SVDNF: An R Package for Estimating Stochastic Volatility Models with Jumps Using Discrete Nonlinear Filtering

Louis Arsenault-Mahjoubi
Simon Fraser University

Jean-François Bégin
Simon Fraser University

Mathieu Boudreault
Université du Québec à Montréal

Abstract

The R package **SVDNF** provides a comprehensive toolset for estimating stochastic volatility models with jumps using discrete nonlinear filtering. This package allows users to simulate and estimate various stochastic volatility models, including those with return and volatility jumps and asset return predictors, leveraging advanced maximum likelihood estimation techniques. The package's primary contributions lie in its flexibility to handle both built-in and user-defined models that include jumps, making it a valuable resource for financial econometricians and statisticians. The article details the package's functionality, including its ability to handle complex stochastic volatility models, execute efficient filtering and estimation processes, and offer user-friendly interfaces for model customization and simulation. Through detailed examples and simulation studies, we demonstrate the package's effectiveness, highlighting its potential to improve the estimation efficiency in financial time series analysis.

Keywords: predictive-update algorithm, discrete nonlinear filtering, stochastic volatility models, maximum likelihood estimation.

1. Introduction

Financial time series exhibit a number of well-known stylized facts such as volatility clustering and heavy tails (see, e.g., Cont 2001; Eraker, Johannes, and Polson 2003). Various families of models have been developed in the research literature to capture the bulk of these features. For instance, autoregressive conditional heteroskedasticity (ARCH) and generalized ARCH (GARCH) dynamics *à la* Engle (1982) and Bollerslev (1986), respectively, are some of the first attempts to capture volatility clustering. The latter models allow for volatility clustering by letting the future (one-step-ahead) variance be a deterministic function of past observations.

To better capture the variance dynamics, a related literature has developed around stochastic volatility (SV) models which allow for a time-varying variance process that has a probabilistic relationship with returns—not deterministic like ARCH and GARCH dynamics. SV models are either formulated directly in discrete time or through continuous-time stochastic differential equations, which are discretized in practical applications. The discrete-time SV model of Taylor (1986) and the continuous-time SV model of Heston (1993) are some of the first attempts at probabilistic modelling of volatility.

Later frameworks introduced jumps to capture the behaviour of markets during financial crises. For example, in the continuous-time literature, Bates (1996) presented an affine SV model with return jumps. A similar return jump specification is found in the discrete-time model of Pitt, Malik, and Doucet (2014). Volatility dynamics also allow for discontinuities: indeed, Duffie, Pan, and Singleton (2000) proposed a flexible affine modelling framework with both return and volatility jumps in continuous time. The inclusion of jumps brought important improvements to model fit as it helps to capture the heavy tails of the return distribution and helps with rapid increases in the volatility factor via volatility jumps (see, e.g., Eraker, Johannes, and Polson 2003; Eraker 2004).

Another improvement in the continuous-time literature relates to nonaffine dynamics: nonaffine models without jumps can be found in the continuous-time literature in the works of Lewis (2000), Jones (2003), and Chacko and Viceira (2003). Nonaffine specifications have been investigated by some over the years; for instance, Christoffersen, Jacobs, and Mimouni (2010) studied continuous-time nonaffine models with fixed constant elasticity of variance (CEV) parameters and return jumps. Evidence suggests that including jumps significantly improves the fit of one-factor nonaffine models (see, e.g., Kaeck and Alexander 2012; Durham 2013; Ignatieva, Rodrigues, and Seeger 2015).[1]

SV models both with or without jumps have the ability to fit the return data better; nonetheless, they require an estimation methodology that is more computationally cumbersome than standard GARCH-type models. Specifically, computational difficulties arise because volatility is assumed to be a latent Markov process that is oftentimes nonlinear and non-Gaussian. Moreover, because the jumps are also unobserved, they must be inferred along with the volatility factor in the estimation procedure. Not only are the estimation methods computationally intensive, but there is also a wide array of options available in the literature to choose from. Bos (2012) argued that one of the reasons for the popularity of GARCH-type models over SV models is the fact that GARCH models have a common and efficient estimation procedure, while there exist multiple procedures to estimate SV models without jumps. This is also true when estimating SV models with jumps.

In the frequentist context, filters are often used to evaluate the likelihood and estimate the model parameters of SV models with return and volatility jumps. Many recent contributions have used *stochastic* filters to capture the posterior distribution of latent factors and compute the likelihood function. The most frequently employed approach in this stochastic paradigm is the sequential Monte Carlo (SMC) sampler (also known as the particle filter) as proposed by Gordon, Salmond, and Smith (1993). This method relies on a large number of particles that are propagated using the transition and measurement densities to approximate the posterior distribution of the latent states and the likelihood function (see, e.g., Johannes, Polson, and Stroud 2009; Christoffersen, Jacobs, and Mimouni 2010; Pitt, Malik, and Doucet 2014; Bardgett, Gourier, and Leippold 2019; Bégin, Amaya, Gauthier, and Malette 2020; Amaya, Bégin, and Gauthier 2022; Dufays, Jacobs, Liu, and Rombouts 2023, for applications of particle filtering).

However, there are two main issues with particle-based methods. First, these methods are computationally demanding and require high computational power for implementation, even for one-factor stochastic volatility models (see, e.g., Hurn, Lindsay, and McClelland 2015).

[1] Other authors have proposed the use of multi-factor SV models (see, e.g., Gallant, Hsu, and Tauchen 1999; Kaeck and Alexander 2012; Andersen, Fusari, and Todorov 2015; Bégin and Boudreault 2021, among others). These models have the potential to capture richer variance dynamics in theory; they are also out of the scope of this article.

Second, the likelihood function is generally not smooth in the parameter space when using naive resampling algorithms, which adds complexity to parameter estimation.[2]

Deterministic filters generally rely on approximate linearization methods or numerical integration techniques. For example, among the linearization methods, one of the most prevalent techniques is a generalization of the Kalman filter known as the extended Kalman filter (EKF); applications of the EKF to SV models can be found in Trolle and Schwartz (2009) and Wang, He, Zhao, and Zuo (2017), among others.

Another deterministic approach is to rely on numerical integration to evaluate the likelihood function recursively in the prediction and update steps. This method—the discrete nonlinear filter or DNF—was first introduced by Kitagawa (1987) and is the object of this very paper. The method involves computing integrals by generating a discrete grid of potential values for the latent variables and determining the associated posterior probability for each node in the grid. The DNF has mostly been used for SV models without jumps (see, e.g., Watanabe 1999; Clements, Hurn, and White 2006; Langrock, MacDonald, and Zucchini 2012) until the recent work of Bégin and Boudreault (2021). The latter authors apply the DNF to complex SV models with simultaneous return and volatility jumps as well as a stochastic jump intensity component (i.e., an additional persistent latent factor). In their study, they conclude that the DNF provides accurate likelihood evaluations that are faster than that of the bootstrap particle filter.

Another deterministic approach involves using numerical integration to recursively evaluate the likelihood function during the prediction and update steps. This method, known as the discrete nonlinear filter (DNF), was first introduced by Kitagawa (1987). It involves computing integrals by generating a discrete grid of potential values for the latent variables and determining the posterior probability for each node in the grid. Fridman and Harris (1998) applied this method to financial econometric models, specifically for SV parameter estimation. Other variations have been used by Watanabe (1999), Clements, Hurn, and White (2006), and Langrock, MacDonald, and Zucchini (2012). More recently, Bégin and Boudreault (2021) extended Kitagawa's methodology to SV models with jumps in return and volatility, demonstrating that the method is faster than the SMC sampler and yields a smooth likelihood, making it suitable for parameter estimation.

Bayesian methods are also sometimes used to estimate SV models with return and volatility jumps, with most implementations relying on Markov chain Monte Carlo (MCMC) samplers. Simple implementations of such MCMC samplers in the literature combined Gibbs sampling with the Metropolis–Hastings algorithm to obtain the posterior distribution of the model parameters and the volatility. We find MCMC samplers applied to discrete-time models in the work of Jacquier, Polson, and Rossi (1994), Kim, Shephard, and Chib (1998), Omori, Chib, Shephard, and Nakajima (2007), and to continuous-time models in Eraker (2001) Eraker, Johannes, and Polson (2003), and Eraker (2004), among others. Other authors have also combined standard MCMC tools with some of the filters mentioned above (see, e.g., Andrieu, Doucet, and Holenstein 2010).[3]

[2] Malik and Pitt (2011) proposed a resampling method to address the smoothness issue. Nevertheless, as stated by Creal (2012), the advantage of this resampling method diminishes when moving to multifactor specifications.

[3] Other methods have been applied to SV models without jumps. For example, Martino, Aas, Lindqvist, Neef, and Rue (2011) and de Zea Bermudez, Marín, Rue, and Veiga (2021) applied integrated nested Laplace approximations (INLA) to SV models without jumps. This frequentist approach is implemented in the **stoch-**

In addition to SV models having different estimation methods than GARCH-type frameworks, Bos (2012) argued that the adoption of SV models has been slow due to the lack of freely available software packages for their estimation. Over the last ten years, however, some resources have been made available by the community. MCMC methods for discrete-time SV models without jumps can be found in the **stochvol** (Kastner 2016) and **ASV** (Omori 2022) R packages. While both packages utilize mixture samplers as in Kim, Shephard, and Chib (1998) and Omori, Chib, Shephard, and Nakajima (2007), they offer different extensions to these methods. On the one hand, the **stochvol** package utilizes the MCMC sampling scheme detailed in Kastner and Frühwirth-Schnatter (2014), which employs the ancillarity-sufficiency interweaving strategy (ASIS) sampling method of Yu and Meng (2011) and the "all without a loop" (AWOL) sampling, a technique based on the Cholesky factorization described in McCausland, Miller, and Pelletier (2011).[4] On the other hand, the **ASV** R package's MCMC algorithm is based on an extension of the mixture sampler of Omori, Chib, Shephard, and Nakajima (2007); it also allows users to estimate these SV models using the particle filter. R packages **bvarsv**, **bsvars**, and **shrinkTVP** also provide Bayesian analyses via MCMC schemes for SV models but in the context of larger modelling frameworks (i.e., vector autoregressive models, time-varying parameter models, and structural vector autoregressive models); for more details on these packages, see Krueger (2015), Woźniak (2024), and Knaus, Bitto-Nemling, Cadonna, and Frühwirth-Schnatter (2021), respectively. The **nimbleSMC** R package offers a flexible framework where users can apply the particle filter to a wide class of state-space models, which includes some SV models as demonstrated in Michaud, de Valpine, Turek, Paciorek, and Nguyen (2021).[5] Similar flexible particle filtering scripts can be found in C++ (Johansen 2009; Brown 2020), in MATLAB (Chen, Lee, Budhiraja, and Mehra 2007), and in Python (Nordh 2017). We also find an implementation of the particle filter in Ox and MATLAB for simple SV models in the materials accompanying Creal (2012) on the author's website.

As for the DNF, there are some replication scripts available in MATLAB (Bégin and Boudreault 2021) and in R (Langrock, MacDonald, and Zucchini 2012; Zucchini, MacDonald, and Langrock 2016). However, there are no flexible user-friendly packages. The R codes of Langrock, MacDonald, and Zucchini (2012) and Zucchini, MacDonald, and Langrock (2016) offer implementations of the DNF to particular discrete-time SV models without jumps. The replication materials in Bégin and Boudreault (2021) give scripts for specific affine jump-diffusion models only.

The present article discusses the **SVDNF** package for R—a powerful tool designed for the estimation of SV models with jumps using the DNF—and aims to help users easily apply the DNF to a wide class of SV models that encompasses the most popular one-factor continuous-time and discrete-time models. Unlike most implementations cited above, **SVDNF** allows for general return and volatility jump distributions; this is the first contribution of our article. With the package, users are able to simulate observations from and estimate customized SV models, which can include return and volatility jumps as well as affine or nonaffine volatility

volTMB R package (Wahl 2020). For a discussion and comparison of various methods (either Bayesian or frequentist) applied to the basic SV model (besides INLA), we refer the reader to Bos (2012).

[4] The **factorstochvol** R package of Hosszejni and Kastner (2021) makes similar procedures available for multivariate SV model estimation.

[5] In addition to frequentist estimation, the **nimbleSMC** package allows users to perform Bayesian estimation by combining the particle filter with MCMC (pMCMC) algorithms via the methods proposed in Andrieu, Doucet, and Holenstein (2010).

dynamics. This is our second contribution.

The **SVDNF** package offers significant flexibility by enabling users to create custom models through the `dynamicsSVM` function, where they can specify their own functions and jump distributions. The package also contains well-known models that are readily available. For these built-in models, the package automatically sets the necessary functions, covering three standard discrete-time SV models, three SV jump-diffusion models, and a factor model with SV.

Key functionalities of the **SVDNF** package include the `DNF` function, which applies the DNF algorithm to provide likelihood evaluations and filtering distribution estimates, and the `DNFOptim` function, which uses optimization to find the maximum likelihood estimates of the model parameters. Additionally, the `modelSim` function allows for the simulation of volatility and returns from either built-in or custom model dynamics. This combination of features provides a user-friendly and versatile tool for researchers and practitioners in financial econometrics and risk management, making it easier to apply advanced filtering and estimation techniques to a wide range of SV models, including those with complex jump dynamics.

The remainder of this article is organized as follows. Section 2 introduces a general framework for one-factor SV models. Section 3 outlines how the DNF is applied to this general SV model framework. Section 4 describes the various functions available in the package and gives some examples of the package's functions on simulated data. Section 5 presents a simulation study to demonstrate its effectiveness. Section 6 displays the package's ability to estimate and compare the fit of a variety of models on real-world data. Finally, Section 7 summarizes the article and provides concluding remarks.

2. Stochastic volatility models

We fix a filtered probability space $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ and a filtration $\mathbb{F} = \{\mathcal{F}_t : t \in \{1, 2, ..., T\}\}$ satisfying the usual conditions. Let y_t be the time-t observed return on a security and x_t be the time-t unobserved volatility factor.[6] The return and volatility factor dynamics are given by the following equations:

$$\begin{aligned} y_t &= \mathbf{F}_t \mathbf{c}^\top + \mu^y(x_{t-1}) + \sigma^y(x_{t-1})\,\varepsilon_t^y + j_t^y, \\ x_t &= \mu^x(x_{t-1}) + \sigma^x(x_{t-1})\,\varepsilon_t^x + j_t^x, \end{aligned}$$

where

- $\mathbf{F}_t c^\top$ allows for a portion of the return average to be explained by market factors, captured by the row vector $\mathbf{F}_t = [\ F_{t,1} \quad ... \quad F_{t,d}\ ]$, with $\mathbf{c}^\top$ being a column vector of regression coefficients $c_1, ..., c_d$,
- μ^y: $\mathbb{R} \to \mathbb{R}$ is a function representing the portion of the average return that depends on the volatility factor,
- σ^y: $\mathbb{R} \to \mathbb{R}_+$ is a function of the volatility factor that provides the conditional return standard deviation,
- $\varepsilon_t^y \sim \mathcal{N}(0, 1)$ is the time-t return innovation modelled by a standard Gaussian random variable,

[6] In some cases, it is more practical to define y_t as an excess return (e.g., when working with factor models such as Model 7 in Section 2.3). We make this distinction explicit when needed.

- j_t^y is the time-t return jump random variable,
- μ^x: $\mathbb{R} \to \mathbb{R}$ is a function of the volatility factor that gives the conditional volatility factor average,
- σ^x: $\mathbb{R} \to \mathbb{R}_+$ is a function of the volatility factor that provides the conditional volatility factor standard deviation,
- $\varepsilon_t^x \sim \mathcal{N}(0,1)$ is the time-t volatility factor innovation modelled by a standard Gaussian random variable, and
- j_t^x is the time-t volatility factor jump random variable.

To capture the leverage effect, we allow ε_t^y and ε_t^x to be correlated; that is, $\text{Corr}\left[\varepsilon_t^y, \varepsilon_t^x\right] = \rho$. We assume that both processes jump at the same time and that there are n_t jumps at time t. Here, the discrete random variables n_t follow the same distribution for $t = 1, ..., T$. In practice, the jump components are often modelled via two different assumptions: the first specification relies on a Poisson distribution to model the number of jumps, and the second specification uses a Bernoulli distribution. These specifications give rise to the following assumptions on j_t^y and j_t^x:

1. Compound Poisson jump components: $j_t^y = \sum_{n=1}^{n_t} z_{t,n}^y$ and $j_t^x = \sum_{n=1}^{n_t} z_{t,n}^x$, where the number of jumps is distributed according to a Poisson random variable such that $n_t \sim \text{Poi}(\lambda)$, $\{z_{t,n}^y\}_{n\in\mathbb{N}}$ are return jump sizes, and $\{z_{t,n}^x\}_{n\in\mathbb{N}}$ are volatility factor jump sizes.
2. Bernoulli jump components: $j_t^y = n_t\, z_{t,1}^y$ and $j_t^x = n_t\, z_{t,1}^x$, where n_t is equal to one if there is a jump and zero otherwise, or that $n_t \sim \text{Ber}(p)$.

