یادگیری ماشین

پروژه پایانی

کاربرد کوپمن در یادگیری ماشین

نازنین بنداریان ۴۰۲۱۳۸۷۶

تابستان ۱۴۰۳

فهرست مطالب

عنوان **صفحه**

آدرس گوگل کولب :

https://drive.google.com/drive/folders/1cmnVq2VN7JRvpxYHSpCO2cRNX0GYS-V0

آدرس گیت هاب:https://github.com/nazaninbondarian/MachineLearning2024/tree/main/Project

فهرست شکل‌ها

عنوان **صفحه**

۱- مقدمه

PyKoopman یک بسته پایتون برای تقریب داده-راند عملگر کوپمن مرتبط با یک سیستم پویا است. عملگر کوپمن یک تعبیه خطی اصولی از دینامیک غیرخطی است و پیش‌بینی، تخمین و کنترل دینامیک شدیدا غیرخطی را با استفاده از تئوری سیستم‌های خطی تسهیل می‌کند. به طور خاص، PyKoopman ابزارهایی را برای شناسایی سیستم-های داده-راند برای سیستم‌های غیراجباری و فعال ارائه می‌دهد که بر اساس تجزیه مود پویا بدون معادله (DMD) و انواع آن ساخته می‌شوند. در این کار، ما شرح مختصری از زیربنای ریاضی عملگر Koopman، نمای کلی و نمایش ویژگی‌های پیاده‌سازی شده در PyKoopman (با مثال‌های کد)، توصیه‌های عملی برای کاربران، و فهرستی از برنامه‌های افزودنی بالقوه PyKoopman ارائه می‌کنیم. در بخش آخر نیز جندین مورد از نحوه استفاده از این بسته ارائه شده است.

نرم افزار در https://github.com/dynamicslab/pykoopman موجود است.

۲- مروری بر ادبیات گذشته

۲-۱- مقدمه

مهندسان برای پر کردن شکاف بین توصیف‌های ساده و خطی در جایی که ابزارهای تحلیلی قدرتمند وجود دارد و پیچیدگی‌های پیچیده دینامیک غیرخطی در جایی که راه‌حل‌های تحلیلی گریزان هستند، بر خطی‌سازی تکیه کرده‌اند. خطی‌سازی محلی، که از طریق تقریب سری تیلور مرتبه اول پیاده سازی شده است، به طور گسترده در شناسایی سیستم، بهینه سازی و بسیاری از زمینه‌های دیگر برای قابل حل کردن مشکلات استفاده شده است. با این حال، بسیاری از سیستم‌های دنیای واقعی اساساً غیرخطی هستند و به راه‌حل‌هایی در خارج از همسایگی محلی که خطی‌سازی معتبر است نیاز دارند. پیشرفت سریع در روش‌های یادگیری ماشین و داده‌های بزرگ باعث پیشرفت در مدل‌سازی داده-راند چنین سیستم‌های غیرخطی در علم و مهندسی شده است. نظریه عملگر کوپمن به طور خاص به عنوان یک رویکرد اصولی برای تعبیه دینامیک غیرخطی در یک چارچوب خطی که فراتر از خطی‌سازی ساده است، ظهور کرده است.

PyKoopman یک بسته پایتون است که برای تقریب عملگر کوپمن طراحی شده است. به طور خاص، PyKoopman ابزارهایی را برای طراحی قابل مشاهده‌ها (یعنی توابع حالت سیستم) و استنباط یک عملگر خطی با ابعاد محدود ارائه می‌دهد که بر تکامل دینامیکی این مشاهدات در زمان حاکم است. همانطور که در مدل‌های شبکه عصبی اخیر نشان داده شده است، این مراحل می‌توانند به صورت متوالی یا ترکیبی انجام شوند. هنگامی که یک تعبیه خطی از داده ها کشف شد، خطی بودن سیستم دینامیکی تبدیل شده را می توان برای افزایش تفسیرپذیری یا برای طراحی مشاهده‌گر یا کنترل‌کننده‌های نزدیک به بهینه برای سیستم غیرخطی اصلی مورد استفاده قرار داد.

بسته PyKoopman هم برای محققین و هم برای پزشکان طراحی شده است و هر کسی را که به داده‌ها دسترسی دارد قادر می‌سازد تعبیه‌های سیستم‌های غیرخطی را که در آن دینامیک تقریباً خطی می‌شود، را کشف کند. با پیروی از PySINDy و Deeptime، PyKoopman به گونه‌ای طراحی شده است که برای کسانی که دانش اولیه سیستم‌های خطی را دارند، کاربر پسند باشد، به استانداردهای scikit-learn پایبند باشد، در حالی که اجزای مدولار را برای کاربران پیشرفته‌تر نیز ارائه می‌دهد.

