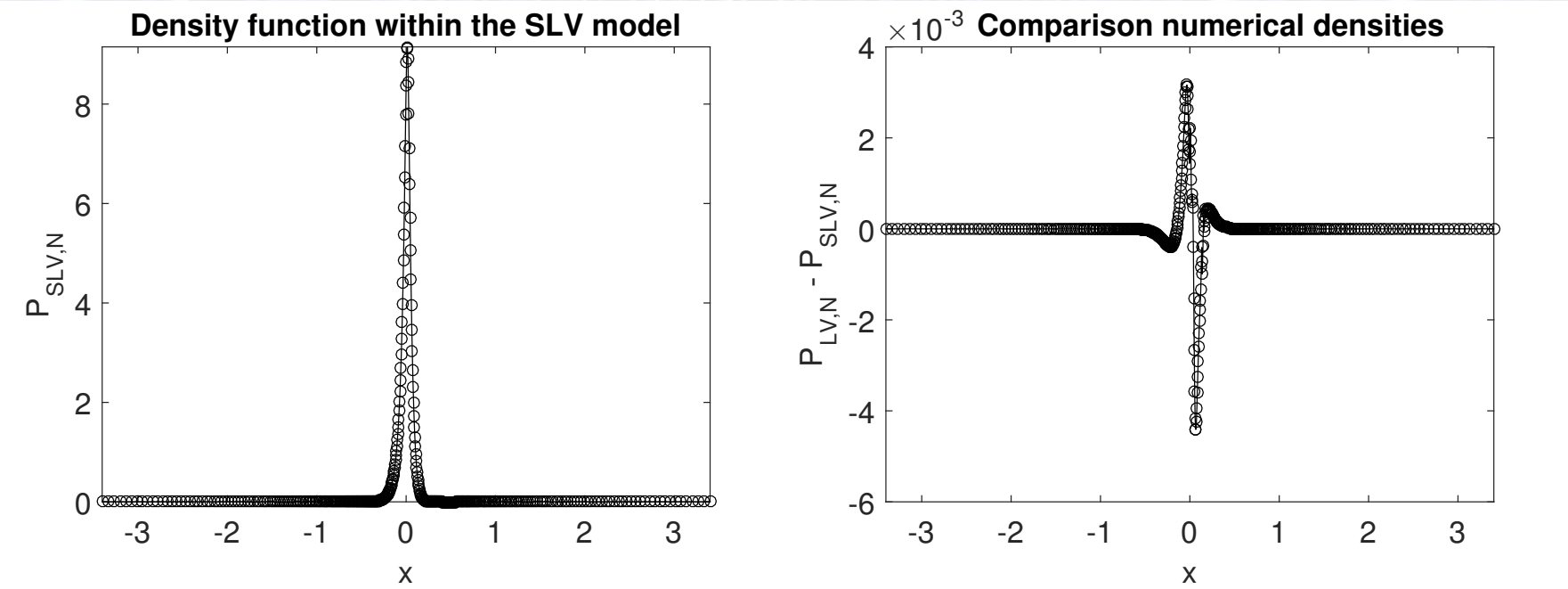


Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 0.25$

SLV Calibration: Feller Strongly Violated

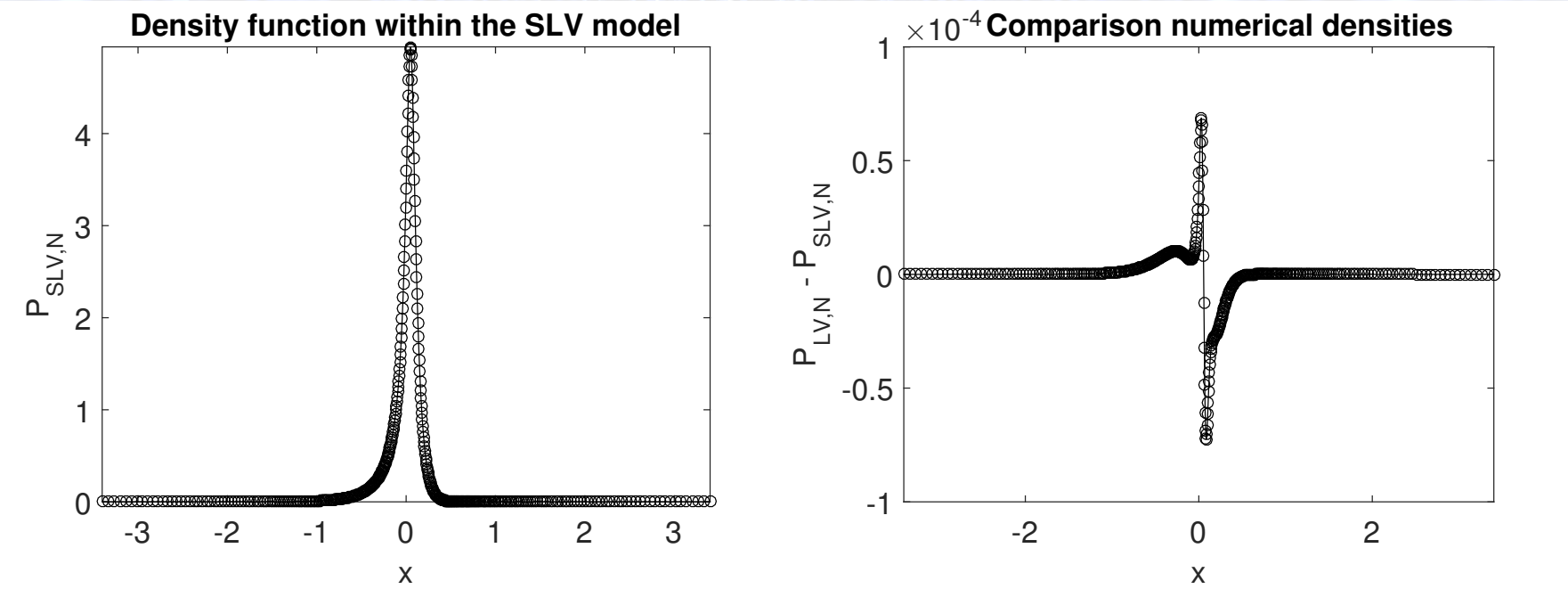


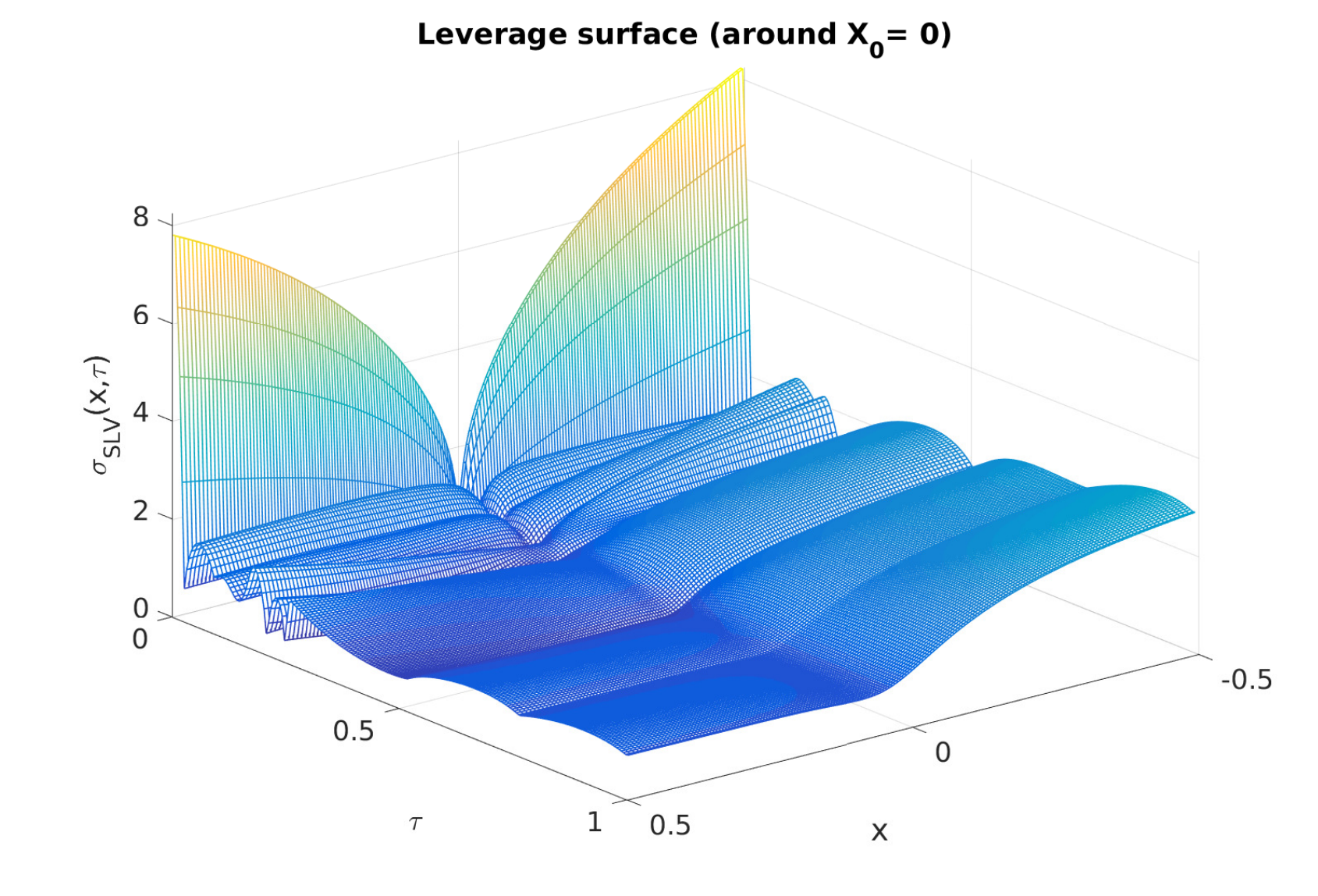
Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 1$

SLV Calibration: Differences in Implied Vol

	$T = 0.25$	Set E	Set F	$T = 1$	Set G
K/S_0	$\sigma_{imp,LV}$	ϵ_{imp}	ϵ_{imp}	$\sigma_{imp,LV}$	ϵ_{imp}
0.75	19.18	0.1005	0.1208	21.94	0.0021
0.80	18.40	0.0212	0.0454	20.20	0.0015
0.90	15.01	0.0033	0.0154	16.65	0.0008
1.0	11.26	0.0011	0.0030	13.14	0.0004