In all the cases, we assume that $z_{t,n}^y$ are normally distributed with $z_{t,n}^y \sim \mathcal{N}(\alpha + \rho_z\, z_{t,n}^x, \delta^2)$ and that $z_{t,n}^x$ are exponentially distributed with $z_{t,n}^x \sim \text{Exp}(\nu)$ for all $n \in \mathbb{N}$. This very general jump specification embeds many well-known models in the literature; for example, the SV models with jumps of Duffie, Pan, and Singleton (2000), Christoffersen, Jacobs, and Mimouni (2010), Pitt, Malik, and Doucet (2014), and Ignatieva, Rodrigues, and Seeger (2015) fall within this framework.[7]

2.1. Standard stochastic volatility models

The standard discrete-time SV models of Taylor (1986) with or without leverage and the SV model with leverage and return jumps of Pitt, Malik, and Doucet (2014) can be obtained from the above return and volatility factor equations.

1. SV model of Taylor (1986):
$$\begin{aligned} \mu^y(x_{t-1}) &= 0, & \sigma^y(x_{t-1}) &= \exp\left(\tfrac{x_{t-1}}{2}\right), & j_t^y &= 0, \\ \mu^x(x_{t-1}) &= \theta + \phi\left(x_{t-1} - \theta\right), & \sigma^x(x_{t-1}) &= \sigma, & j_t^x &= 0, \end{aligned}$$
where $\mathbf{F}_t c^\top$ and ρ are zero.
2. SV with leverage: same as 1 but with $\rho \in (-1, 1)$.
3. SV with leverage and return jumps of Pitt, Malik, and Doucet (2014): same as 2 but with Bernoulli return jumps such that $n_t \sim \text{Ber}(p)$ and $j_t^x = 0$ for all t.[8]

[7] The proposed framework focuses on Gaussian innovations, similar to those used in most of the literature. Extending the derivations for non-Gaussian unconditional innovation distributions would require that the distribution of return and variance innovations, as well as jumps, belong to the same family of distributions in addition to being closed in convolution. We leave this interesting question for future research.

[8] Note that unlike our SV with leverage and return jumps, the model of Pitt, Malik, and Doucet (2014)

Table 1 summarizes the discrete-time SV model parameters and their descriptions. These models could be modified by adding non-zero return drift function $\mu^y(x_{t-1})$ or volatility factor jumps. Also, Poisson-based jumps could be used instead of Bernoulli ones.

	Description	Support	Taylor	Taylor with leverage	Pitt, Malik, and Doucet
ϕ	Volatility persistence	$[-1,\, 1]$	✓	✓	✓
θ	Long-run mean volatility	$\mathbb{R}$	✓	✓	✓
σ	Volatility of volatility	$\mathbb{R}_+$	✓	✓	✓
ρ	Noise term correlation	$[-1,\, 1]$		✓	✓
δ	Standard deviation of return jumps	$\mathbb{R}_+$			✓
α	Average of return jumps	$\mathbb{R}$			✓
p	Jump probability	$[0,\, 1]$			✓

Table 1: Parameter description for discrete-time SV models (Models 1–3).

2.2. Stochastic volatility jump-diffusion models

The discretized version of continuous-time models is obtained by using a discretization method (e.g., Euler–Maruyama, Milstein). Specifically, we rely on the full truncation scheme of Lord, Koekkoek, and Dijk (2010) in this package. The constant h is used in the discretization to represent the time step between two observations (e.g., $h = \frac{1}{252}$ for daily observations).

4. The SV model of Heston (1993):

$$\begin{aligned} \mu^y(x_{t-1}) &= \left(\mu - \tfrac{x_{t-1}}{2}\right) h & \sigma^y(x_{t-1}) &= \sqrt{h \max\left[0, x_{t-1}\right]}, & j_t^y &= 0, \\ \mu^x(x_{t-1}) &= x_{t-1} + \kappa\left(\theta - \max\left[0, x_{t-1}\right]\right) h, & \sigma^x(x_{t-1}) &= \sigma\sqrt{h \max\left[0, x_{t-1}\right]}, & j_t^x &= 0, \end{aligned}$$

   where $\mathbf{F}_t c^\top = 0$.
5. SV with return jumps *à la* Bates (1996): same as 4 but with Poisson return jumps such that $n_t \sim \text{Poi}(\omega\, h)$, and $\mu^y(x_{t-1}) = \left(\mu - \frac{x_{t-1}}{2} - \bar{\alpha}\omega\right) h$, where $\bar{\alpha} = \exp\left(\alpha + \frac{1}{2}\delta^2\right) - 1$ and $\omega = \lambda / h$ is the annualized jump intensity parameter.
6. SV with correlated return and volatility jumps proposed by Duffie, Pan, and Singleton (2000): same as 5 but with $j_t^x = \sum_{n=1}^{n_t} z_{t,n}^x$ and $\bar{\alpha} = \exp\left(\alpha + \frac{1}{2}\delta^2\right) / \left(1 - \nu\, \rho_z\right) - 1$, where ρ_z captures the correlation between the return and volatility jumps.

Table 2 summarizes the jump-diffusion SV model parameters and their descriptions. These models could be modified by having a return drift term proportional to the variance or different $\sigma^x(x_{t-1})$ functions; for example,

$$\sigma^x(x_{t-1}) = \sqrt{h}\sigma \max\left[0, x_{t-1}\right]^\beta,$$

where β could be $\frac{1}{2}$, 1, $\frac{3}{2}$ (see, e.g., the custom model example in Section 4.3), or simply a free parameter (see, e.g., the empirical results for CEV models in Section 6). Also, Bernoulli jumps could also be used here instead of the Poisson jump specification.

fixes the jump size parameter $\alpha = 0$.

	Description	**Support**	**Heston**	**Bates**	**Duffie, Pan, and Singleton**
μ	Mean return	$\mathbb{R}$	✓	✓	✓
α	Average of return jump	$\mathbb{R}$		✓	✓
δ	Standard deviation of return jumps	$\mathbb{R}_+$		✓	✓
ρ_z	Correlation between return and volatility jumps	$\mathbb{R}$			✓
ν	Average of volatility jumps	$\mathbb{R}_+$			✓
ω	Jump arrival intensity	$\mathbb{R}_+$		✓	✓
κ	Volatility mean reversion	$\mathbb{R}_+$	✓	✓	✓
θ	Long-run mean volatility	$\mathbb{R}_+$	✓	✓	✓
σ	Volatility of volatility	$\mathbb{R}_+$	✓	✓	✓
ρ	Noise term correlation	$[-1, 1]$	✓	✓	✓

Table 2: Parameter description for jump-diffusion SV models (Models 4–6).

2.3. Factor models with stochastic volatility

7. The CAPM with stochastic volatility (CAPM-SV) model:

$$\begin{aligned} \mathbf{F}_t c^\top &= c_0 + (R_t^m - R_t^f)c_1, & \sigma^y(x_{t-1}) &= \exp\left(\tfrac{x_{t-1}}{2}\right), & j_t^y &= 0, \\ \mu^x(x_{t-1}) &= \theta + \phi\,(x_{t-1} - \theta)\,, & \sigma^x(x_{t-1}) &= \sigma, & j_t^x &= 0, \end{aligned}$$

where $\mu^y(x_{t-1}) = \rho = 0$ and the observation y_t is the time-t asset excess return over the market, R_t^m. The variable R_t^f is risk-free rate at time t. The coefficient c_0 represents the measure of the active return on an investment (i.e., the asset's alpha), and the coefficient c_1 represents the asset return's sensitivity to excess market returns (i.e., the market beta). Table 3 summarizes the factor models with SV model parameters and their descriptions.

These models can be extended by incorporating additional factors (see, e.g., Fama and French 1992, 2015), such as small market capitalization minus big (SMB) and high book-to-market ratio minus low (HML), or adding terms in the drift function μ^y. For instance, using the equations

$$\mu^y(x_{t-1}) = d\,\zeta^2 \exp\left(x_{t-1}\right) \quad \text{and} \quad \sigma^y(x_{t-1}) = \zeta\,\exp\left(\frac{x_{t-1}}{2}\right),$$

where d measures the effect of the volatility on the average return and ζ is a positive scaling factor, results in the CAPM with stochastic volatility in the mean (SVM) model, as proposed by Koopman and Hol Uspensky (2002).

3. Discrete nonlinear filtering-based method

The discrete nonlinear filter computes:

1. The time-t filtering distribution

$$p_\Theta(x_t \mid y_{1:t}) \equiv p(x_t \mid y_{1:t}, \Theta),$$

where Θ is the set of model parameters, and

	Description	Support	CAPM-SV
c_0	Excess return over market returns	$\mathbb{R}$	✓
c_1	Sensitivity to excess market returns	$\mathbb{R}$	✓
ϕ	Volatility persistence	$[-1, 1]$	✓
θ	Long-run mean volatility	$\mathbb{R}$	✓
σ	Volatility of volatility	$\mathbb{R}_+$	✓

Table 3: Parameter description for factor models with SV models (Model 7).

2. The likelihood function via iterated numerical integration across the latent states.

This methodology introduced by Kitagawa (1987) is applied to jump-diffusion stochastic volatility models with up to two persistent latent factors in Bégin and Boudreault (2021). Although the DNF computational costs increase exponentially with the number of latent variables, they found that the method can be used for quick and accurate likelihood evaluations in this context. We present a brief outline of the algorithm in this section.

We compute the likelihood $\mathcal{L}(\Theta \mid y_{1:T})$ for a parameter set Θ iteratively using the following decomposition, as is commonly done in time-series analysis:

$$\mathcal{L}(\Theta \mid y_{1:T}) \equiv p_\Theta(y_{1:T}) = p_\Theta(y_1) \prod_{t=2}^{T} p_\Theta(y_t \mid y_{1:t-1}). \tag{1}$$

The time-t likelihood contribution $p_\Theta(y_t \mid y_{1:t-1})$ is computed by integrating the joint density $p_\Theta(y_t, n_t, j_t^x, x_t, x_{t-1} \mid y_{1:t-1})$ over the latent factors' domain by iteratively conditioning on each variable,

$$\begin{aligned} p_\Theta(y_t \mid y_{1:t-1}) = \sum_{n_t=0}^{\infty} \iiint_{\mathbb{R}^3_+} & p_\Theta(y_t \mid x_t, x_{t-1}, j_t^x, n_t) \\ & \times p_\Theta(x_t \mid x_{t-1}, j_t^x, n_t)\, p_\Theta(j_t^x \mid n_t) p_\Theta(n_t) \\ & \times p_\Theta(x_{t-1} \mid y_{1:t-1})\, dx_t\, dx_{t-1}\, dj_t^x, \end{aligned} \tag{2}$$

where n_t can be either Poisson, Bernoulli, or any user-chosen discrete distribution, $j_t^x \mid n_t$ is Gamma distributed such that $j_t^x \mid n_t \sim \Gamma(n_t, \nu)$ with scale parameter ν,

$$\begin{aligned} & x_t \mid x_{t-1}, n_t, j_t^x \sim \mathcal{N}\left(\mu^x(x_{t-1}) + j_t^x, \sigma^x(x_{t-1})^2\right), \quad \text{and} \\ & y_t \mid x_t, x_{t-1}, n_t, j_t^x \sim \mathcal{N}(\mu_t, \sigma_t^2) \end{aligned}$$

for which the mean and variance parameters are given by

$$\begin{aligned} & \mu_t = \mathbf{F}_t \mathbf{c}^\top + \mu^y(x_{t-1}) + \rho\, \sigma^y(x_{t-1}) \left(\frac{x_t - \mu^x(x_{t-1}) - j_t^x}{\sigma^x(x_{t-1})} \right) + n\alpha + \rho_z j_t^x, \quad \text{and} \\ & \sigma_t^2 = (1-\rho^2)\sigma^y(x_{t-1})^2 + n\delta^2, \end{aligned}$$

respectively. Similarly, we can obtain the filtering density $p_\Theta(x_t \mid y_{1:t})$ by computing the following integral:

$$p_\Theta(x_t \mid y_{1:t}) = \frac{1}{p_\Theta(y_t \mid y_{1:t-1})} \Bigg(\sum_{n_t=0}^{\infty} \iint_{\mathbb{R}_+^2} p_\Theta(y_t \mid x_t, x_{t-1}, j_t^x, n_t)\, p_\Theta(x_t \mid x_{t-1}, j_t^x, n_t) \times p_\Theta(x_{t-1} \mid y_{1:t-1})\, p(j_t^x \mid n_t)\, p_\Theta(n_t)\, dx_{t-1}\, dj_t^x \Bigg). \tag{3}$$

Algorithm 1 describes how the likelihood function and filtering densities are obtained.

Algorithm 1 Discrete Nonlinear Filter

```
1: initialize by selecting density $p_\Theta(x_0)$
2: for $j = 1, 2, \ldots, T$ do
3:     compute $p_\Theta(y_t \mid y_{1:t-1})$ by numerical integration using Equation (2)
4:     compute $p_\Theta(x_t \mid y_{1:t})$ by numerical integration using Equation (3)
5: end for
6: compute $p_\Theta(y_{1:T})$ using Equation (1)
```

Although the prediction distribution $p_\Theta(x_{t+1} \mid y_{1:t})$ is not explicitly computed in Algorithm 1, it can be retrieved from the filtering distribution using the transition density as

$$p_\Theta(x_{t+1} \mid y_{1:t}) = \int_{\mathbb{R}_+} p_\Theta(x_{t+1} \mid x_t) p_\Theta(x_t \mid y_{1:t}) dx_t, \tag{4}$$

with

$$p_\Theta(x_{t+1} \mid x_t) = \sum_{n_t=0}^{\infty} \int_{\mathbb{R}_+} p_\Theta(x_{t+1} \mid x_t, j_{t+1}^x, n_{t+1}) p_\Theta(x_t \mid y_{1:t})\, p(j_{t+1}^x \mid n_t)\, p_\Theta(n_{t+1})\, dj_{t+1}^x. \tag{5}$$

For the numerical integration in Algorithm 1, we apply the quadrature method used in Langrock, MacDonald, and Zucchini (2012) and Bégin and Boudreault (2021). This rule states that for two functions f_1 and f_2 and two (close enough) constants a and b, we have

$$\int_a^b f_1(x)\, f_2(x)\, dx \approx f_1(c) \int_a^b f_2(x)\, dx,$$

where c is the midpoint of the interval $[a, b]$. If f_2 is a probability density function and F_2 is its associated cumulative distribution function, then this rule implies that

$$\int_a^b f_1(x)\, f_2(x)\, dx \approx f_1(c)(F_2(b) - F_2(a)).$$

We now define the default intervals and midpoints used in the package. Let N be the number of volatility factor midpoints, K the number of jump size midpoints, and R the maximum number of jumps per time step. The grids for the volatility factor and the volatility jumps midpoints are defined as $\mathbf{X} = \begin{bmatrix} x^{(1)} & \ldots & x^{(N)} \end{bmatrix}$ and $\mathbf{J} = \begin{bmatrix} J^{(1)} & \ldots & J^{(K)} \end{bmatrix}$, respectively. We define the intervals as

$$\mathbf{X}^{(i)} = \left[\frac{x^{(i-1)} + x^{(i)}}{2}, \frac{x^{(i)} + x^{(i+1)}}{2}\right), \quad i = 1, \ldots, N,$$

$$\mathbf{J}^{(l)} = \left[\frac{J^{(l-1)} + J^{(l)}}{2}, \frac{J^{(l)} + J^{(l+1)}}{2}\right), \quad l = 1, \ldots, K,$$

where $x^{(0)} = x^{(1)} - \left(x^{(2)} - x^{(1)}\right)$, $j^{(0)} = -j^{(1)}$, and $x^{(N+1)} = j^{(K+1)} = \infty$.