۲-۲- نظریه عملگر کوپمن

در این قسمت به طور مختصر به توضیح تئوری عملگر کوپمن برای سیستم‌های دینامیکی می‌پردازیم. به طور خاص، تئوری سیستم‌های دینامیکی مستقل در بخش ۲-۲-۲- ارائه شده است در حالی که نظریه برای سیستم‌های کنترل‌شده در بخش ۳-۲-۲- ارائه شده است.

۲-۲-۱- پیش‌زمینه

PyKoopman پیاده‌سازی پایتون از چندین الگوریتم پیشرو را برای تقریب داده-راند عملگر کوپمن مرتبط با یک سیستم دینامیکی فراهم می‌کند.

$$\frac{d}{dt}x(t) = f(x(t)u(t))$$

که در آن $x \in \mathcal{M} \subseteq \mathbb{R}^n$ حالت سیستم است و f یک میدان برداری است که دینامیک و اثر ورودی کنترل $u \in \mathbb{R}^q$ را توصیف می‌کند.

سیستم خودگردان زیر را در نظر بگیرید.

$$\frac{d}{dt}x(t) = f\big(x(t)\big)$$

داده‌ها معمولاً به صورت گسسته در زمان در فواصل Δt نمونه‌برداری می‌شوند و سیستم دینامیکی زمان گسسته مربوطه توسط تبدیل غیرخطی $F: \mathcal{M} \mapsto \mathcal{M}$ داده می‌شود:

$$x(t + \Delta t) = F\big(x(t)\big)$$

که $F(x) = x(t) + \int_t^{t+\Delta t} f\big(x(s)\big)ds$ می‌باشد.

با توجه به داده‌ها به شکل بردارهای اندازه‌گیری $x(t)$، هدف نظریه کوپمن داده-راند (با توجه به شکل ۱-۲) یافتن یک سیستم مختصات جدید است.

$$z \coloneqq \Phi(x)$$

که در آن دینامیک ساده شده، یا به طور ایده‌آل، خطی به معنای دینامیک پیوسته است،

$$\frac{d}{dt}z = A_c z$$

یا دینامیک زمان گسسته

$$z(t + \Delta t) = Az(t)$$

که در آن زیرنویس c برای زمان پیوسته و $A = \exp(\Delta t A_c)$ است. برای سادگی، PyKoopman بر روی سیستم دینامیکی گسسته در معادله بالا متمرکز شده است.

هدف از یادگیری مختصات Φ و دینامیک خطی A ممکن است به عنوان یک مسئله رگرسیون از نظر یافتن عملگر خطی که وضعیت سیستم را به بهترین شکل تبدیل می‌کند، یا یک نسخه تبدیل شده از حالت، در زمان به جلو مطرح شود. این ممکن است بر اساس دو ماتریس داده زیر فرموله شود:

$$X = \begin{bmatrix} | & | & & | \\ x(t_1) & x(t_2) & \cdots & x(t_m) \\ | & | & & | \end{bmatrix}, \acute{X} = \begin{bmatrix} | & | & & | \\ x(\acute{t}_1) & x(\acute{t}_2) & \cdots & x(\acute{t}_m) \\ | & | & & | \end{bmatrix}$$

یا ماتریس‌های داده تبدیل شده مشاهدات غیرخطی نامزد به صورت زیر است

$$\Phi(X) = \begin{bmatrix} | & | & & | \\ \Phi\big(x(t_1)\big) & \Phi\big(x(t_2)\big) & \cdots & \Phi\big(x(t_m)\big) \\ | & | & & | \end{bmatrix}, \Phi\big(\acute{X}\big) = \begin{bmatrix} | & | & & | \\ x(\acute{t}_1) & x(\acute{t}_2) & \cdots & x(\acute{t}_m) \\ | & | & & | \end{bmatrix}$$