1.10	11.59	0.0011	0.0153	11.38	0.0003
1.20	13.20	0.0009	0.0937	11.77	0.0003
1.25	14.03	0.0006	0.1888	12.12	0.0003

Table: Set E = Feller satisfied, Set F = Feller violated and Set G = Feller strongly violated

Calibrated Leverage Surface: Feller Strongly Violated



Stressing the Method

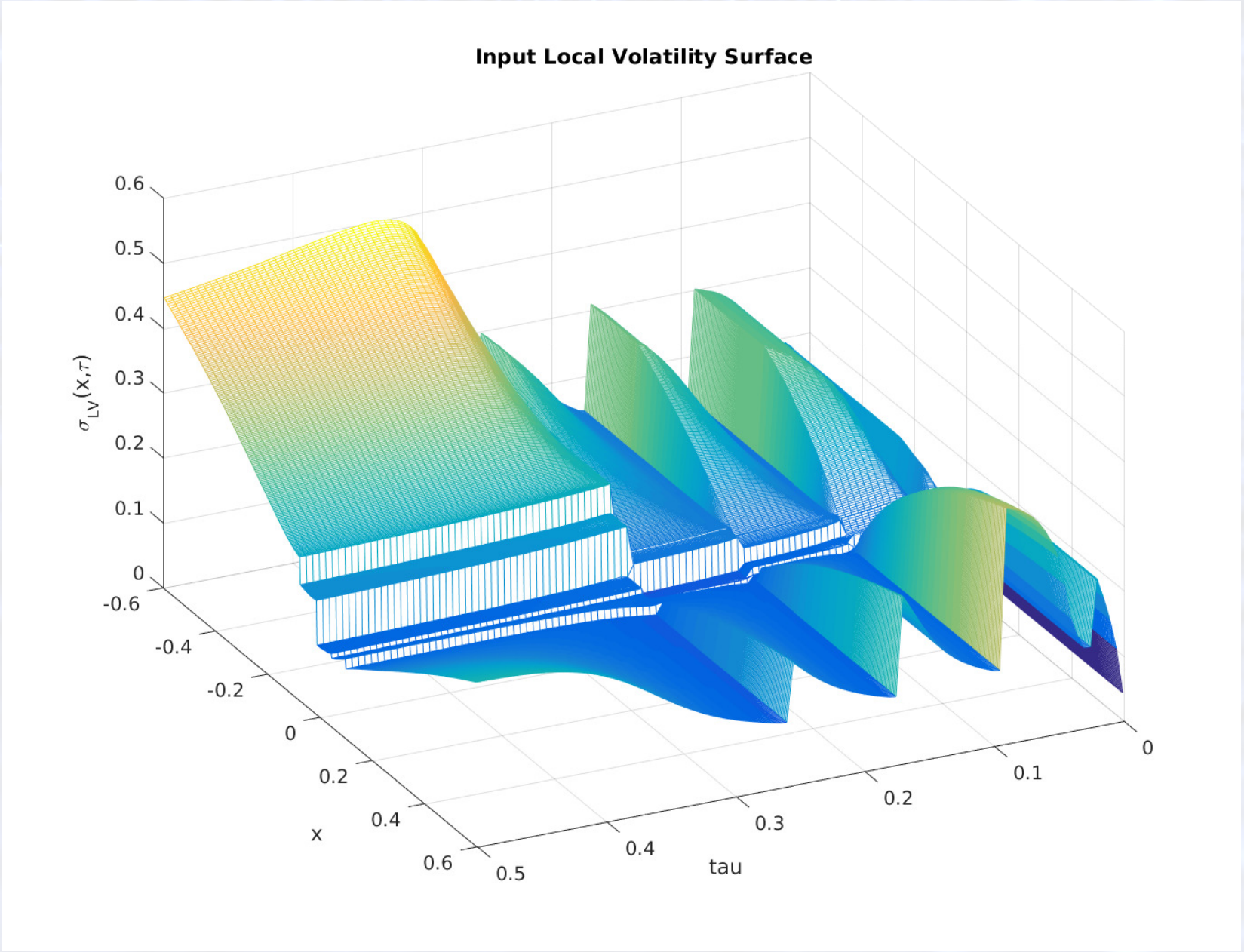
More Extreme Parameter Sets

More parameter sets from Iain Clark's book on FX modeling

	Set H	Set I	Set J
κ	3.02	3.02	0.3
η	0.04	0.04	0.18
σ	0.61	0.61	2.44
ρ	0.63	0.63	-0.58
T	0.5	0.5	5
σ_{LV}	A-H	G-J	G-J
Feller	0.65	0.65	0.018