With these, we estimate the marginal likelihood contribution of Equation (2) with

$$\begin{aligned}\hat{p}_\Theta(y_t \mid y_{1:t-1}) = \sum_{n_t=0}^{R} \sum_{i_t=1}^{N} \sum_{i_{t-1}=1}^{N} \sum_{l_t=1}^{K} & p_\Theta(y_t \mid x^{(i_t)}, x^{(i_{t-1})}, J^{(l_t)}, n_t) \\ & \times p_\Theta(x_t \in \mathbf{X}^{(i_t)} \mid x^{(i_{t-1})}, J^{(l_t)}, n_t) \\ & \times p_\Theta(j_t^x \in \mathbf{J}^{(l_t)} \mid n_t)\, p_\Theta(n_t) p_\Theta(x^{(i_{t-1})} \mid y_{1:t-1}).\end{aligned}$$

Then, using Equation (3), we approximate the filtering density with

$$\begin{aligned}\hat{p}_\Theta(x^{(i_t)} \mid y_{1:t}) = \frac{1}{\hat{p}_\Theta(y_t \mid y_{1:t-1})} \sum_{n_t=0}^{R} \sum_{i_{t-1}=1}^{N} \sum_{l_t=1}^{K} & p_\Theta(y_t \mid x^{(i_t)}, x^{(i_{t-1})}, J^{(l_t)}, n_t) \\ & \times p_\Theta(x_t \in \mathbf{X}^{(i_t)} \mid x^{(i_{t-1})}, J^{(l_t)}, n_t) \\ & \times p_\Theta(j_t^x \in \mathbf{J}^{(l_t)} \mid n_t)\, p_\Theta(n_t) p_\Theta(x^{(i_{t-1})} \mid y_{1:t-1}).\end{aligned}$$

From the filtering density estimate above, we the prediction density with

$$\hat{p}_\Theta(x^{(i_{t+1})} \mid y_{1:t}) = \sum_{i_t=1}^{N} \hat{p}_\Theta(x^{(i_{t+1})} \mid x^{(i_t)}) \hat{p}_\Theta(x^{(i_t)} \mid y_{1:t}),$$

with

$$\begin{aligned}\hat{p}_\Theta(x^{(i_{t+1})} \mid x^{(i_t)}) = \sum_{n_{t+1}=0}^{R} \sum_{l_{t+1}=1}^{K} & p_\Theta(x_{t+1} \in \mathbf{X}^{(i_{t+1})} \mid x^{(i_t)}, J^{(l_{t+1})}, n_{t+1}) \\ & \times p_\Theta(j_{t+1}^x \in \mathbf{J}^{(l_{t+1})} \mid n_{t+1})\, p_\Theta(n_{t+1}).\end{aligned}$$

The midpoints $\mathbf{X}$ and $\mathbf{J}$ must be selected by users when using a custom—not built-in—model. However, for built-in models, users have the option of using the package default grids.

For the default grids of built-in models, we follow in essence the approach outlined in Bégin and Boudreault (2021) where, for a given latent factor H, the integration bounds are obtained using

$$E_H \pm (3 + \log(L)) \sqrt{V_H},$$

where L is the number of nodes used in the quadrature grid. Moreover,

$$E_H = \lim_{j \to \infty} \mathrm{E}\left[H_j \mid \mathcal{F}_0\right] \quad \text{and} \quad V_H = \lim_{j \to \infty} \mathrm{Var}\left[H_j \mid \mathcal{F}_0\right]$$

are the long-run expected value and variance of process H, respectively.

Within these integration bounds, we want to place nodes in high-probability density regions. For the built-in discrete-time models, this translates into putting more nodes close to the long-run average E_H, which is set as the central node in the grid. Specifically, we increase the distance between nodes as we get further than E_H by creating an equidistant grid for the square root of the volatility factor and squaring it. Assuming that $E_H = \theta$ and $V_H = \frac{\sigma^2}{1-\phi^2}$, we have that

$$x^{(i)} = \begin{cases} \theta - \left(\frac{\lfloor N/2 \rfloor + 1 - i}{\lfloor N/2 \rfloor} \sqrt{(3+\log(N))\sqrt{\frac{\sigma^2}{1-\phi^2}}} \right)^2 & \text{if } i = 1, ..., \lfloor N/2 \rfloor + 1 \\ \theta + \left(\frac{i - \lfloor N/2 \rfloor - 1}{\lfloor N/2 \rfloor} \sqrt{(3+\log(N))\sqrt{\frac{\sigma^2}{1-\phi^2}}} \right)^2 & \text{if } i = \lfloor N/2 \rfloor + 2, ..., N \end{cases},$$

where the nodes of integration get sparser as they are further from the long-run mean log-volatility (i.e., from θ) and $\lfloor N/2 \rfloor$ is the floor function of $N/2$.

For the built-in SV jump-diffusion models, we use a similar trick and create a grid that a higher density of nodes close to zero. Assuming that $E_H = \theta$ and $V_H = \frac{\theta\sigma^2}{2\kappa}$, we have

$$x^{(i)} = \begin{cases} \max\left(\theta - (3+\log(N))\sqrt{\frac{\theta\,\sigma^2}{2\kappa}},\, x_{\min}\right) & \text{if } i = 1 \\ \left(\sqrt{x^{(1)}} + \left(\frac{i-1}{N-1}\right)\left(\sqrt{x^{(N)}} - \sqrt{x^{(1)}}\right)\right)^2 & \text{if } i = 2, \ldots, N-1 \\ \max\left(\theta + (3+\log(N))\sqrt{\frac{\theta\,\sigma^2}{2\kappa}},\, x_{\max}\right) & \text{if } i = N \end{cases},$$

and

$$J^{(l)} = \begin{cases} \max\left(\nu - (3+\log(K))\sqrt{\nu^2},\, 0\right) & \text{if } l = 1 \\ J^{(1)} + \left(\frac{l-1}{K-1}\right)\left(J^{(K)} - J^{(1)}\right) & \text{if } l = 2, \ldots, K-1 \\ \nu R + (3+\log(K))\sqrt{R\nu^2} & \text{if } l = K \end{cases},$$

where the intervals between the variance nodes get larger in i as they are uniformly distributed in the volatility (i.e., square root of the variance), and then squared to obtain the actual variance grid. Unlike the logarithm of the volatility used in discrete-time models, variance cannot be negative. To avoid negative variance node values, we set the lower bound of the built-in SV jump-diffusion models variance grid as the maximum of the usual grid lower bound and $x_{\min}$ which is set near zero in this package.[9] Moreover, numerical issues can occur if the variance node grid is too small and the nodes are near zero. To avoid these problems, the variance grid upper bound $x_{\max}$ is set to 0.05 in this package.

4. Package description and illustration

The **SVDNF** package contains four primary functions: `dynamicsSVM`, `DNF`, `DNFOptim`, and `modelSim`.

- `dynamicsSVM` defines the SV model dynamics by either choosing from a set of built-in model dynamics or using custom drift functions, diffusion functions, and jump distributions. This information—the functions and the parameters specifying the model—is stored in a `dynamicsSVM` object that is built by the function.

  For built-in models, the function automatically sets the appropriate drift, diffusion, and jump distribution functions. Users only need to specify the model type, and the function takes care of the rest. The built-in models are the three standard discrete-time SV models (Models 1–3), the three SV jump-diffusion models (Models 4–6), and

[9] The actual value of $x_{\min}$ is set to 10^{-7} to avoid divisions by 0.

the factor model with SV model (Model 7) described in Sections 2.1, 2.2, and 2.3, respectively.

For custom models, the user must specify functions μ^y, σ^y, μ^x, σ^x, and a jump distribution for the return and volatility dynamics presented in Section 2. These custom functions allow for a high degree of flexibility in modelling complex financial time series data. Objects with class `dynamicsSVM` have `DNF`, `DNFOptim`, `modelSim`, and `print` methods available.

- `DNF` applies the DNF of Kitagawa (1987) as per the implementation of Bégin and Boudreault (2021), summarized in Section 3 of this paper, to obtain likelihood evaluations and filtering distribution estimates for the model dynamics and given parameter values stored in the `dynamicsSVM` object. This function returns a list of class `SVDNF`, which contains the likelihood evaluation, the log-likelihood contribution at each step t, the grids used by the DNF for numerical integration, estimates of the filtering distribution at each step t, and the model dynamics used by the DNF. These computations are run in C++ thanks to the **Rcpp** package of Eddelbuettel and François (2011). The `SVDNF` objects allow for the following methods: `logLik`, `plot`, `predict`, and `print`.
- `DNFOptim` finds maximum likelihood estimates Θ for either built-in or custom model dynamics stored in a `dynamicsSVM` object using the `DNF` function. The maximization relies on R's `optim` function to find the maximum likelihood estimates. This function returns a `DNFOptim` object that contains two lists. The first list contains the values computed by `optim`; that is, the set of parameters found, the log-likelihood evaluated at those parameters, the number of `DNF` evaluations used in the optimizer, the error codes in case of convergence issues, any additional messages, and an estimate of the Hessian matrix (R Core Team 2019).[10] The second list is the `SVDNF` object obtained from running the `DNF` function at the MLE parameter values. `DNFOptim` objects allows for the `logLik`, `plot`, `predict`, `print`, and `summary` methods.
- `modelSim` generates return and variance series for a wide class of built-in or custom model dynamics stored in a `dynamicsSVM` object.

The relationships among the **SVDNF**'s packages primary functions are summarized in Figure 1. The package also offers the `extractVolPerc` function which returns a given percentile value of either the filtering or the prediction distribution at each time t for `DNFOptim` and `SVDNF` objects, and the `pars` function to print the parameters name in a `dynamicsSVM` object in the order that initial parameters should be passed to the `DNFOptim` function.

We now explore possible applications of the **SVDNF** package. We start by simulating two sets of returns using `modelSim`. The first time series showcases the use of built-in models, while the second illustrates how to use the framework of Section 2 to create a custom model.

4.1. Model simulation

Built-in model dynamics

To demonstrate how to use `modelSim`, we first generate data from a built-in model. For this example, we use a discretized version of the model of Duffie, Pan, and Singleton (2000) (i.e., Model 6 in Section 2.2). To start, we create a `dynamicsSVM` object where we set `model = DuffiePanSingleton` and keep the parameters to their default values. Then, we generate ten

[10] The estimate of the Hessian is only returned if the argument `hessian = TRUE` is passed to `optim`.

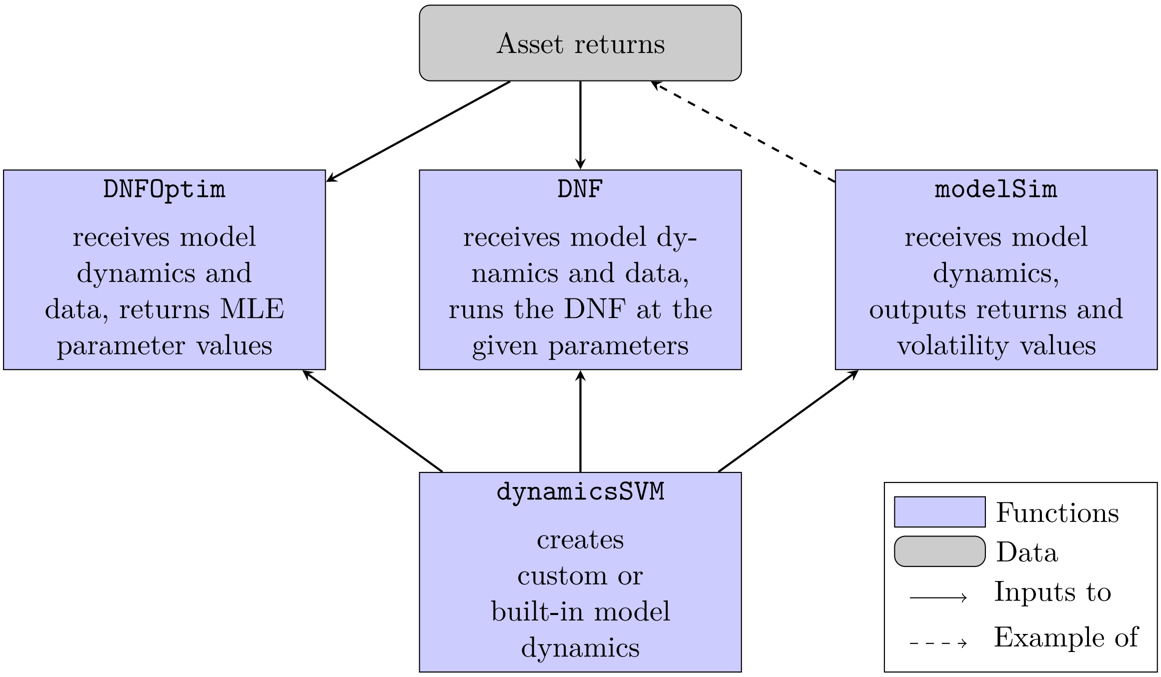
Asset returns
DNFOptim
receives model dynamics and data, returns MLE parameter values
DNF
receives model dynamics and data, runs the DNF at the given parameters
modelSim
receives model dynamics, outputs returns and volatility values
dynamicsSVM
creates custom or built-in model dynamics
Functions
Data
Inputs to
Example of

Figure 1: Flowchart of the various **SVDNF** package functions.

years of daily return data (i.e., 2,520 business days) using `modelSim` and plot the return and the volatility factor values, which is shown in Figure 2.

```
R> library("SVDNF")
R> set.seed(1)
R> DPS_mod <- dynamicsSVM(model = "DuffiePanSingleton")
R> DPS_sim <- modelSim(t = 2520, dynamics = DPS_mod)
R> head(DPS_sim$returns)
R> plot(DPS_sim$returns, type = "l", ylim = c(-0.1, 0.22), ylab = "Returns")
R> lines(DPS_sim$volatility_factor, col = "red")
```

Custom model dynamics

For the second example, we demonstrate the package's ability to create custom models. The custom dynamics that we define are based on a discretized version of the ONEN model of Christoffersen, Jacobs, and Mimouni (2010) with simultaneous and correlated return and volatility jumps. This model is similar to Model 6 in Section 2.2, except that

$$\begin{aligned}\mu^x(x_{t-1}) &= x_{t-1} + \kappa \max\left[0, x_{t-1}\right]\left(\theta - \max\left[0, x_{t-1}\right]\right)h, \quad \text{and}\\ \sigma^x(x_{t-1}) &= \sigma \max\left[0, x_{t-1}\right]\sqrt{h}.\end{aligned}$$

We now set the model parameters as follows:

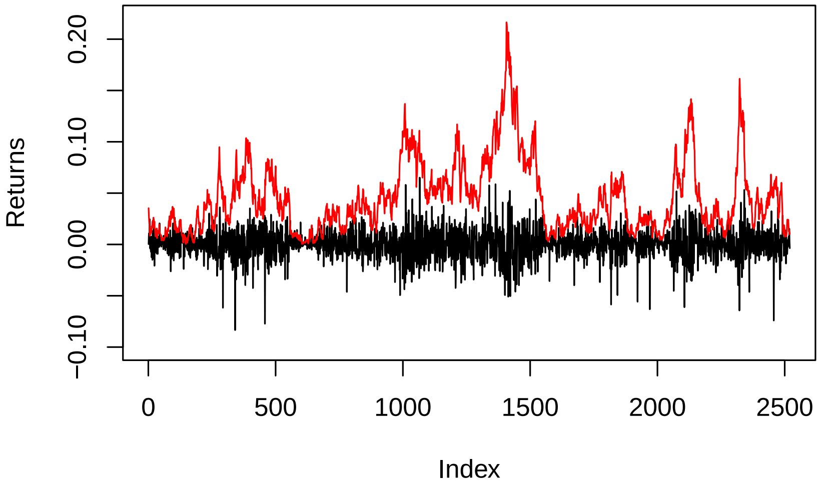

Figure 2: Simulated returns (in black) and volatility factors values (in red) from a discretized version of the model of Duffie, Pan, and Singleton (2000) with simultaneous return and volatility jumps.

```
R> h <- 1/252
R> mu <- 0.04; kappa <- 100; theta <- 0.05; sigma <- 2.3; rho <- -0.8
R> omega <- 5; alpha <- -0.025; delta <- 0.025; nu <- 0.004; rho_z <- -1
```

Then, we define the drift and diffusion functions while storing their parameters in lists. When defining custom model functions, it is important that the parameters in the lists are in the same order they appear in the drift and diffusion function definitions. Moreover, since the custom functions receive a vector of volatility factor values (i.e., all points of the grid $\mathbf{X}$), we must apply vectorized versions of certain built-in R functions instead of their scalar versions (e.g., pmax in the sigma_y function below instead of max).

```
R> mu_y <- function(x, mu, alpha, delta, omega, rho_z, nu){
+    alpha_bar <- exp(alpha + 0.5 * delta ^ 2) / (1 - rho_z * nu) - 1
+    return(h * (mu - x / 2 - alpha_bar * omega))
+ }
R> mu_y_params <- list(mu, alpha, delta, omega, rho_z, nu)
R> sigma_y <- function(x, sigma_y_params){
+    return(sqrt(h * pmax(x, 0)))
+ }
R> sigma_y_params <- list(0)
R> mu_x <- function(x, kappa, theta){
+    return(x + h * kappa * pmax(0, x) * (theta - pmax(0, x)))
+ }
R> mu_x_params <- list(kappa, theta)
```

```
R> sigma_x <- function(x, sigma){
+    return(sigma * sqrt(h) * pmax(0, x))
+ }
R> sigma_x_params <- list(sigma)
```

Next, we set our jump distribution, density function, and parameters:

```
R> jump_pois_d <- function(n, omega){return(dpois(n, h * omega))}
R> jump_pois_r <- function(n, omega){return(rpois(n, h * omega))}

R> jump_dist <- jump_pois_r
R> jump_density <- jump_pois_d
R> jump_params <- list(omega)
```

Finally, we pass all arguments to the `dynamicsSVM` function to store the dynamics of this custom model and use `modelSim` to simulate 2,520 observations from the ONEN model.

```
R> custom_mod <- dynamicsSVM(model = "Custom", mu_x = mu_x, mu_y = mu_y,
+    sigma_x = sigma_x, sigma_y = sigma_y, rho = rho,
+    mu_x_params = mu_x_params, mu_y_params = mu_y_params,
+    sigma_x_params = sigma_x_params, sigma_y_params = sigma_y_params,
+    jump_dist = jump_dist, jump_params = jump_params,
+    jump_density = jump_density, nu = nu, rho_z = rho_z,
+    alpha = alpha, delta = delta)
R> custom_sim <- modelSim(t = 2520, dynamics = custom_mod, init_vol = 0.05)
R> plot(custom_sim$returns, type = "l", ylim = c(-0.1, 0.22),
+    ylab = "Returns")
R> lines(custom_sim$volatility_factor, col = "red")
```

When creating custom models in the **SVDNF** package, identifiability issues can pose significant challenges. Identifiability refers to the ability to uniquely estimate model parameters based on the observed data. In complex SV models with multiple parameters and jumps, it can be difficult to distinguish the individual contributions of each parameter to the overall model behaviour, particularly when many parameters exert similar influences on the model. Ensuring identifiability typically requires careful model specification to help disentangle the effects of different parameters. Users should be vigilant when building custom models.

4.2. The filtering problem

We can estimate the log-likelihood and the filtering distribution for the data generated above with the `DNF` function. Throughout this section, we assume that the parameters used to generate the data are known and constant. Section 4.3 considers the case where parameters are unknown and uses MLE to obtain parameter estimates.

First, we evaluate the log-likelihood from the data generated using the built-in model by passing the simulated return data stored in `DPS_sim` and the dynamics object to the `DNF` function.[11]

[11] Note that `df=NULL` occurs due to the fact that the `DNF` function receives a fixed set of parameters. When using `DNFOptim`, we will get a degree of freedom value equal to the number of free parameters in the optimization.

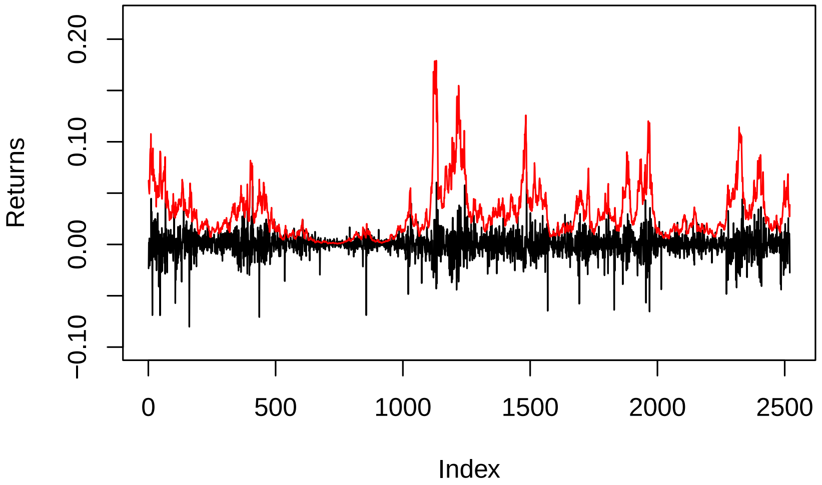

Figure 3: Simulated returns (in black) and volatility factors values (in red) from the ONEN model of Christoffersen, Jacobs, and Mimouni (2010) with simultaneous and correlated return and volatility jumps.

```
R> DPS_dnf <- DNF(dynamics = DPS_mod, data = DPS_sim$returns)
R> logLik(DPS_dnf)
```

```
 'log Lik.' 7542.083 (df=NULL)
```

By default, the grids used for numerical integration for the `DuffiePanSingleton` model are constructed as presented in Section 3. The arguments `N`, `K`, and `R` in the `DNF` function determine the size of the grids. The volatility factor is integrated over a grid of length `N`, while the integration for the volatility jump dimension uses a grid of length `K`. The maximum number of jumps per time step is `R`.

We can extract the prediction and filtering distributions from the `DPS_dnf` object above. The results for the particular time-point $t = 1000$ are shown in Figure 4. We plot both distributions (the dotted blue line for the prediction distribution and the magenta line for the filtering distribution) as well as the simulated volatility factor value at time 1000 (black vertical line).

```
R> tlim <- 1000
R> plot(x = DPS_dnf, tlim = tlim, type = "l",
+    ylab = "Volatility Factor", xlab = "Time")
R> abline(v = DPS_sim$volatility_factor[tlim], col = "black", lwd = 1.5)
R> legend("topright", legend = c("Prediction", "Filtering", "True Value"),
+    lwd = c(1.5, 1.5, 1.5), col = c("blue", "magenta", "black"),
+    lty = c(2, 1, 1))
```

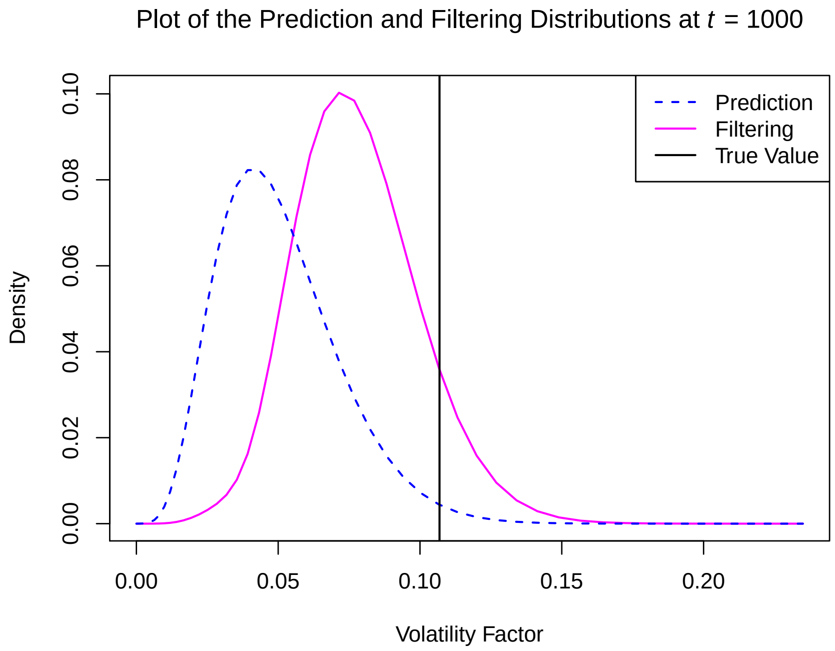

Figure 4: Prediction (dotted blue line) and filtering (solid magenta line) distributions from the DNF as well as the simulated value of the volatility factor (vertical black line).

We now apply the `DNF` function to the data generated from the custom model in Section 4.1. When using built-in models, we only need to pass the grid size arguments (i.e., `N`, `K`, and `R`) to the `DNF` function, and the function automatically creates grids adjusted to the dynamics of the built-in model. However, when working with custom models, we must define our own grids. Ideally, these grids cover the integration range of the variables and have more nodes in regions of high probability density. In this example, we create grids with uniformly spaced nodes for the volatility factor and volatility jumps for simplicity's sake.

```
R> N <- 75; R <- 1; K <- 25
R> var_mid_points <- seq(from = 0.000001, to = 0.2, length = N)
R> j_nums <- seq(from = 0, to = R, length = R + 1)
R> jump_mid_points <- seq(from = 0.00000001, to = 4 * nu, length = K)
R> grids <- list(var_mid_points = var_mid_points,
+    j_nums = j_nums, jump_mid_points = jump_mid_points)
```

Then, we pass our custom model dynamics and grids to the `DNF` function along with the ONEN model returns simulated in Section 4.1:

```
R> dnf_custom <- DNF(dynamics = custom_mod,
+    data = custom_sim$returns, grids = grids)
```

The `DNF` function results are passed to the `plot` function which shows the median along with 5^{th} and 95^{th} percentile estimates of the volatility factors filtering and prediction distributions.

We add the simulated volatility factor values to the plot to compare our estimates with their targets.

```
R> par(mfrow = c(1, 2), cex.main = 0.85)
R> plot(dnf_custom, upper_p = 0.05, lower_p = 0.95,
+    ylim = c(0, 0.25), type = "l",
+    ylab = "Volatility Factor", xlab = "Time")
R> matplot(y = custom_sim$volatility_factor,
+    add = TRUE, type = "l", col = "blue", lty = 1)
R> legend("topright", cex = 0.75,
+    legend = c("Median", "5th/95th Percentiles", "True Value"),
+    col = c("black", "gray", "blue"), lty = c(1, 2, 1))
R> par(mfg=c(1, 1))
R> matplot(y = custom_sim$volatility_factor,
+    add = TRUE, type = "l", col = "blue", lty = 1)
```

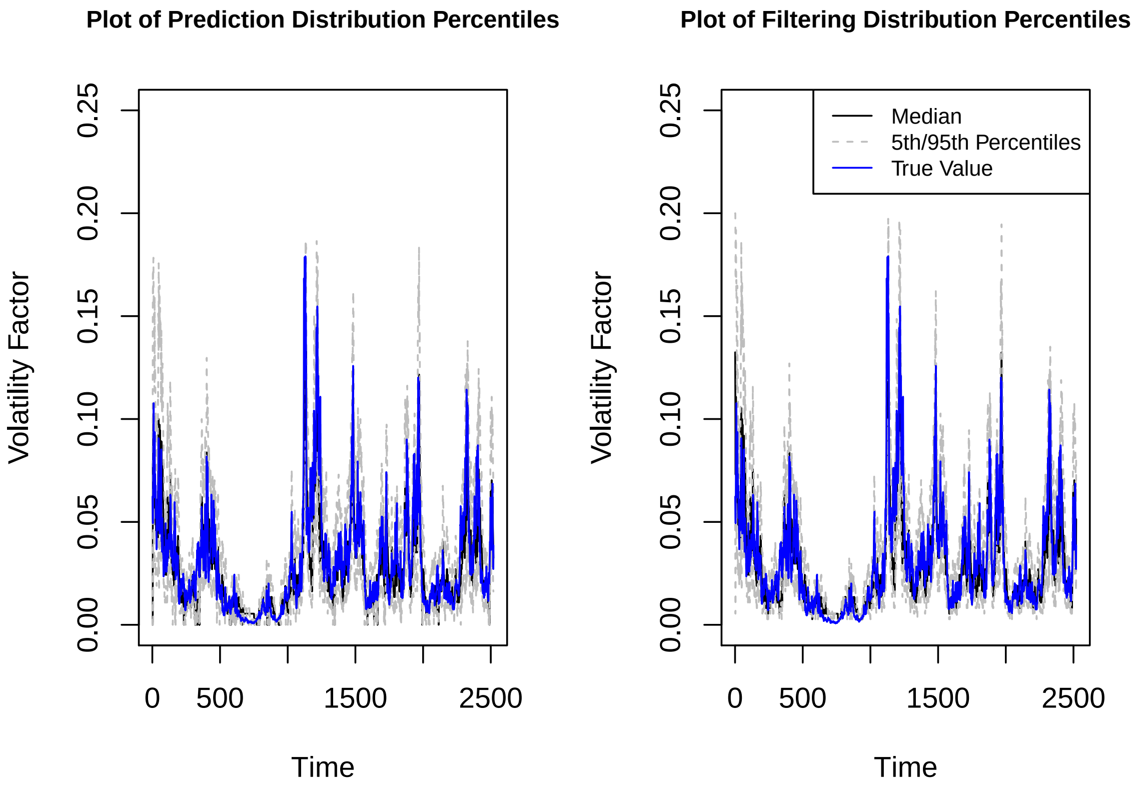

Figure 5: Simulated values of the volatility factor (in blue) along with the median estimate (in black) and the 5th and 95th percentile estimates (dotted grey lines) for the filtering and prediction distributions.

The resulting graph is shown in Figure 5. The estimate of the median of the filtering distribution (black line) is similar to the simulated volatility factor (blue line) and remains in between our estimates of the 5th and 95th percentiles (dotted grey lines) for both the filtering and prediction distributions. Generally, both distributions have similar medians, 5th, and 95th percentiles. However, the 5th and 95th percentile estimates tend to be slightly closer to one another for the filtering distribution than for the prediction distribution, which is expected as the filtering distribution contains additional information about the volatility factor.

4.3. Maximum likelihood estimation

We illustrate how the `DNFOptim` function can be used to obtain MLEs using simulated data from the SV model of Taylor with leverage (i.e., Model 2 in Section 2.1). We generate a ten-year return series with the `modelSim` function.

```
R> Taylor_mod <- dynamicsSVM(model = "TaylorWithLeverage",
+    phi = 0.9, theta = -7.36, sigma = 0.363, rho = -0.75)
R> Taylor_sim <- modelSim(t = 2520, dynamics = Taylor_mod, init_vol = -7.36)
```

When passing the initial parameter vector to the optimizer, the parameters should follow a specific order. For the `PittMalikDoucet` model (and all other nested specifications; that is, Models 1–3), the parameters should be in the following order: `phi`, `theta`, `sigma`, `rho`, `delta`, `alpha`, and `p`. For the `DuffiePanSingleton` model (and all other nested specifications; that is, Models 4–6), the parameters should be in the following order: `mu`, `alpha`, `delta`, `rho_z`, `nu`, `omega`, `kappa`, `theta`, `sigma`, and `rho`. The `CAPM_SV` model (Model 7) parameters should be passed as: `c_0`, `c_1`, `phi`, `theta`, and `sigma`. The parameters not in use in a given model should be discarded and not included to the list of parameters. For example, the `TaylorWithLeverage` model does not contain jumps; its four parameters are passed in the following order: `phi`, `theta`, `sigma`, and `rho`. Alternatively, users can pass the model dynamics to the `pars` function to find the initial parameter order.

```
R> pars(Taylor_mod)
```

```
[1] "phi"   "theta" "sigma" "rho"
```

This determines the order in which we define the elements of the initial parameter vector `init_par` below.

```
R> init_par <- c(0.8, -10, 0.5, -0.8)
```

We pass the data and initial parameters to the `DNFOptim` function that uses the optimization algorithm of Nelder and Mead (1965).

```
R> optim_test <- DNFOptim(data = Taylor_sim$returns,
+     dynamics = Taylor_mod, par = init_par, method = "Nelder-Mead",
+     hessian = TRUE)
```

The MLE can be extracted from `optim_test`, which stores the results of the `DNFOptim` method. Running the optimizer takes about 1.5 minutes on a laptop with a 4 GHz Intel Core i7-8550U processor.

```
R> summary(optim_test)

Model:
TaylorWithLeverage

Coefficients:
      Estimate Std Error   2.5 %  97.5 %
phi     0.8895   0.01341  0.8632  0.9158
theta  -7.3321   0.05387 -7.4377 -7.2265
sigma   0.3555   0.02715  0.3023  0.4087
rho    -0.7536   0.03714 -0.8264 -0.6808


Log-Likelihood:
'log Lik.' 5673.654 (df=4)
```

The 95% confidence intervals based on the MLE parameters and their standard errors cover the true parameters.