Sets H,I are US/TKY pairs, while Set J is AUD/JPY. Note strong violation of Feller condition. Local vol via Andreasen-Huge gives a very uneven input local vol surface

Set H: Input Local Volatility Surface



Set H: SLV Density vs LV Density

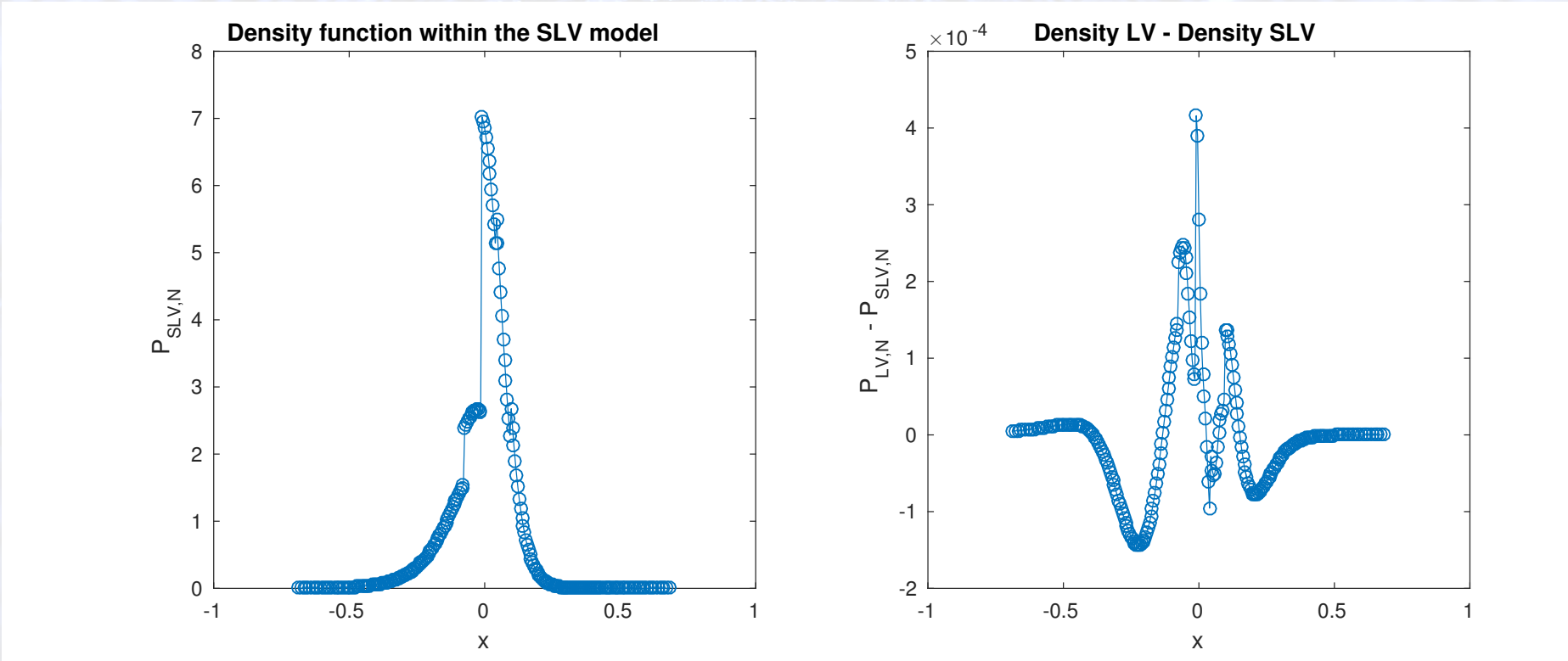
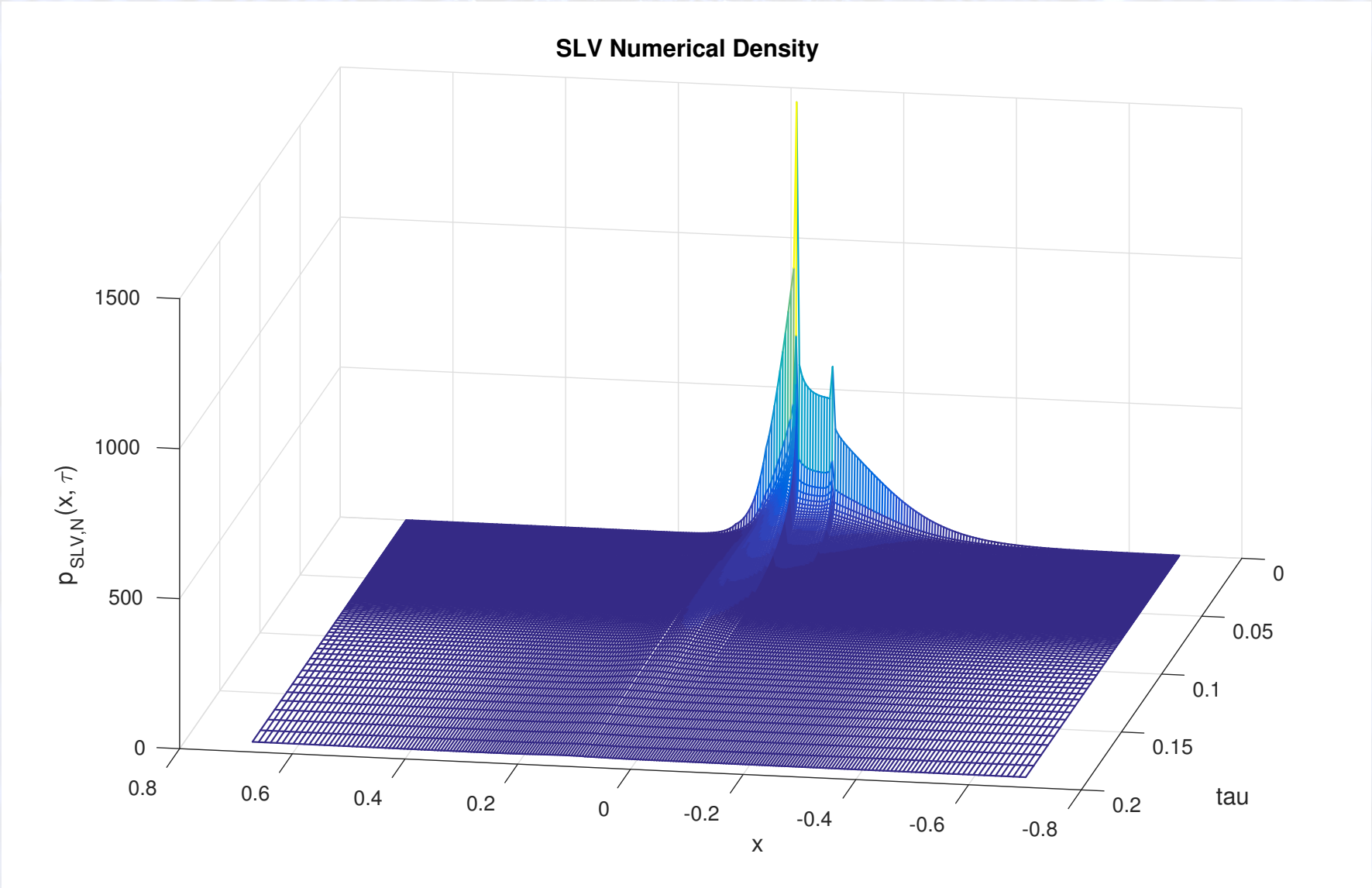
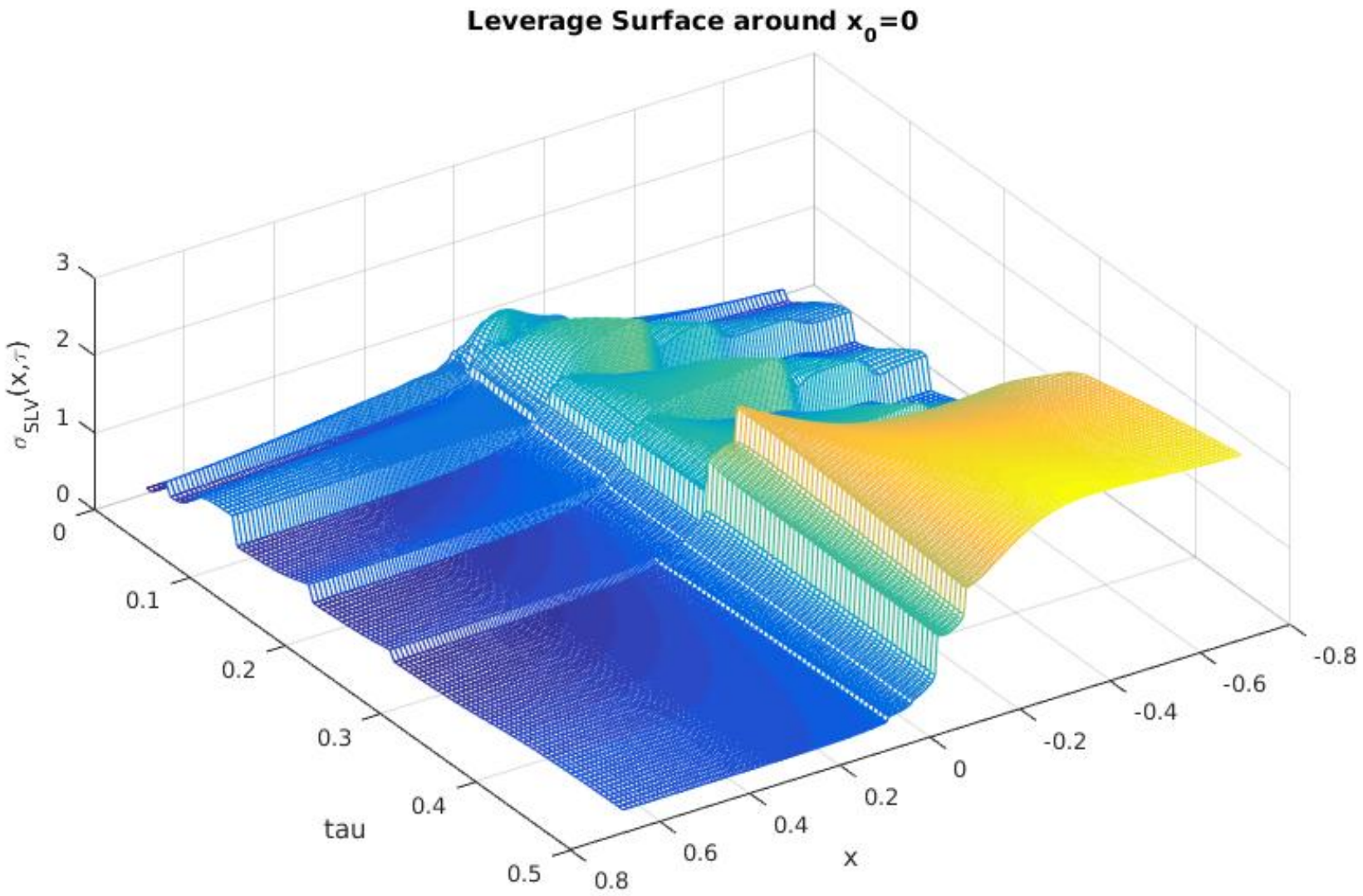


Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 0.5$

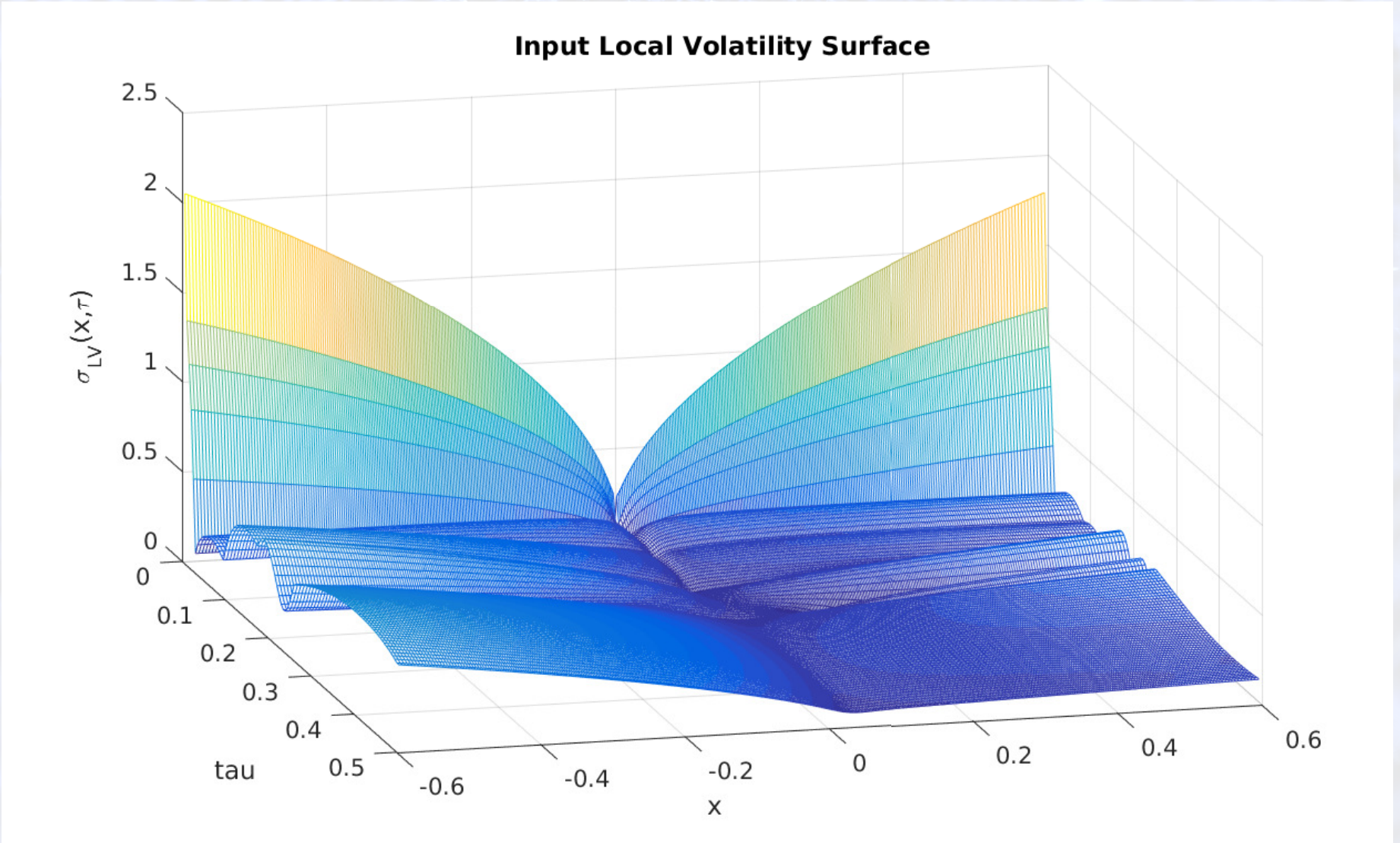
Set H: SLV 3D Density



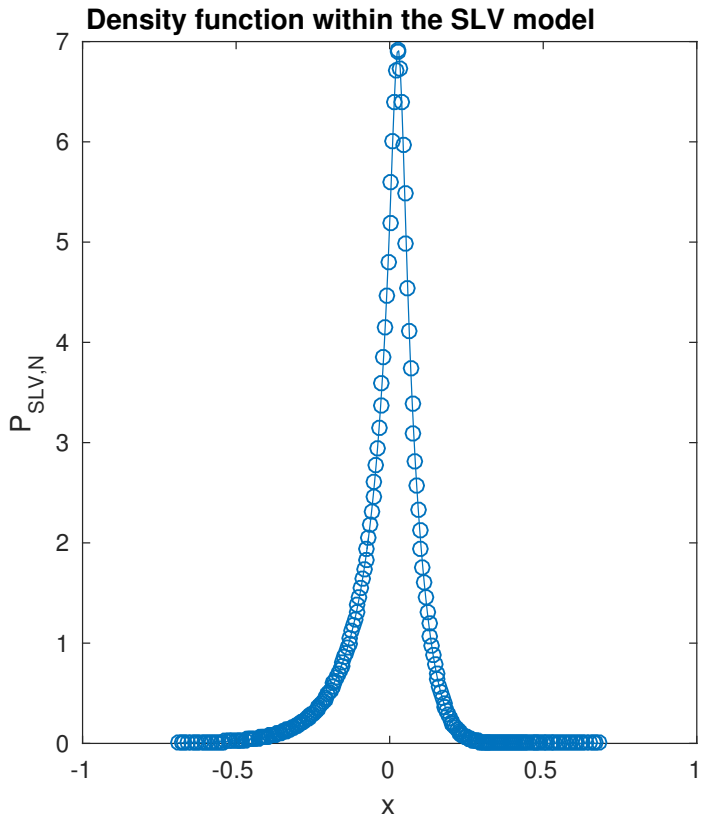
Set H: Calibrated Leverage Surface