For built-in models, users need not pass initial parameters to the optimizer. If the `par` argument of the `DNFOptim` function is left to its default values, `par = NULL`, the initial parameters are found using a heuristic algorithm detailed in Appendix A. In this case, omitting the initial parameters gives virtually identical results.

```
R> optim_no_init <- DNFOptim(data = Taylor_sim$returns,
+     dynamics = Taylor_mod, method = "Nelder-Mead",
+     hessian = TRUE)

No initial parameters given.
Obtaining initial guess for starting parameters...
Initial par vector is:
0.9509383 -7.155225 0.1196848 0.3119251

>R summary(optim_no_init)

Model:
TaylorWithLeverage

Coefficients:
      Estimate Std Error   2.5 %  97.5 %
phi     0.8895   0.01342  0.8632  0.9158
theta  -7.3317   0.05389 -7.4373 -7.2261
sigma   0.3559   0.02718  0.3027  0.4092
rho    -0.7535   0.03714 -0.8263 -0.6808


Log-Likelihood:
'log Lik.' 5673.654 (df=4)
```

We can use the model dynamics estimated by the `DNFOptim` function to generate forecasts of future return and volatility factor values. The forecasts from the MLE model dynamics are obtained by generating sample paths (1,000 by default) that are simulated from `modelSim` where the initial volatility is sampled from the filtering distribution at time T (here, $T = 2,520$).[12]

```
R> par(mfrow = c(1, 2), cex.main = 0.85)
R> plot(predict(optim_test, n_ahead = 250))
R> legend("topright", cex = 0.75,
+    legend = c("Actual", "95 % Confidence Intervals", "Predicted"),
+    col = c("black", "blue", "magenta"), lty = c(1, 1, 1))
```

Figure 6 shows the resulting predictions. The left plot showcases how the expected log-volatility reverts to its long-run mean value of $\theta = -8.972$. As the forecast long-run mean log-volatility decreases, we can observe in the right plot the width of the 95% confidence around our expected future returns shrinking as the volatility falls.

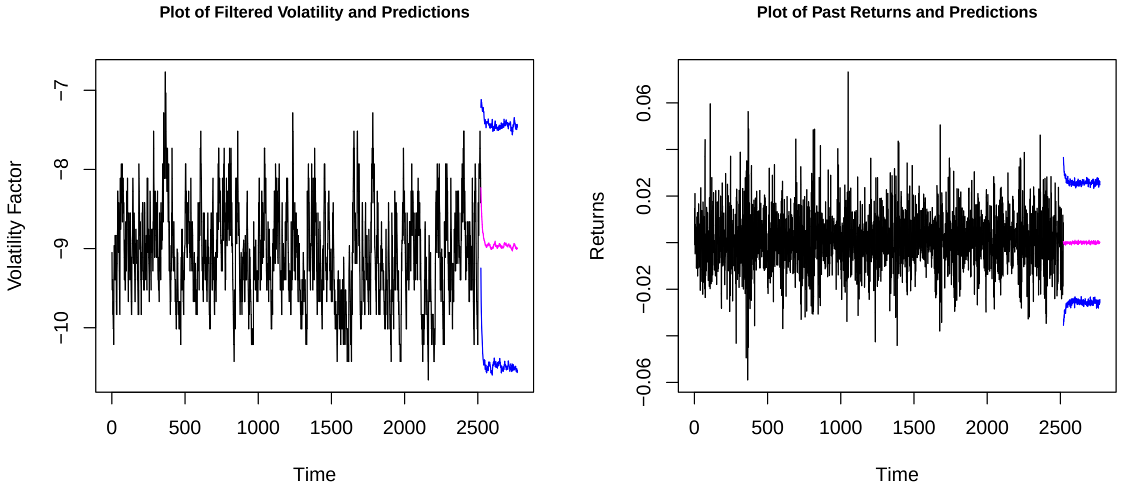

Figure 6: The filtered volatility factor values (in black on the left) and the observed returns (in black on the right) along with their expected forecasts (in magenta) with 95% confidence intervals (in blue).

We can also use the `DNFOptim` function to get MLE parameters for custom models. The order in which we pass the initial custom model parameters will depend on the function in which they appear. Some parameters appear in user-specified functions (i.e., `mu_y_params`, `sigma_y_params`, `mu_x_params`, `sigma_x_params`, and `jump_params`), while others are present in the general framework for custom models in Section 2 (i.e., `rho`, `delta`, `alpha`, `rho_z`, and `nu`). The order to pass the parameters is: `mu_y_params`, `sigma_y_params`, `mu_x_params`,

[12] Specifically, sample paths based on the estimated parameters are generated using Monte Carlo simulation (via the `modelSim` function). The initial volatility for each of these paths is sampled from the filtering distribution of the volatility factor at time T, $p(x_T \mid y_1 : T)$. The `predict` function then computes the mean of the the simulated paths and constructs confidence intervals based on their standard deviation at each time step.

`sigma_x_params`, `rho`, `delta`, `alpha`, `rho_z`, `nu`, and `jump_params`. If an argument is repeated (e.g., both `mu_y_params` and `sigma_y_params` use the same parameter), we write it only when it first appears in the custom model definition.

In this case, we apply the `DNFOptim` function to returns generated from the ONEN model built from custom model dynamics; that is, the model of Section 4.1 without jumps (i.e., $\omega = \alpha = \delta = \rho_z = \nu = 0$). Thus, `mu_y` is a function of `mu`, `sigma_y` is not a function of any model parameters, `mu_x` is a function of `kappa` and `theta`, and `sigma_x` is a function of `sigma`. Following the parameter order for custom models given above, the initial parameters must be passed as follows: `mu`, `kappa`, `theta`, `sigma`, and `rho`. The parameter order for custom models can also be found using the `pars` function.

```
R> N <- 50
R> var_mid_points <- seq(from = 0.000001, to = 0.2, length = N)
R> j_nums <- 0
R> jump_mid_points <- 0
R> grids <- list(var_mid_points = var_mid_points,
+     j_nums = j_nums, jump_mid_points = jump_mid_points)

R> init_par <- c(0.03, 120, 0.04, 3.5, -0.75)
R> lower <- c(-0.2, 0, 0, 0, -1)
R> upper <- c(0.2, Inf, 0.15, 10, 0)

R> dnf_custom_opt <- DNFOptim(data = custom$returns, grids = grids,
+     dynamics = custom_mod, lower = lower,
+     upper = upper, par = init_par, method = "L-BFGS-B",
+     rho = "var", hessian = TRUE)
```

For built-in models, `DNFOptim` constrains the parameter search to the ranges given in the support column of Tables 1, 2, and 3. For custom models, only certain model parameters are constrained to their support (i.e., `rho`, `delta`, `alpha`, `rho_z`, and `nu`). In order to place limits on the search for the other parameters in the ONEN model, we set `method = "L-BFGS-B` and pass `upper` and `lower` bounds to the `DNFOptim` function.[13] The optimization takes roughly 10 minutes.[14] After running `DNFOptim`, we now print out the estimated parameters and get the standard errors:

```
R> summary(dnf_custom_opt)

Model:
Custom

Coefficients:
       Estimate Std Error    2.5 %    97.5 %
mu     -0.02025  0.027772 -0.07468   0.03418
kappa 120.03658 25.964937 69.14624 170.92693
```

[13] The L-BGFS-B optimization algorithm is the only optimizer in the `optim` function to support parameter constraints.

[14] This timing is also obtained from a laptop equipped with a 4 GHz Intel Core i7-8550U processor.

```
theta   0.06091  0.007494  0.04622   0.07559
sigma   2.62066  0.150532  2.32562   2.91569
rho    -0.60071  0.024960 -0.64963  -0.55179


Log-Likelihood:
'log Lik.' 16829.34 (df=5)
```

We find that most of the MLE parameters are fairly close to those used to generate the data, except for σ and ρ, which are outside the 95% confidence interval obtained from their standard errors. Overall, the estimated parameters are reasonable for a single simulated path.

The filtering and predictions distribution along with the true volatility are displayed in Figure **??**. Once again, the prediction and filtering distributions generally cover the true volatility values from the simulated data.

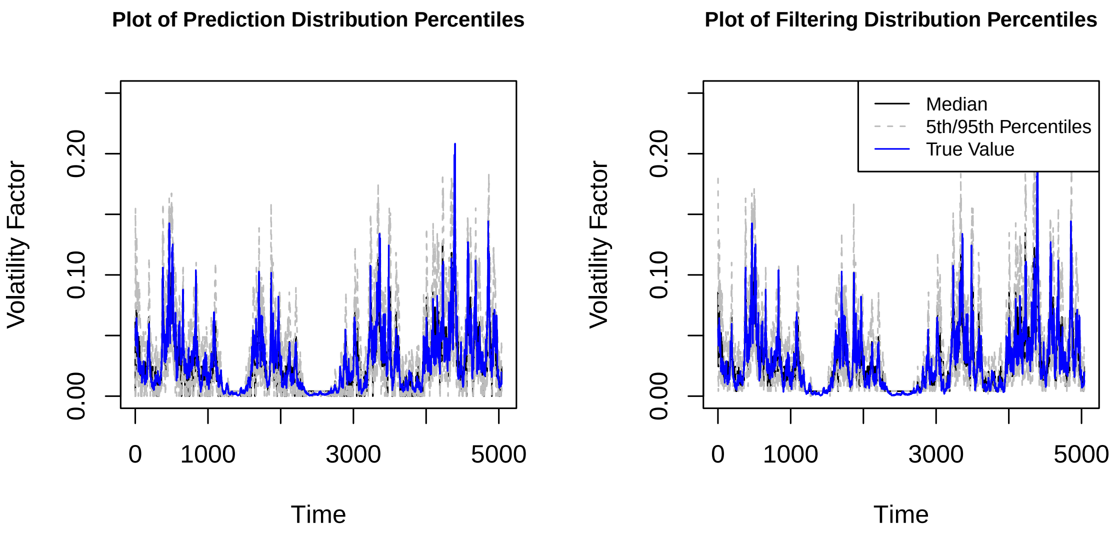

Figure 7: Simulated values of the volatility factor (in blue) along with the median estimate (in black) and the 5$^{\text{th}}$ and 95$^{\text{th}}$ percentile estimates (dotted grey lines) for the filtering and prediction distributions of simulated data from the ONEN model of Christoffersen, Jacobs, and Mimouni (2010)

5. MLE simulation study

In this section, we study the parameter bias that results from using the DNF in maximum likelihood estimation. We perform a simulation study that extends previous work on the DNF.

In their study, Bégin and Boudreault (2021) assessed the bias in the DNF likelihood evaluations for jump-diffusion models. Using likelihood evaluations from a particle filter with a large number of particles as a proxy for the true likelihood, they found that the DNF likelihood evaluations have a mean absolute percentage error (MAPE) that is less than 0.1% with 50–60 volatility nodes.

Through simulation studies, both Watanabe (1999) and Langrock, MacDonald, and Zucchini (2012) investigated the parameter bias that results from using the DNF to estimate discrete-time SV models without jumps. Both studies found that the DNF has comparable or lower errors than alternative methods, and Langrock, MacDonald, and Zucchini (2012) found that the DNF achieved these results with lower or comparable computing times when using between 50 and 100 volatility nodes.

The present simulation study has three objectives. First, we want to verify whether the slight biases in the likelihood evaluations of jump-diffusion models noted in Bégin and Boudreault (2021) translate into accurate parameter MLEs. Second, we determine whether the promising results of Watanabe (1999) and Langrock, MacDonald, and Zucchini (2012) extend to discrete-time models that allow for the leverage effect and return jumps. Finally, the study provides users with an overview of the finite-sample performance they can anticipate when using the `DNFOptim` function.

For this exercise, we generate 100 random daily return series of 20 years for each built-in model using the parameters in the "True value" columns of Table 4 for standard discrete-time SV models and Table 5 for SV jump-diffusion models. We then use the `DNFOptim` function to obtain the MLE of each series with the limited-memory Broyden–Fletcher–Goldfarb–Shanno with box constraints (L–BFGS–B) optimization algorithm of Byrd, Lu, Nocedal, and Zhu (1995); that is, using the `method = "L-BGFS-B"` argument in the `DNFOptim` function.[15] The other columns of Tables 4 and 5 give the average and standard deviations (SDs) of the MLE from the 100 simulated return series. All estimations are performed with the default grid lengths (i.e., `N = 50`, `K = 20`, and `R = 1`).

	True value	**Taylor**	**Taylor with leverage**	**Pitt, Malik, and Doucet**
ϕ	0.900	0.898 (0.013)	0.899 (0.007)	0.900 (0.008)
θ	−7.360	−7.354 (0.052)	−7.366 (0.034)	−7.375 (0.038)
σ	0.363	0.362 (0.027)	0.359 (0.016)	0.358 (0.017)
ρ	−0.745		−0.745 (0.024)	−0.749 (0.029)
δ	0.026			0.022 (0.011)
α	−0.050			−0.053 (0.023)
p	0.010			0.013 (0.013)

Table 4: Average maximum likelihood estimates and standard deviations for standard discrete-time stochastic volatility models.

[15] We change the optimization algorithm to show that it is possible to do so. Results obtained with the Nelder–Mead method are virtually the same as those obtained with the L–BFGS–B method.

Table 4 displays the average MLEs and their SDs for the SV model of Taylor (1986), the SV model with leverage, and the SV model with leverage and return jumps of Pitt, Malik, and Doucet (2014). These correspond to Models 1, 2, and 3 of Section 2.1, respectively.

Throughout the models, the MLE for the parameters that are not related to jumps (i.e., ϕ, θ, σ, and ρ) are close to their true values, on average, for the standard discrete-time SV models. The accuracy of the estimates for the model of Taylor (1986) are congruent with the results of Watanabe (1999) and Langrock, MacDonald, and Zucchini (2012). The jump parameters (i.e., p, δ, and α) tend to be more difficult to estimate, displaying fairly large SDs.

	True value	**Heston**	**Bates**	**Duffie, Pan, and Singleton**
μ	0.038	0.048 (0.027)	0.044 (0.036)	0.021 (0.037)
κ	3.689	4.810 (0.824)	4.811 (0.525)	4.352 (0.379)
θ	0.032	0.027 (0.004)	0.029 (0.003)	0.036 (0.006)
σ	0.446	0.419 (0.024)	0.431 (0.027)	0.452 (0.033)
ρ	−0.745	−0.728 (0.030)	−0.737 (0.037)	−0.731 (0.040)
ω	5.125		4.165 (0.386)	4.665 (0.339)
δ	0.026		0.025 (0.004)	0.026 (0.006)
α	−0.050		−0.055 (0.005)	−0.051 (0.006)
ρ_z	−1.000			−1.008 (0.138)
ν	0.020			0.022 (0.004)

Table 5: Average maximum likelihood estimates and standard deviations for stochastic volatility jump-diffusion models.

Table 5 reports the average MLE parameters and their SDs for the jump-diffusion SV models of Heston (1993), Bates (1996), and Duffie, Pan, and Singleton (2000), corresponding to Models 4, 5, and 6 in Section 2.2, respectively. For the models, we find that the parameters that are not associated with jumps (i.e., μ, κ, θ, σ, and ρ) are well estimated. As for the jump parameters, we find relatively small SDs when using jump-diffusion models. For both return and volatility jumps, the MLE slightly overestimates the jump sizes, and underestimates their frequency, on average. However, the only parameters that are significantly misestimated are the ω and κ parameters in the model of Bates (1996).

6. Empirical applications

We can also use the `DNFOptim` function to obtain model MLE parameters from real data. In this section, we use the **quantmod** package of Ryan and Ulrich (2022) to load return data from individual equities, the S&P 500 index, and obtain MLE parameters for both built–in and custom SV models. The data are from January 2007 to November 2022, accounting for roughly 16 years of daily observations. This period includes the 2008 Global Financial Crisis and the COVID-19 pandemic.

6.1. Parameter estimation for the Standard and Poor's 500 index

We start by extracting the S&P 500 returns.

```
R> library("quantmod")
R> library("xts")
R> getSymbols("^GSPC", src = "yahoo")
R> SP500 <- periodReturn(GSPC, period = "daily", subset = "2007/20221107")
R> date <- index(SP500)
R> plot(x = date, y = SP500, main = "", type = "l",
+    ylab = "Returns", xlab = "Time")
```

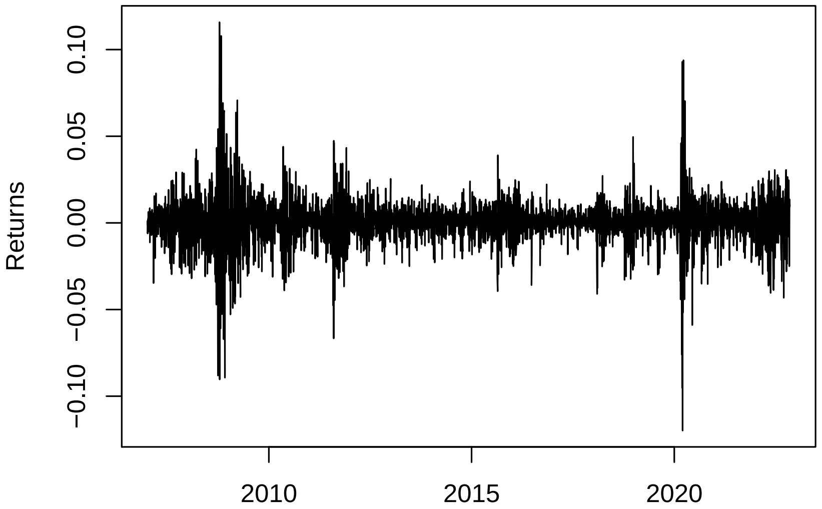

Figure 8: Daily S&P 500 index returns from January 2007 to November 2022.

The above code generates the plot of our return data in Figure 8.

We give the code for this workflow with market data using the Heston model. The `DNF` and `DNFOptim` functions support extensible time series objects from the **xts** package of Ryan and Ulrich (2024). Thus, the `SP500` object can be passed directly to the **SVDNF** package's DNF functions. In this exercise, we use the optimization algorithm of Nelder and Mead (1965).[16]

```
R> init_par <- c(0.038, 3.689, 0.032, 0.446, -0.745)
R> Heston_mod <- dynamicsSVM(model = "Heston")
R> optim_Heston <- DNFOptim(data = SP500,
+    dynamics = Heston_mod, par = init_par,  method = "Nelder-Mead", tol = 1,
+    hessian = TRUE)
```

[16] Similar results can be obtained using the L-BFGS-B optimization algorithm.

The `tol` argument of the `DNF` function allows users to rerun the optimizer using the MLEs from the previous iteration as starting values until the likelihood does not improve by at least the value `tol`. All the parameter MLEs obtained in this section are computed with `tol = 1` in the `DNFOptim` function.

```
R> summary(optim_Heston)

Model:
Heston

Coefficients:
      Estimate Std Error      2.5 %   97.5 %
mu     0.04313  0.025114 -0.006094  0.09235
kappa  4.31819  0.321101  3.688846  4.94754
theta  0.04919  0.003013  0.043289  0.05510
sigma  0.57642  0.026777  0.523934  0.62890
rho   -0.77202  0.028515 -0.827910 -0.71614


Log-Likelihood:
'log Lik.' 13042.06 (df=5)
```

Tables 6, 7, and 8 give the MLE parameter estimates, $\hat{\Theta}$, and their standard errors (SEs) obtained for the built-in standard discrete-time (Models 1–3), the built-in jump-diffusion models (Models 4–6), and custom CEV versions of the jump-diffusion models, respectively. The custom CEV models are similar to Models 4, 5, and 6, but with the volatility factor diffusion function specified as in Equation (1).

Tables 6, 7, and 8 also provide the log-likelihood at the MLE, $\log \mathcal{L}(\hat{\Theta} \mid y_{1:T})$, and the Bayesian information criterion (BIC) of Schwarz (1978) for each model. The BIC provides a criterion for us to compare SV models with different numbers of parameters and from different model families (i.e., discrete-time versus continuous-time). It is defined as

$$\text{BIC} = -2 \log \mathcal{L}(\hat{\Theta} \mid y_{1:T}) + k \ln(T),$$

where k is the number of parameters in a given model, and is used to adjust the log-likelihood value by penalizing for model complexity (i.e., the number of parameters). The BIC can be easily computed using the `BIC` function.

```
 BIC(logLik(optim_Heston))

[1] -26042.65
```

For the standard discrete-time models, the MLE parameters are reported in Table 6. We find that the log-likelihood increases both when incorporating the leverage effect in the standard SV model and adding return jumps. Moreover, the more complicated models outperform the simpler ones in terms of BIC despite the penalty for additional model parameters. These results are similar to those in Pitt, Malik, and Doucet (2014), where they find that the

	Taylor	**Taylor with leverage**	**Pitt, Malik, and Doucet**
ϕ	0.975	0.959	0.964
	(0.005)	(0.004)	(0.004)
θ	−9.470	−9.245	−9.097
	(0.163)	(0.088)	(0.096)
σ	0.248	0.308	0.298
	(0.019)	(0.018)	(0.017)
ρ		−0.740	−0.838
		(0.029)	(0.028)
p			0.039
			(0.018)
δ			0.011
			(0.001)
α			−0.007
			(0.004)
Log-likelihood	12,928.99	13,050.98	13,072.40
BIC	−25,833.10	−26,068.78	−26,086.76

Table 6: Maximum likelihood estimates for the S&P 500 index with standard discrete-time stochastic volatility models.

	Heston	**Bates**	**Duffie, Pan, and Singleton**
μ	0.043	0.032	−0.001
	(0.025)	(0.028)	(0.027)
κ	4.318	4.288	4.427
	(0.321)	(0.350)	(0.352)
θ	0.049	0.046	0.041
	(0.003)	(0.003)	(0.001)
σ	0.576	0.557	0.526
	(0.027)	(0.027)	(0.024)
ρ	−0.772	−0.793	−0.813
	(0.029)	(0.027)	(0.027)
ω		3.816	2.824
		(1.218)	(0.810)
δ		0.006	0.003
		(0.004)	(0.003)
α		−0.016	−0.021
		(0.003)	(0.003)
ρ_z			−0.136
			(0.098)
ν			0.024
			(0.007)
Log-likelihood	13,042.06	13,053.92	13,065.71
BIC	−26,042.65	−26,041.52	−26,048.50

Table 7: Maximum likelihood estimates for the S&P 500 index with stochastic volatility jump-diffusion models.

	Heston-CEV	**Bates-CEV**	**Duffie, Pan, and Singleton-CEV**
μ	0.099 (0.024)	0.066 (0.024)	−0.029 (0.030)
κ	4.849 (0.084)	4.343 (0.067)	4.508 (0.129)
θ	0.040 (0.002)	0.041 (0.001)	0.037 (0.001)
σ	2.039 (0.325)	1.882 (0.249)	1.822 (0.251)
ρ	−0.768 (0.028)	−0.803 (0.026)	−0.834 (0.025)
ω		4.281 (2.329)	5.879 (1.860)
δ		0.005 (0.004)	0.003 (0.005)
α		−0.015 (0.005)	−0.016 (0.006)
ρ_z			−0.301 (0.318)
ν			0.014 (0.005)
β	0.780 (0.035)	0.767 (0.028)	0.766 (0.027)
Log-likelihood	13,085.40	13,099.36	13,115.15
BIC	–26,121.04	–26,124.09	–26,139.10

Table 8: Maximum likelihood estimates for the S&P 500 index with nonaffine jump-diffusion models.

standard discrete-time stochastic volatility model with leverage and jumps often performs better when compared to other models studied for different periods of S&P 500 return data.

In Table 7, we observe a similar pattern. As with the discrete-time models, the more complex continuous-time models better capture the dynamics of the S&P 500 return data in terms of BIC. For the models with jumps, we find around two to four jumps annually and negative return jumps, on average. The return jumps are less frequent than those for the model of Pitt, Malik, and Doucet (2014), but larger in magnitude for both the model of Bates (1996) and Duffie, Pan, and Singleton (2000). The jump parameter estimates resemble those found in the literature (see, e.g., Hurn, Lindsay, and McClelland 2015; Jacobs and Liu 2018).

For the model of Duffie, Pan, and Singleton (2000), we estimate $\hat{\mu} = -0.001$. Although not statistically significant, this parameter is not usually negative as this would indicate a negative return drift. However, the constants in return drift function remain positive in this case due to the jump compensator. Indeed, the annualized constant terms in the return drift are estimated to be

$$\hat{\mu} - \hat{\omega}\,\hat{\alpha} = -0.001 - (2.824 \times -0.021) = 0.058,$$

which is similar to the drift estimated for the model of Heston (1993). This also explains why we get $\hat{\mu} = -0.029$ in Table 8 for the CEV version of the model of Duffie, Pan, and Singleton (2000).

In Table 8, we find, once again, that adding return and volatility jumps improves model performance in terms of BIC. For the nonaffine jump-diffusion models, the estimated CEV parameter β is around 0.75, which is in the same range as values found in the literature (see, e.g., Aït-Sahalia and Kimmel 2007; Kaeck and Alexander 2012).

Across the model families considered, we find that BIC improves when adding jumps. This confirms previous results on the importance of jumps to capture market behaviour during times of financial turmoil in the literature (see, e.g., Bates 1996; Eraker, Johannes, and Polson 2003; Eraker 2004). Additionally, we find that the BIC favours nonaffine jump-diffusion models over the discrete-time models, which are favored over the affine jump-diffusion models.

We can easily visualize the difference between affine and nonaffine volatility dynamics using the `extractVolPerc` function. We extract the median filtering distribution from the non-affine version of the model of Duffie, Pan, and Singleton (2000) and the model of Heston (1993) that were fit with the S&P 500 data for comparison.

```
R> DPS_vol <- extractVolPerc(optim_DPS_CEV)[-1]

R> Hest_vol <- extractVolPerc(optim_Heston_SP500)[-1]

R> plot(x = index(SP500), y = (DPS_vol - Hest_vol), type ='l',
+    ylab = "Difference in volatility", xlab = 'Date')
```

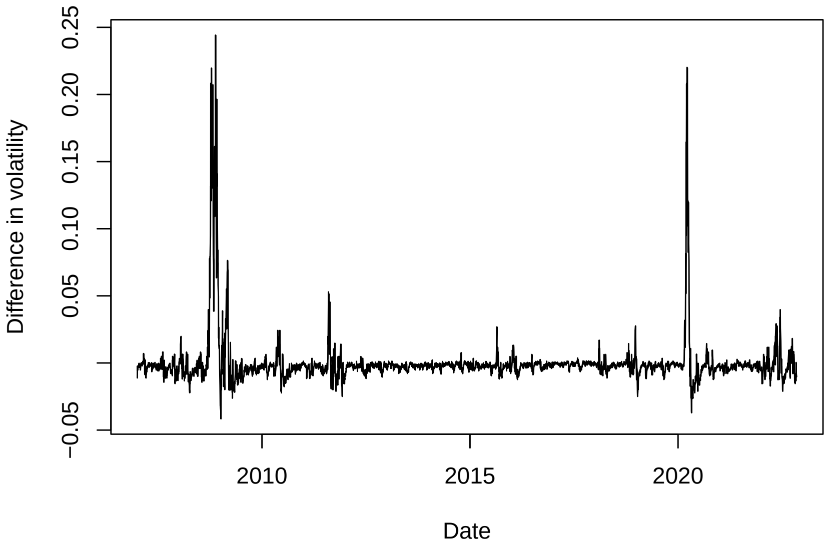

Figure 9: Plot of the difference between the filtering distribution estimates for S&P 500 daily returns from the model of Heston (1993) and a non-affine version of the model of Duffie, Pan, and Singleton (2000).

The plot of the difference of the filtering distribution medians appears in Figure 9 and shows that the volatilities for both models are usually near one another except during the 2008 financial crisis and the 2020 stock market crash caused by the Covid-19 pandemic. The `extractVolPerc` function for other fitted models and quickly compare the impact of selecting a different SV model on the filtering distribution's median volatility estimate.

In Figure 10, we plot the two filtering distribution medians and add the average volatility computed via Monte Carlo simulation from the `predict` function. The graph shows that despite having a lower long-run volatility value θ (0.037 versus 0.049), the nonaffine model with jumps still yields a slightly higher average simulated volatility. This showcases the ability of the nonaffine model with jumps to keep a lower long-run value and yet to spike up and capture extreme volatility values.

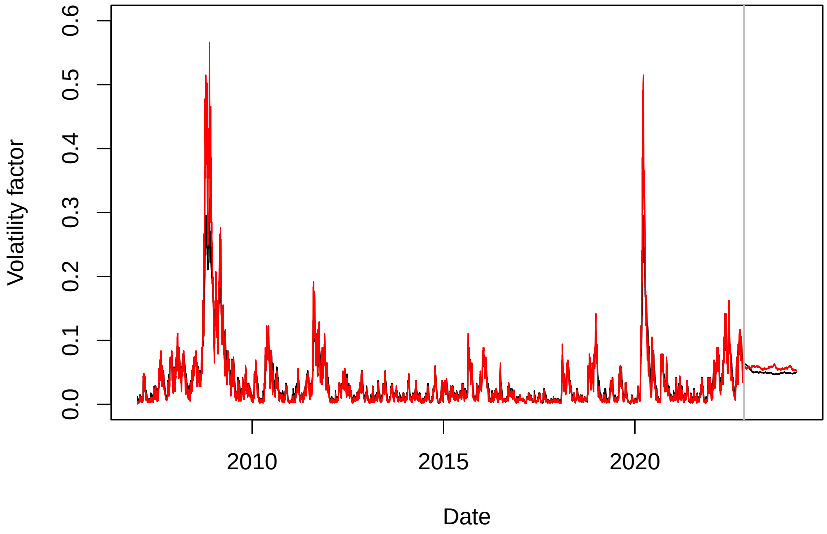

Figure 10: Plot of the filtering distribution estimates for S&P 500 daily returns from the model of Heston (1993) (in black) and a non-affine version of the model of Duffie, Pan, and Singleton (2000) (in red) with average forecast values (after grey line).

6.2. CAPM-SV parameter estimates for individual equities

We now estimate the parameters of the CAPM-SV (i.e., Model 7) for the three largest publicly traded equities. At the time of writing, these are: Microsoft, Apple Inc., and Nvidia. The returns data from the S&P 500 will serve as a proxy for overall market returns and the 3-month constant maturity U.S. Treasuries will serve as the risk-free rate (i.e., they will be R_t^m and R_t^f, respectively).

We first obtain the risk-free rate proxy, extracting its values at the dates when the market is open, and filling NAs values using the previous observation carried forward.

```
R> getSymbols('DGS3MO', src = "FRED", subset = "2007/20221107")
R> r_f = DGS3MO["2007/20221107"] / (100 * 252)
```

```
R> r_f <- r_f[index(SP500)]
R> r_f[is.na(r_f)] <- r_f[which(is.na(r_f)) - 1]
```

The excess returns from investing in the market and in Microsoft are computed below.

```
getSymbols("MSFT", src = "yahoo")
MSFT <- periodReturn(MSFT, period = "daily", subset = "2007/20221107")

r_excess <- as.vector(SP500 - r_f)
MSFT_excess <- MSFT - r_f
```

We then create a models dynamics object for the CAPM-SV, select initial parameters for the optimizer.

```
R> CAPM_dyn <- dynamicsSVM(model = "CAPM_SV")
R> init_par = c(0.1, 1, 0.98, -8, 0.3)
```

We now specify the design matrix in the $\mathbf{F}_t c^\top$ term in Model 7. We set the first column of this matrix to 1 to multiply the intercept term c_0 and the second column to be the excess market returns, which are multiplied by c_1.

```
R> factor_mat <- cbind(rep(1, times = length(r_excess)), r_excess))
R> MSFT_opt = DNFOptim(data = MSFT_excess, dynamics = CAPM_dyn, tol = 1,
                       factors = factor_mat, par = init_par, hessian = TRUE)
```

	MSFT	**AAPL**	**NVDA**
$c_0 \times 100$	0.015	0.051	0.016
	(0.013)	(0.017)	(0.027)
c_1	1.062	1.085	1.486
	(0.014)	(0.017)	(0.027)
ϕ	0.801	0.867	0.800
	(0.025)	(0.019)	(0.028)
θ	–9.498	–8.933	–8.053
	(0.057)	(0.066)	(0.056)
σ	0.626	0.504	0.611
	(0.041)	(0.039)	(0.045)
Log-likelihood	12,730.42	11,672.36	9,862.16
BIC	–25,419.39	–23,303.26	–19,682.86

Table 9: Maximum likelihood estimates of the CAPM-SV model parameters for the three largest equities in the S&P 500 by market capitalization.