Set I: Input Local Volatility Surface



Set I: SLV Density vs LV Density



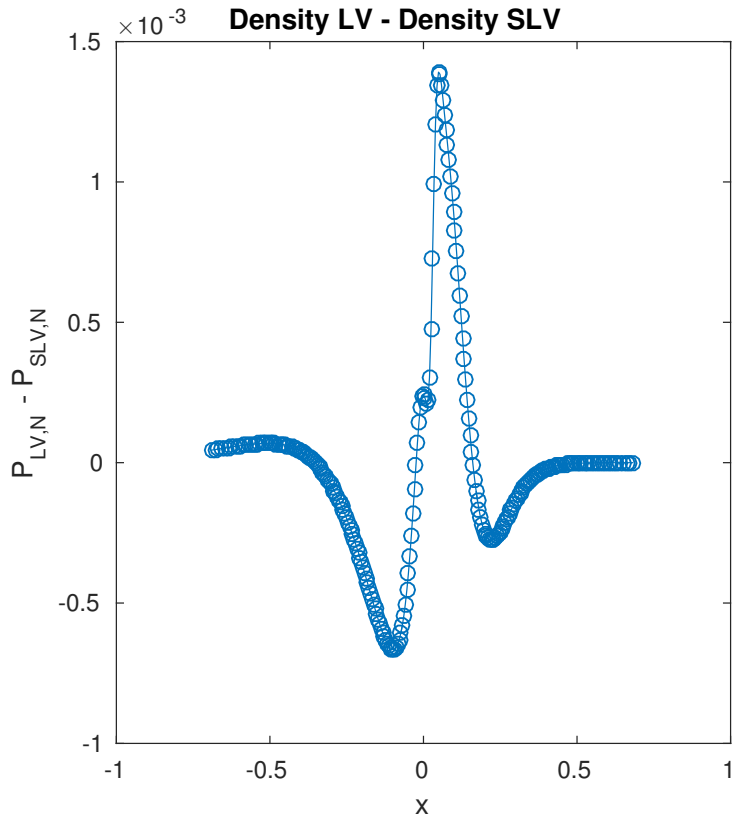
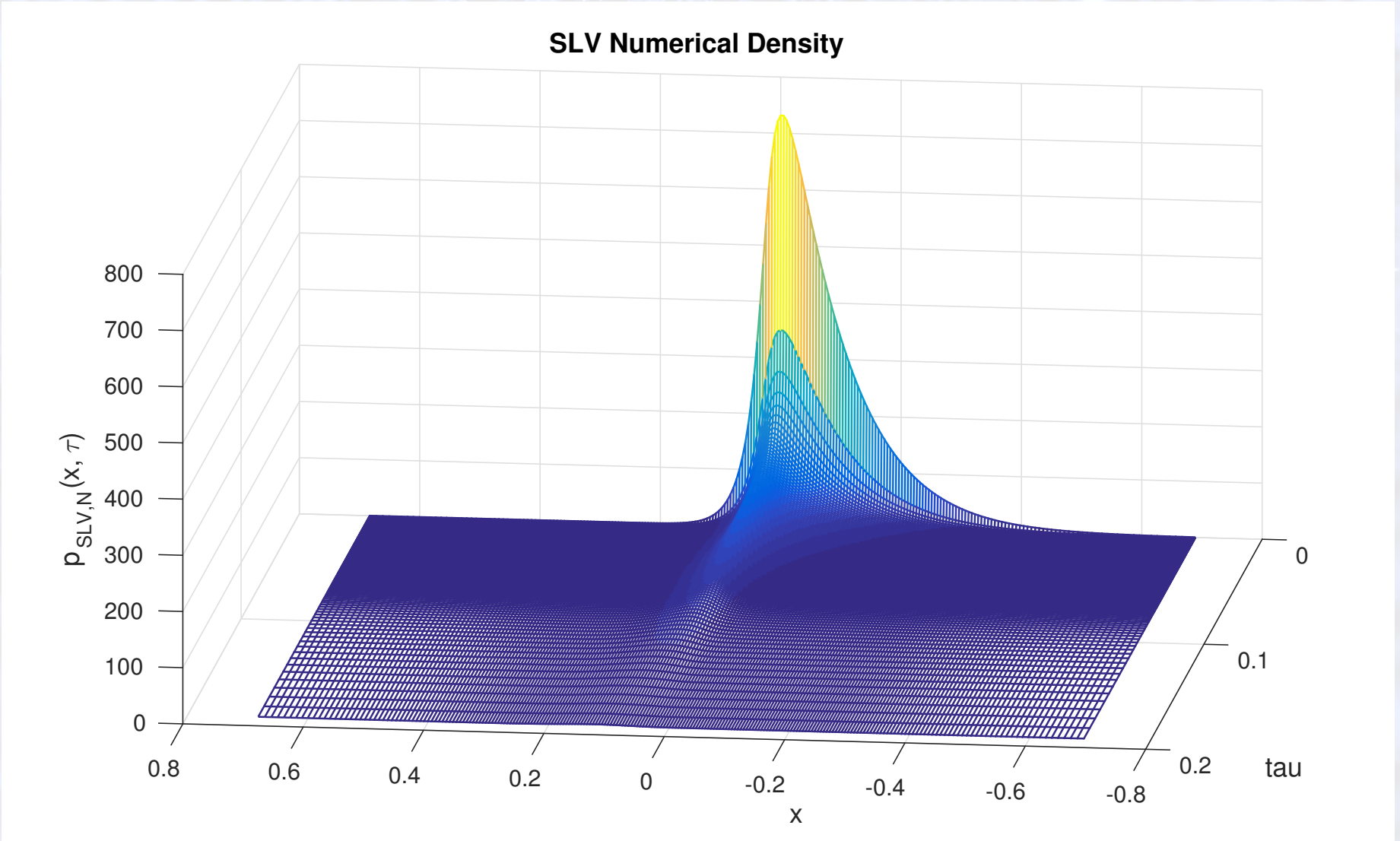
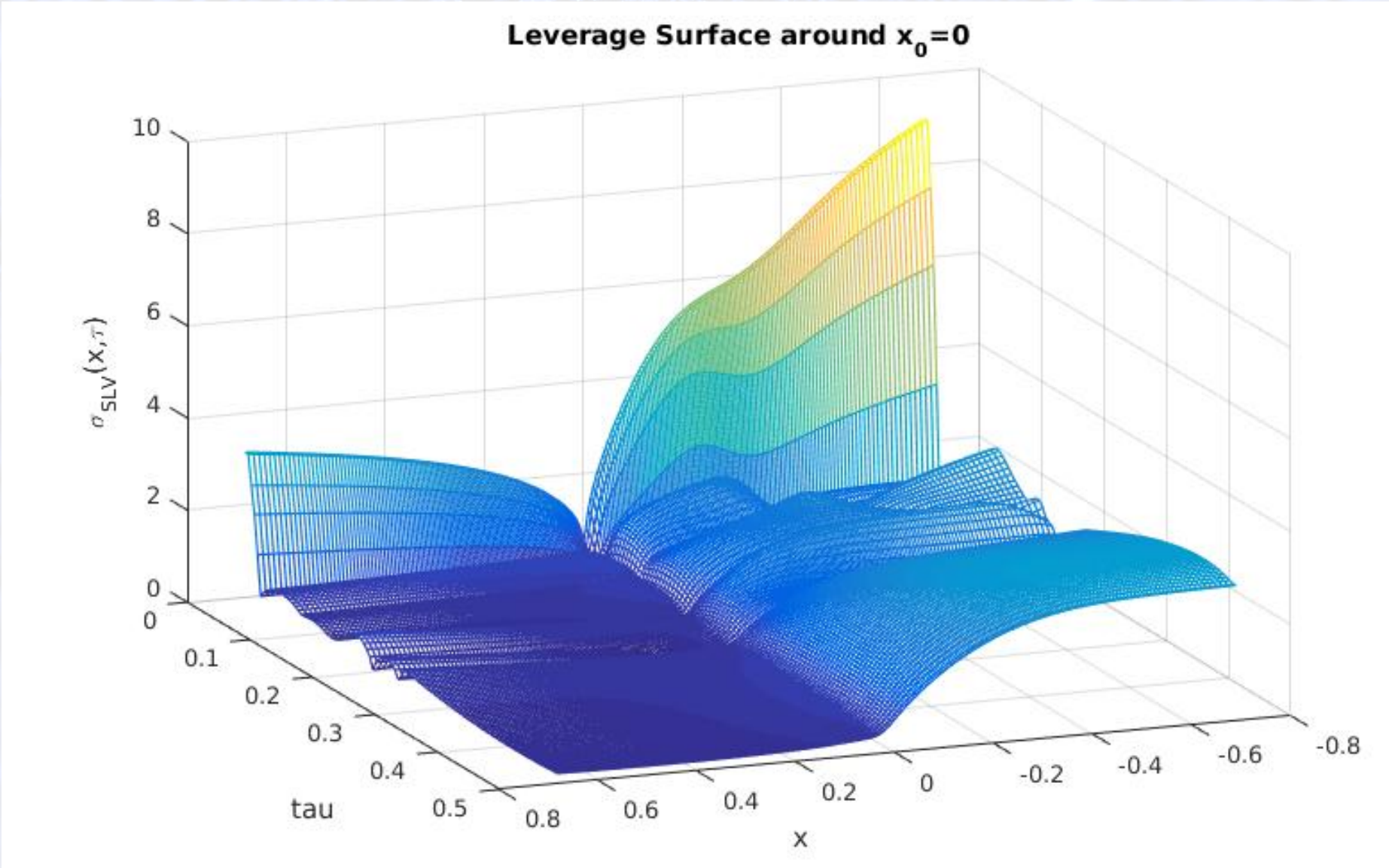


Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 0.5$

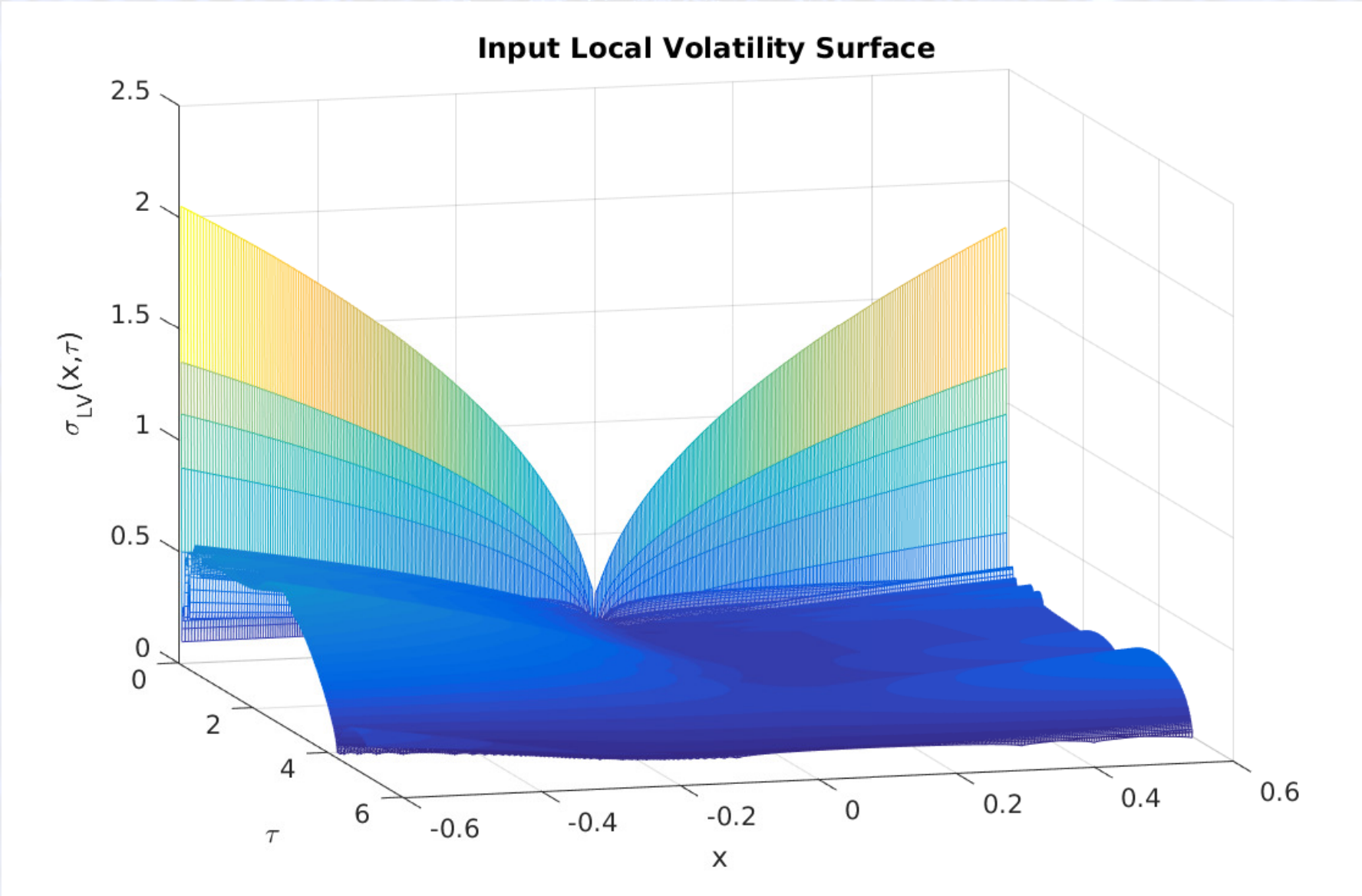
Set I: SLV 3D Density



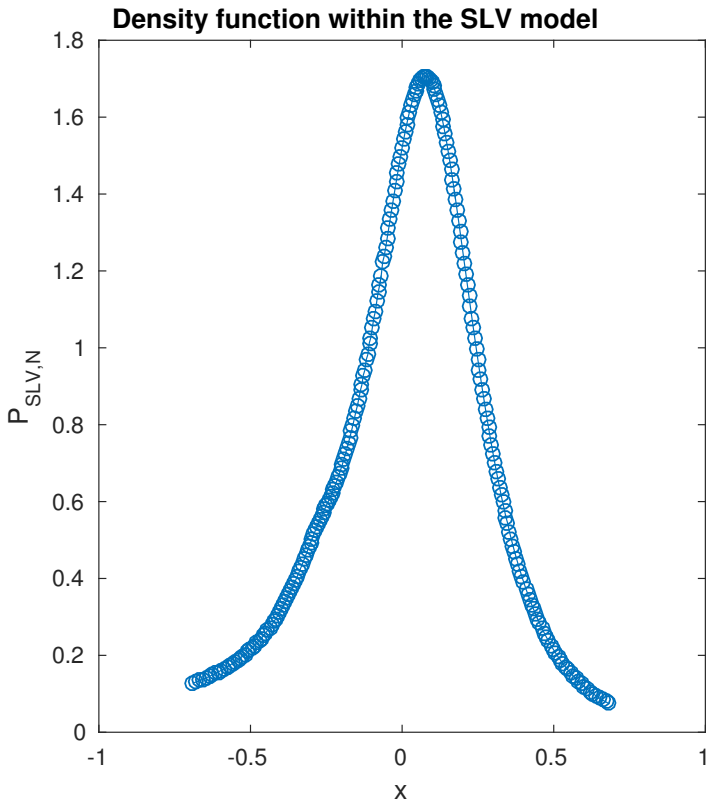
Set I: Calibrated Leverage Surface



Set J: Input Local Volatility Surface



Set J: SLV Density vs LV Density



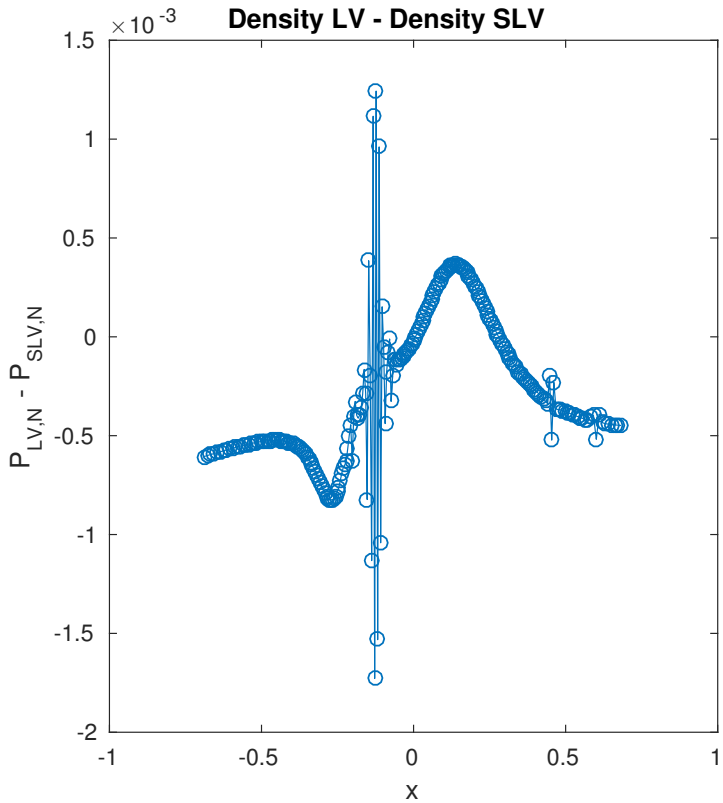
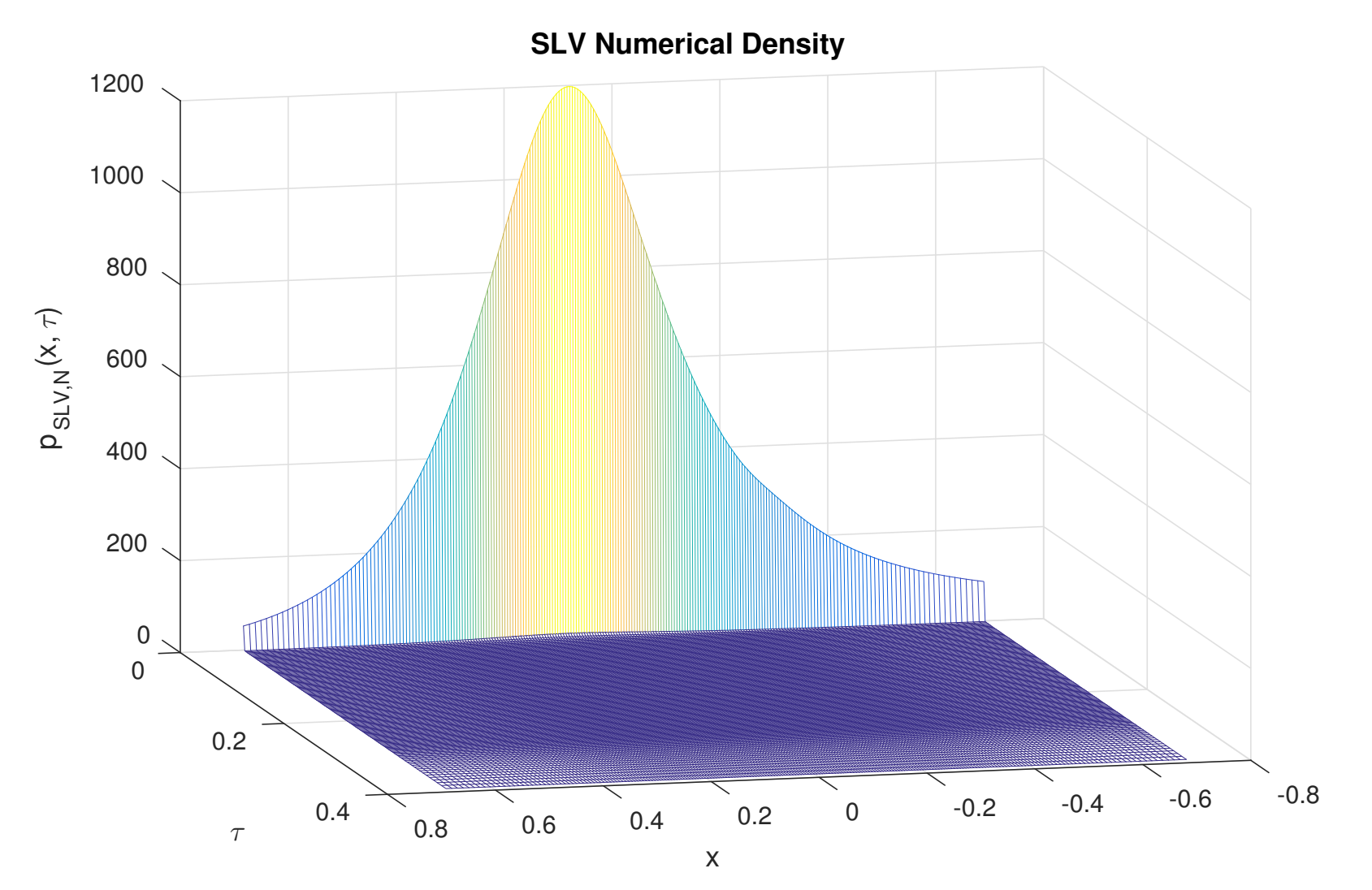
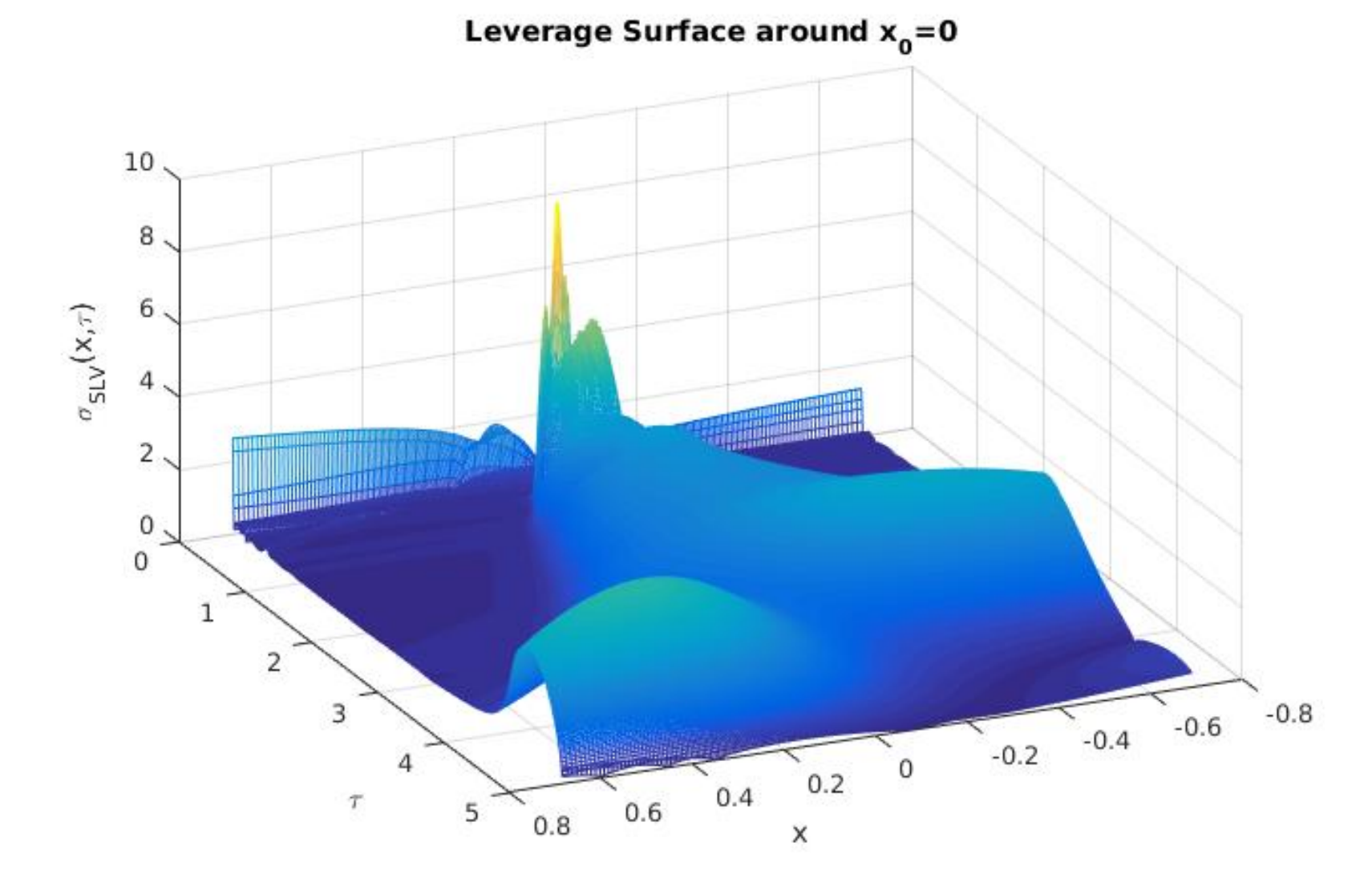


Figure: Marginal SLV density, and difference between LV and marginal SLV densities at $T = 5$

Set J: SLV 3D Density



Set J: Calibrated Leverage Surface



Implied Volatilities Compared

K/S_0	Set H		Set I		Set J	
	$\sigma_{imp,LV}$	ε_{imp}	$\sigma_{imp,LV}$	ε_{imp}	$\sigma_{imp,LV}$	ε_{imp}
0.75	20.48	7.3e-4	21.63	1.3e-2	19.68	1.8e-1
0.80	19.10	2.2e-5	19.93	8.9e-3	18.64	1.6e-1
0.90	16.16	8.1e-4	16.33	2.9e-3	16.82	1.3e-1
1.0	12.50	8.4e-4	12.53	7.4e-4	15.40	1.1e-1
1.1	11.52	1.1e-3	11.55	4.3e-3	14.52	1.1e-1
1.20	11.93	2.6e-3	12.26	1.0e-2	14.18	1.1e-1
1.25	12.30	3.6e-3	12.62	1.5e-2	14.12	1.1e-1

Table: Local volatility model implied vols $\sigma_{imp,LV}$, and differences $\varepsilon_{imp} = |\sigma_{imp,LV} - \sigma_{imp,SLV}|$ between SLV and LV implied volatilities for a range of call options

SLV Calibration Code

- Code will eventually be available in the NAG Library
- Advance access can be arranged
- Feedback is welcome!