Table 9 displays the results of repeating this exercise for Apple Inc. and Nvidia. We find that Nvidia has the highest long-run log-volatility θ of the three stocks, while Microsoft has the lowest. Apple Inc. boasts the highest average excess market returns parameter c_0 with similar systemic risk c_1 to Microsoft, both of which are lower than Nvidia's.

As models with jumps better fit the S&P 500 index data, we create a custom model that follows the dynamics of Model 7, the CAPM-SV model, but with Gaussian return jumps that arrive following a Bernoulli distribution. This is the capital asset pricing model with yield jumps or CAPM-SVYJ. We define its drift and diffusion functions below.

```
R> phi <- 0.9; theta <- -9; sigma <- 0.3
R> alpha <- -0.05; p <- 0.025; delta <- 0.025
R> coefs <- c(0, 1)
R> mu_y_capm <- function(x, dummy) {
+    return(0)
+  }
R> mu_y_params <- list()

R> sigma_y_capm <- function(x, dummy) {
+    return(exp(x / 2))
+  }
R> sigma_y_params <- list(0)

R> mu_x_capm <- function(x, phi, theta) {
+    return(theta + phi * (x - theta))
+  }
R> mu_x_params <- list(phi, theta)
R> sigma_x_capm <- function(x, sigma) {
+    return(sigma)
+  }
R> sigma_x_params <- list(sigma)
```

Next, we set the return jump distribution.

```
R> jump_density <- function(x, p) {
+    return(dbinom(x, 1, p))
+  }
R> jump_dist <- rbinom
R> jump_params <- p
```

Finally, we can create the CAPM-SVYJ model by passing the drift functions, diffusion functions, jump distribution, and parameters to the `dynamicsSVM` function.

```
R> custom_CAPMSVM <- dynamicsSVM(
+    model = "Custom", mu_y = mu_y_capm, sigma_y = sigma_y_capm,
+    rho = 0, rho_z = 0, nu = 0, alpha = alpha, delta = delta, coefs = coefs,
+    mu_x = mu_x_capm, sigma_x = sigma_x_capm, jump_density = jump_density,
+    jump_dist = jump_dist, jump_params = jump_params,
+    mu_y_params = mu_y_params, sigma_y_params = sigma_y_params,
+    mu_x_params = mu_x_params, sigma_x_params = sigma_x_params)
```

As we are working with a custom model, we must construct a quadrature point grid. To this end, we follow the procedure for discrete models in Section 3.

```
R> mean <- theta
R> sd <- sqrt((sigma^2) / (1 - phi^2))
R> N <- 50
```

```
R> var_mid_points <- seq(from = 0, to = sqrt((3 + log(N)) * sd),
+    length = floor(N / 2 + 1))^2

R> var_mid_points <- (sort(c(-var_mid_points[2:N],
+    var_mid_points)) + mean)[1:N]
R> j_nums <- c(0, 1)
R> jump_mid_points <- 0
R> cap_grid <- list(var_mid_points = var_mid_points, j_nums = j_nums,
+    jump_mid_points = jump_mid_points)
```

After setting an initial parameter vector, we run the `DNFOptim` function and print the parameter MLEs and model BIC.

```
R> init_par <- c(0, 1, 0.9, -9, 0.3, 0.025, -0.05, 0.025)

R> MSFT_SVYJ <- DNFOptim(data = MSFT_excess, dynamics = custom_CAPMSVM,
+    grid = cap_grid, tol = 1, factors = factor_mat, par = init_par,
+    hessian = TRUE, jump_params_list = "p", alpha = "var", delta = "var")

R> summary(MSFT_SVYJ)

Model:
Custom

Coefficients:
        Estimate Std Error      2.5 %     97.5 %
c_0:   7.583e-05 0.0001303 -0.0001796  0.0003312
c_1:   1.094e+00 0.0144321  1.0653755  1.1219481
phi    8.673e-01 0.0513138  0.7667364  0.9678828
theta -9.542e+00 0.0607762 -9.6613862 -9.4231479
sigma  4.433e-01 0.1039113  0.2396749  0.6469998
delta  4.605e-02 0.0083327  0.0297134  0.0623768
alpha  4.031e-03 0.0072857 -0.0102484  0.0183108
p      1.365e-02 0.0062032  0.0014872  0.0258034


Log-Likelihood:
'log Lik.' 12752.77 (df=8)

R> BIC(logLik(MSFT_SVYJ))

[1] -25439.2
```

The parameter MLEs and standard errors for Microsoft, Apple Inc., and Nvidia for the CAPM-SVYJ model are given in Table 10. Unlike our estimate for the S&P500, we find positive average jumps sizes α. While the individual equities jump size have similar magnitudes on average to the those of the S&P 500, they have larger standard deviations δ.

The estimates of the volatility of the volatility σ are smaller while the persistence parameters ϕ are higher after adding returns jumps. We find similar values for the estimates of the long-run log-volatility parameter θ and the slope parameters c_1 across both the CAPM-SV and CAPM-SVYJ models. For each of the three equities, the addition of a return jump component to the model improves its BIC value.

	MSFT	**AAPL**	**NVDA**
$c_0 \times 100$	0.008	0.047	0.025
	(0.013)	(0.018)	(0.027)
c_1	1.094	1.078	1.478
	(0.014)	(0.018)	(0.028)
ϕ	0.867	0.923	0.987
	(0.014)	(0.031)	(0.004)
θ	−9.542	−8.998	−8.090
	(0.061)	(0.078)	(0.149)
σ	0.443	0.338	0.104
	(0.104)	(0.080)	(0.019)
δ	0.046	0.033	0.067
	(0.008)	(0.005)	(0.006)
α	0.004	0.014	0.012
	(0.007)	(0.007)	(0.006)
p	0.014	0.072	0.041
	(0.006)	(0.009)	(0.007)
Log-likelihood	12,752.77	11,699.12	9,939.04
BIC	−25,439.20	−23,33‘.90	−19,811.74

Table 10: Maximum likelihood estimates of the CAPM-SVYJ model parameters for the three largest equities in the S&P 500 by market capitalization.

7. Summary and discussion

This article introduced the **SVDNF** R package. Namely, we presented the discrete nonlinear filtering algorithm and a flexible stochastic volatility modelling framework with jumps that is made available to package users via customized model dynamics. For this general framework, the package allows users to simulate data, get prediction and filtering distribution estimates, and obtain maximum likelihood parameter estimates. Given the importance of jumps (and stochastic volatility), this R package fills an important gap in the literature by enabling users in finance, financial econometrics, and risk management to integrate jumps while studying a wide range of stochastic volatility model dynamics.

Furthermore, we gave examples of each of the functions. The DNF algorithm was applied to both simulated data from built-in and customized models and market data from the S&P 500 index and individual equity daily return data obtained using the **quantmod** package. We found that including jumps improves the models' ability to capture the the financial assets' behaviour in terms of BIC. This further supports existing evidence that the inclusion of jumps is important to capture asset return behaviour during periods of financial turmoil. Moreover, we found that nonaffine volatility dynamics fit S&P 500 market index data better and that the nonaffine jump-diffusion models outperform the discrete-time models studied. In turn, these discrete-time models improve BIC values over the affine jump-diffusion models as long as they allow for the leverage effect.

Computational details

The results in this paper were obtained using R 4.3.1 with the **SVDNF** 0.1.10, **Rcpp** 1.0.11, **xts** 0.13.1, and **quantmod** 0.4.24 packages. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/`.

Acknowledgments

The authors gratefully acknowledge the research assistance of Jonathan Grégoire in the early stages of this research project. This research was made possible, in part, by support provided by the BC DRI Group and the Digital Research Alliance of Canada (alliancecan.ca). Arsenault-Mahjoubi is grateful to Simon Fraser University for financial support. Bégin wishes to acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC), and Simon Fraser University. Boudreault would like to thank the financial support of NSERC.

References

Aït-Sahalia Y, Kimmel R (2007). "Maximum Likelihood Estimation of Stochastic Volatility Models." *Journal of Financial Economics*, **83**(2), 413–452. `doi:10.1016/j.jfineco.2005.10.006`.

Amaya D, Bégin JF, Gauthier G (2022). "The Informational Content of High-Frequency Option Prices." *Management Science*, **68**(3), 2166–2201. `doi:10.1287/mnsc.2020.3949`.

Andersen TG, Fusari N, Todorov V (2015). "The Risk Premia Embedded in Index Options." *Journal of Financial Economics*, **117**(3), 558–584. `doi:10.1016/j.jfineco.2015.06.005`.

Andrieu C, Doucet A, Holenstein R (2010). "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society B*, **72**(3), 269–342. `doi:10.1111/j.1467-9868.2009.00736.x`.

Bardgett C, Gourier E, Leippold M (2019). "Inferring Volatility Dynamics and Risk Premia from the S&P 500 and VIX Markets." *Journal of Financial Economics*, **131**(3), 593–618. `doi:10.1016/j.jfineco.2018.09.008`.

Bates DS (1996). "Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options." *Review of Financial Studies*, **9**(1), 69–107. `doi:10.1093/rfs/9.1.69`.

Bégin JF, Amaya D, Gauthier G, Malette ME (2020). "On the Estimation of Jump-Diffusion Models Using Intraday Data: A Filtering-Based Approach." *SIAM Journal on Financial Mathematics*, **11**(4), 1168–1208. `doi:10.1137/19M1266915`.

Bégin JF, Boudreault M (2021). "Likelihood Evaluation of Jump-Diffusion Models Using Deterministic Nonlinear Filters." *Journal of Computational and Graphical Statistics*, **30**(2), 452–466. doi:10.1080/10618600.2020.1840995.

Bollerslev T (1986). "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*, **31**(3), 307–327. doi:10.1016/0304-4076(86)90063-1.

Bos CS (2012). "Relating Stochastic Volatility Estimation Methods." In *Handbook of Volatility Models and Their Applications*, pp. 147–174. John Wiley & Sons, New York, NY, United States of America. doi:10.1002/9781118272039.ch6.

Brown TR (2020). "A Short Introduction to **PF**: A C++ Library for Particle Filtering." *Journal of Open Source Software*, **5**(54), 2599. doi:10.21105/joss.02599.

Byrd RH, Lu P, Nocedal J, Zhu C (1995). "A Limited Memory Algorithm for Bound Constrained Optimization." *SIAM Journal on Scientific Computing*, **16**(5), 1190–1208. doi:10.1137/0916069.

Chacko G, Viceira LM (2003). "Spectral GMM Estimation of Continuous-Time Processes." *Journal of Econometrics*, **116**(1-2), 259–292. doi:10.1016/S0304-4076(03)00109-X.

Chen L, Lee C, Budhiraja A, Mehra RK (2007). "**PFLib**: An Object-Oriented MATLAB Toolbox for Particle Filtering." In *Signal Processing, Sensor Fusion, and Target Recognition XVI*, volume 6567, pp. 335–342. SPIE. doi:10.1117/12.719951.

Christoffersen P, Jacobs K, Mimouni K (2010). "Volatility Dynamics for the S&P500: Evidence from Realized Volatility, Daily Returns, and Option Prices." *Review of Financial Studies*, **23**(8), 3141–3189. doi:10.1093/rfs/hhq032.

Clements A, Hurn S, White S (2006). "Estimating Stochastic Volatility Models Using a Discrete Non-linear Filter." *Working Paper*.

Cont R (2001). "Empirical Properties of Asset Returns: Stylized Facts and Statistical Issues." *Quantitative Finance*, **1**(2), 223–236. doi:10.1080/713665670.

Creal D (2012). "A Survey of Sequential Monte Carlo Methods for Economics and Finance." *Econometric Reviews*, **31**(3), 245–296. doi:10.1080/07474938.2011.607333.

de Zea Bermudez P, Marín JM, Rue H, Veiga H (2021). "Integrated Nested Laplace Approximations for Threshold Stochastic Volatility Models." *Econometrics and Statistics*, **Forthcoming**. doi:10.1016/j.ecosta.2021.08.006.

Dufays A, Jacobs K, Liu Y, Rombouts J (2023). "Fast Filtering with Large Option Panels: Implications for Asset Pricing." *Journal of Financial and Quantitative Analysis*, pp. 1–32. doi:10.1017/S0022109023000753.

Duffie D, Pan J, Singleton K (2000). "Transform Analysis and Asset Pricing for Affine Jump-Diffusions." *Econometrica*, **68**(6), 1343–1376. doi:10.1111/1468-0262.00164.

Durham GB (2013). "Risk-Neutral Modeling With Affine and Nonaffine Models." *Journal of Financial Econometrics*, **11**(4), 650–681. doi:10.1093/jjfinec/nbt009.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Engle RF (1982). "Autoregressive Conditional Heteroscedasticity With Estimates of the Variance of United Kingdom Inflation." *Econometrica*, **50**(4), 987–1007. doi:10.2307/1912773.

Eraker B (2001). "MCMC Analysis of Diffusion Models With Application to Finance." *Journal of Business & Economic Statistics*, **19**(2), 177–191. doi:10.1198/073500101316970403.

Eraker B (2004). "Do Stock Prices and Volatility Jump? Reconciling Evidence from Spot and Option Prices." *Journal of Finance*, **59**(3), 1367–1403. doi:10.1111/j.1540-6261.2004.00666.x.

Eraker B, Johannes M, Polson N (2003). "The Impact of Jumps in Volatility and Returns." *Journal of Finance*, **58**(3), 1269–1300. doi:10.1111/1540-6261.00566.

Fama E, French K (1992). "The Cross-Section of Expected Stock Returns." *Journal of Finance*, **47**(2), 427–465. doi:10.1111/j.1540-6261.1992.tb04398.x.

Fama EF, French KR (2015). "A Five-Factor Asset Pricing Model." *Journal of Financial Economics*, **116**(1), 1–22. doi:10.1016/j.jfineco.2014.10.010.

Fridman M, Harris L (1998). "A Maximum Likelihood Approach for Non-Gaussian Stochastic Volatility Models." *Journal of Business & Economic Statistics*, **16**(3), 284–291. doi:10.1080/07350015.1998.10524767.

Gallant AR, Hsu CT, Tauchen G (1999). "Using Daily Range Data to Calibrate Volatility Diffusions and Extract the Forward Integrated Variance." *Review of Economics and Statistics*, **81**(4), 617–631. doi:10.1162/003465399558481.

Gordon NJ, Salmond DJ, Smith AF (1993). "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation." In *IEEE Proceedings F: Radar and Signal Processing*, volume 140, pp. 107–113. doi:10.1049/ip-f-2.1993.0015.

Heston SL (1993). "A Closed-Form Solution for Options With Stochastic Volatility With Applications to Bond and Currency Options." *Review of Financial Studies*, **6**(2), 327–343. doi:10.1093/rfs/6.2.327.

Hosszejni D, Kastner G (2021). "Modeling Univariate and Multivariate Stochastic Volatility in R with **stochvol** and **factorstochvol**." *Journal of Statistical Software*, **100**, 1–34. doi:10.18637/jss.v100.i12.

Hurn AS, Lindsay KA, McClelland AJ (2015). "Estimating the Parameters of Stochastic Volatility Models Using Option Price Data." *Journal of Business & Economic Statistics*, **33**(4), 579–594. doi:10.1080/07350015.2014.981634.

Ignatieva K, Rodrigues P, Seeger N (2015). "Empirical Analysis of Affine Versus Nonaffine Variance Specifications in Jump-Diffusion Models for Equity Indices." *Journal of Business & Economic Statistics*, **33**(1), 68–75. doi:10.1080/07350015.2014.922471.

Jacobs K, Liu Y (2018). "Estimation and Filtering with Big Option Data." *Working Paper*.

Jacquier E, Polson NG, Rossi PE (1994). "Bayesian Analysis of Stochastic Volatility Models." *Journal of Business & Economic Statistics*, **12**(4). doi:10.2307/1392199.

Johannes M, Polson N, Stroud J (2009). "Optimal Filtering of Jump Diffusions: Extracting Latent States From Asset Prices." *Review of Financial Studies*, **22**(7), 2759–2799. doi:10.1093/rfs/hhn110.

Johansen AM (2009). "**SMCTC**: Sequential Monte Carlo in C++." *Journal of Statistical Software*, **30**, 1–41. doi:10.18637/jss.v030.i06.

Jones CS (2003). "The Dynamics of Stochastic Volatility: Evidence from Underlying and Options Markets." *Journal of Econometrics*, **116**(1-2), 181–224. doi:10.1016/S0304-4076(03)00107-6.

Kaeck A, Alexander C (2012). "Volatility Dynamics for the S&P 500: Further Evidence From Non-affine, Multi-Factor Jump Diffusions." *Journal of Banking & Finance*, **36**(11), 3110–3121. doi:10.1016/j.jbankfin.2012.07.012.

Kastner G (2016). "Dealing With Stochastic Volatility in Time Series Using the R Package **stochvol**." *Journal of Statistical Software*, **69**(5), 1–30. doi:10.18637/jss.v069.i05.

Kastner G, Frühwirth-Schnatter S (2014). "Ancillarity-Sufficiency Interweaving Strategy (ASIS) for Boosting MCMC Estimation of Stochastic Volatility Models." *Computational Statistics & Data Analysis*, **76**, 408–423. doi:10.1016/j.csda.2013.01.002.

Kim S, Shephard N, Chib S (1998). "Stochastic Volatility: Likelihood Inference and Comparison With ARCH Models." *Review of Economic Studies*, **65**(3), 361–393. doi:10.1111/1467-937X.00050.

Kitagawa G (1987). "Non-Gaussian State-Space Modeling of Nonstationary Time Series." *Journal of the American Statistical Association*, **82**(400), 1032–1041. doi:10.2307/2289375.

Knaus P, Bitto-Nemling A, Cadonna A, Frühwirth-Schnatter S (2021). "Shrinkage in the Time-Varying Parameter Model Framework Using the R Package shrinkTVP." *Journal of Statistical Software*, **100**(13), 1–32. doi:10.18637/jss.v100.i13.

Koopman SJ, Hol Uspensky E (2002). "The Stochastic Volatility in Mean Model: Empirical Evidence from International Stock Markets." *Journal of Applied Econometrics*, **17**(6), 667–689. doi:10.1002/jae.652.

Krueger F (2015). ***bvarsv**: Bayesian Analysis of a Vector Autoregressive Model with Stochastic Volatility and Time-Varying Parameters.* R package version 1.1, URL https://CRAN.R-project.org/package=bvarsv.

Langrock R, MacDonald IL, Zucchini W (2012). "Some Nonstandard Stochastic Volatility Models and their Estimation Using Structured Hidden Markov Models." *Journal of Empirical Finance*, **19**(1), 147–161. doi:10.1016/j.jempfin.2011.09.003.

Lewis A (2000). *Option Valuation Under Stochastic Volatility (1st Edition).* Finance Press, Newport Beach, CA, United States of America.

Lord R, Koekkoek R, Dijk DV (2010). "A Comparison of Biased Simulation Schemes for Stochastic Volatility Models." *Quantitative Finance*, **10**(2), 177–194. doi:10.1080/14697680802392496.

Malik S, Pitt MK (2011). "Particle Filters for Continuous Likelihood Evaluation and Maximisation." *Journal of Econometrics*, **165**(2), 190–209. doi:10.1016/j.jeconom.2011.07.006.

Martino S, Aas K, Lindqvist O, Neef LR, Rue H (2011). "Estimating Stochastic Volatility Models Using Integrated Nested Laplace Approximations." *European Journal of Finance*, **17**(7), 487–503. doi:10.1080/1351847X.2010.495475.

McCausland WJ, Miller S, Pelletier D (2011). "Simulation Smoothing for State–Space Models: A Computational Efficiency Analysis." *Computational Statistics & Data Analysis*, **55**(1), 199–212. doi:10.1016/j.csda.2010.07.009.

Michaud N, de Valpine P, Turek D, Paciorek CJ, Nguyen D (2021). "Sequential Monte Carlo Methods in the **nimble** and **nimbleSMC** R Packages." *Journal of Statistical Software*, **100**, 1–39. doi:10.18637/jss.v100.i03.

Nelder JA, Mead R (1965). "A Simplex Method for Function Minimization." *Computer Journal*, **7**(4), 308–313. doi:https://doi.org/10.1093/comjnl/7.4.308.

Nordh J (2017). "**pyParticleEst**: A Python Framework for Particle-Based Estimation Methods." *Journal of Statistical Software*, **78**, 1–25. doi:10.18637/jss.v078.i03.

Omori Y (2022). ***ASV**: Stochastic Volatility Models With or Without Leverage.* R package version 1.1.1, URL https://CRAN.R-project.org/package=ASV.

Omori Y, Chib S, Shephard N, Nakajima J (2007). "Stochastic Volatility with Leverage: Fast and Efficient Likelihood Inference." *Journal of Econometrics*, **140**(2), 425–449. doi:10.1016/j.jeconom.2006.07.008.

Pitt MK, Malik S, Doucet A (2014). "Simulated Likelihood Inference for Stochastic Volatility Models Using Continuous Particle Filtering." *Annals of the Institute of Statistical Mathematics*, **66**(3), 527–552. doi:10.1007/s10463-014-0456-y.

R Core Team (2019). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ryan JA, Ulrich JM (2022). ***quantmod**: Quantitative Financial Modelling Framework.* R package version 0.4.20, URL https://CRAN.R-project.org/package=quantmod.

Ryan JA, Ulrich JM (2024). ***xts**: Extensible time series.* R package version 0.14.0, URL https://cran.r-project.org/package=xts.

Schwarz G (1978). "Estimating the Dimension of a Model." *The Annals of Statistics*, **6**(2), 461–464. doi:10.1214/aos/1176344136.

Taylor SJ (1986). *Modelling Financial Time Series (1st Edition).* John Wiley & Sons, New York, NY, United States of America.

Trolle AB, Schwartz ES (2009). "Unspanned Stochastic Volatility and the Pricing of Commodity Derivatives." *Review of Financial Studies*, **22**(11), 4423–4461. doi:10.1093/rfs/hhp036.

Wahl J (2020). ***stochvolTMB**: Likelihood Estimation of Stochastic Volatility Models.* R package version 0.2.0, URL https://CRAN.R-project.org/package=stochvolTMB.

Wang X, He X, Zhao Y, Zuo Z (2017). "Parameter estimations of Heston model based on consistent extended Kalman filter." *IFAC-PapersOnLine*, **50**(1), 14100–14105. doi:10.1016/j.ifacol.2017.08.1850.

Watanabe T (1999). "A Non-linear Filtering Approach to Stochastic Volatility Models With an Application to Daily Stock Returns." *Journal of Applied Econometrics*, **14**(2), 101–121. doi:10.1002/(SICI)1099-1255(199903/04)14:2<101::AID-JAE499>3.0.CO;2-A.

Woźniak T (2024). ***bsvars**: Bayesian Estimation of Structural Vector Autoregressive Models.* R package version 3.0, URL https://CRAN.R-project.org/package=bsvars.

Yu Y, Meng XL (2011). "To Center or Not to Center: That Is Not the Question—an Ancillarity—Sufficiency Interweaving Strategy (ASIS) for Boosting MCMC Efficiency." *Journal of Computational and Graphical Statistics*, **20**(3), 531–570. doi:10.1198/jcgs.2011.203main.

Zucchini W, MacDonald IL, Langrock R (2016). *Hidden Markov Models for Time Series: An Introduction Using R (2nd Edition)*. CRC press.

A. Default parameter initialization

The default parameter initialization algorithm returns a set of initial parameters Θ^{init} in cases where users do not provide them to the `DNFOptim` function (i.e., `par = NULL`) for built-in models.

The parameters associated with the return drift function μ^x and the long-run volatility θ are selected first. Also, rho_z is left at zero during as it is difficult to identify. Concretely, to start, we set

$$\mu^{init} = \overline{y}_{1:T}/h, \tag{6}$$

$$\rho_z^{init} = 0, \quad \text{and} \tag{7}$$

$$\theta^{init} = s^2_{y_{1:T}} \quad \text{for the jump-diffusion models (Models 4–6), and} \tag{8}$$

$$\theta^{init} = \log(s^2_{y_{1:T}}), \quad \text{for the discrete-time models (Models 1–3 and 7),} \tag{9}$$

where $\overline{y}_{1:T}$ and $s^2_{y_{1:T}}$ are the sample mean and sample variance of $y_{1:T}$, respectively. For the CAPM-SV model, that is, Model 7, we fit the simple linear regression

$$y_t = c_0 + c_1 R^m_t + \varepsilon^{CAPM}_t, \quad t = 1, \ldots T, \tag{10}$$

where ε^{CAPM}_t are independent identically distributed normal variables with mean 0 and standard deviation σ^{CAPM}, to get initial estimates c_0^{init} and c_1^{init}.

We then find sets of potential initial parameters $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(I)}$ and return $\Theta^{init} = \arg\max_{1 \leq i \leq I} \hat{\mathcal{L}}(\Theta^{(i)})$, the set which yields the highest likelihood. While the parameters set in Equations (6)–(10) remain constant through the parameter sets $\{\Theta^{(i)}\}_{i=1}^{I}$, the remaining model parameters are updated according to the following process.

The first set $\Theta^{(1)}$ has the jump parameters at zero and uses an exponentially weighted moving average (EWMA) to get estimates $\hat{X}_{1:T}^{(1)}$ of the latent variables. The EWMA with weighting factor w starts with $\hat{X}_0 \equiv s^2_{y_{1:T}}$ and sets

$$\hat{X}_t^{(1)} = w\hat{X}_{t-1}^{(1)} + (1-w)y_t^2. \tag{11}$$

With the estimates $\hat{X}_{1:T}^{(i)}$, we fit the regression

$$\hat{X}_t^{(i)} = \beta_0 + \beta_1 \hat{X}_{t-1}^{(i)} + \epsilon_t^{SV}, \quad t = 1, \ldots T, \tag{12}$$

where ϵ_t^{SV} are independent identically distributed normal variables with mean zero and standard deviation σ^{SV} for Models 4–6 and

$$\log(\hat{X}_t^{(i)}) = \beta_0 + \beta_1 \log(\hat{X}_{t-1}^{(i)}) + \epsilon_t^{SV}, \quad t = 1, \ldots T, \tag{13}$$

for Models 1–3 and 7.

The regression of Equations (12) and (13) give the coefficients estimates $\hat{\beta}_0$, $\hat{\beta}_1$, and $\hat{\sigma}^{SV}$ for the intercept, slope, and standard deviation for the linear model, respectively. For Models 4–7, we set

$$\begin{align}
\kappa^{(i)} &= (1 - \hat{\beta}_1)/h, \tag{14}\\
\sigma^{(i)} &= \hat{\sigma}^{SV}/(\sqrt{(\theta^{init} h}), \tag{15}\\
\hat{\varepsilon}_t^y &= \frac{y_t - \mu^{init}}{\sqrt{\theta^{init} h}}, \tag{16}\\
\hat{\varepsilon}_t^x &= \frac{\hat{X}_t^{(i)} - (\hat{X}_{t-1}^{(i)} + h\kappa^{(i)}(\theta^{init} - \hat{X}_{t-1}^{(i)}))}{\sigma^{(i)}\sqrt{\hat{X}_t^{(i)} h}}, \quad \text{and} \tag{17}\\
\rho^{(i)} &= r_{\hat{\varepsilon}_t^x, \hat{\varepsilon}_t^y}, \tag{18}
\end{align}$$

where $r_{\hat{\varepsilon}_t^x, \hat{\varepsilon}_t^y}$ is the sample correlation between $\hat{\varepsilon}_t^x$ and $\hat{\varepsilon}_t^x$. For Models 1–3 and 7, we apply the regression model given in Equation (13) and set the parameters as follows:

$$\begin{align}
\phi^{(i)} &= \hat{\beta}_1, \tag{19}\\
\sigma^{(i)}, &= \hat{\sigma}^{SV} \tag{20}
\end{align}$$

and, for Models 5 and 6,

$$\begin{align}
\hat{\varepsilon}_t^y &= \frac{y_t}{\exp(\hat{X}_{t-1}^{(i)}/2)}, \tag{21}\\
\hat{\varepsilon}_t^x &= \frac{\hat{X}_t^{(i)} - (\theta^{init} + \phi^{(i)}(\hat{X}_{t-1}^{(i)} - \theta^{init}))}{\sigma^{(i)}}, \quad \text{and} \tag{22}\\
\rho^{(i)} &= r_{\hat{\varepsilon}_t^x, \hat{\varepsilon}_t^y}. \tag{23}
\end{align}$$

We use the discrete nonlinear filter to obtain $\hat{\mathcal{L}}(\Theta^{(i)})$ and take the mean of the filtering distribution to get $\hat{X}_{1:T}^{(i+1)}$. The next set of volatility dynamic parameters are found using the regression of Equation (12) or Equation (13), depending on the model used.

When $i > 1$, we also estimate the jump parameters. We start by constructing 99% confidence intervals for y_t. For Models 5 and 6, this means

$$\mathrm{CI}_t^{(i)} = \mu^{(init)} \pm 2.58\sqrt{h\hat{X}_t^{(i)}}, \quad \text{and} \tag{24}$$

$$\mathrm{CI}_t^{(i)} = 0 \pm 2.58 \exp\left(X_t^{(i)}/2\right) \tag{25}$$

for Model 3.

As we expect 1% of returns to fall outside of these intervals, we get

$$p^{(i)} = \max\left[\frac{1}{T}\sum_{t=1}^{T} 1_{\{y_t \notin CI_t^{(i)}\}} - 0.01,\, 0\right] \quad \text{and} \tag{26}$$

$$\omega^{(i)} = p^{(i)}/h. \tag{27}$$

We then set $\alpha^{(i)}$ and $\delta^{(i)}$ to be the mean and standard deviation of $\{y_t : y_t \notin CI_t^{(i)}\}$, respectively. Then, $\nu^{(i)}$ is the average of $\{\hat{X}_t^{(i)} - \mathrm{E}\left[X_t \mid \hat{X}_{t-1}\right]$, for t such that $y_t \notin CI_t^{(i)}\}$. For Model 6, the only built-in model with volatility factor jumps,

$$\mathrm{E}\left[X_t \mid \hat{X}_{t-1}^{(i)}\right] = \hat{X}_{t-1}^{(i)} + \kappa^{(i)}(\theta^{init} - \hat{X}_{t-1}^{(i)})h.$$

With these parameters, we run the DNF to obtain $\hat{X}_{1:T}^{(i+1)}$ and apply the linear regression procedure described above to get the set of parameter $\Theta^{(i+1)}$.

Algorithm 2 summarizes the steps of the `DNFOptim` function's parameter initialization process. The `DNFOptim` function uses Algorithm 2 with $w = 0.9$ and $I = 20$.

Algorithm 2 Parameter initialization

```
set initial parameters using Equations (6) to (9)
obtain $\hat{X}_{1:T}^{(1)}$ using the EWMA of Equation (11) with weighting $w$
run linear regression and set volatility parameters using Equations (12) to (23)
for $i = 1, 2, \ldots, I$ do
    use the DNF evaluate $\hat{\mathcal{L}}(\Theta^{(i)})$ and get mean filtering distribution estimates $\hat{X}_{1:T}^{(i+1)}$
    run linear regression with $\hat{X}_{1:T}^{(i+1)}$ and set parameters $\Theta^{(i+1)}$ using Equations (12)–(23)
    compute the confidence intervals in Equation (24)–(25)
    set $\alpha^{(i+1)}$ and $\delta^{(i+1)}$ to be the mean and standard deviation of $\{y_t : y_t \notin CI_t^{(i)}\}$
    set $\nu^{(i+1)}$ to be the mean of $\{\hat{X}_t - \mathrm{E}\left[X_t \mid \hat{X}_{t-1}\right]$, for t such that $y_t \notin CI_t^{(i)}\}$
end for
return $\Theta^{init} = \arg\max_{1 \le i \le I} \hat{\mathcal{L}}(\Theta^{(i)})$
```

Affiliation:

Louis Arsenault-Mahjoubi
Department of Statistics and Actuarial Science
Simon Fraser University
8888 University Drive
Burnaby, BC, Canada V5A 1S6
E-mail: louis_arsenault-mahjoubi@sfu.ca

Jean-François Bégin
Department of Statistics and Actuarial Science
Simon Fraser University
8888 University Drive
Burnaby, BC, Canada V5A 1S6
E-mail: jbegin@sfu.ca

Mathieu Boudreault
Department of Mathematics
Université du Québec à Montréal
C.P. 8888, Succursale Centre-ville
Montréal, QC, Canada H3C 3P8
E-mail: boudreault.mathieu@uqam.ca