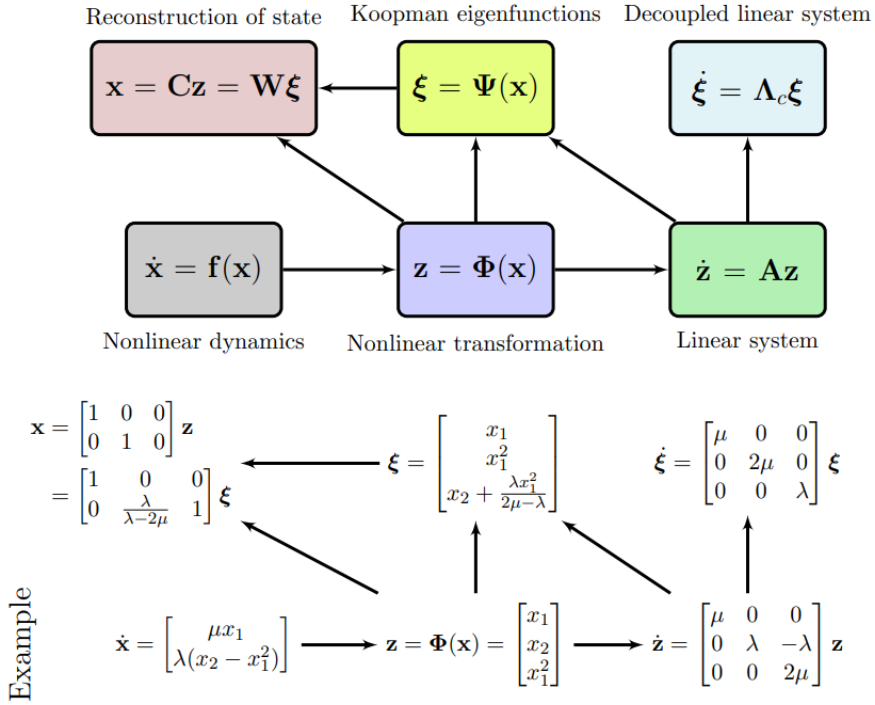
سپس رگرسیون زیر برای یک A ناشناخته حل تقریبی انجام می‌شود.

$$\Phi\big(\acute{X}\big) \approx A\Phi(X)$$

انتخاب Φ وابسته به مسئله است. انتخاب‌های رایج عبارتند از ویژگی‌های چندجمله‌ای، ویژگی‌های ضمنی تعریف شده توسط توابع کرنل، توابع پایه شعاعی، تعبیه تاخیر زمانی، و ویژگی‌های تصادفی فوریه. در حالی که اکثر فرمول‌بندی‌های اولیه تقریب کوپمن داده-راند به شدت به حداقل مربعات معمولی یا SVD-DMD متکی هستند، می‌توان از هر رگرسیون از جامعه DMD (مثلاً با استفاده از PyDMD) برای حل معادله بالا استفاده کرد؛ از جمله مجموع حداقل مربعات (tlsDMD)، DMD بهینه شده (optDMD) و غیره.

اگرچه عملگر کوپمن و انواع آن در زمینه دینامیک سیالات برای تجزیه و تحلیل مودال منشا گرفته است، اما ایده‌های متعددی را در جامعه کنترل الهام گرفته است، مانند کنترل بهینه کوپمن، کنترل پیش بینی مدل کوپمن (MPC)، یادگیری تقویتی کوپمن، و ناظران مبتنی بر کوپمن و فیلترهای کالمن. علاوه بر این، کاربرد عملگر کوپمن به طور گسترده در شناسایی مدل کنترل محور در زمینه هایی مانند رباتیک، پیش بینی آب و هوا و پیش بینی سری های

زمانی به کار گرفته شده است. با این حال، در حال حاضر هیچ پیاده‌سازی منبع باز استانداردی برای تقریب عملگر کوپمن از داده‌ها وجود ندارد. در نتیجه، محققان ملزم به توسعه نسخه‌های خود هستند، حتی اگر علایق اولیه آنها ممکن است در برنامه‌های پایین دستی عملگر کوپمن باشد. این انگیزه این کار فعلی را برای استانداردسازی اجرای عملگر کوپمن با ایجاد PyKoopman ایجاد کرده است. این پلتفرم برای خدمت به عنوان یک مرکز مرکزی برای آموزش عملگر کوپمن، آزمایش با تکنیک‌های مختلف، و یک جعبه ابزار آماده برای کاربران نهایی طراحی شده است تا الگوریتم‌های کوپمن داده-راندرا به طور یکپارچه در خطوط مسیر وظیفه خود ادغام کنند.



شکل ۱-۲ بالا بردن حالت x سیستم دینامیکی مستقل پیوسته به یک سیستم مختصات جدید، که در آن دینامیک غیرخطی اصلی خطی می‌شود و مدیریت آن آسانتر است. همچنین می‌توان به صورت خطی حالت x را از سیستم مختصات جدید بازسازی کرد. این امر با PyKoopman به روش داده-راند تسهیل می‌شود.

۲-۲-۲- نظریه اپراتور کوپمن برای سیستم‌های دینامیکی

با توجه به سیستم دینامیکی زمان پیوسته زیر،

$$\frac{d}{dt}x(t) = f\big(x(t)\big)$$

عملگر تبدیل جریان، یا تبدیل $\text{time}-t$، $F^t: \mathcal{M} \to \mathcal{M}$ شرایط اولیه $x(0)$ را به نقاطی در مسیر t واحدهای زمانی در آینده ترسیم می‌کند، به طوری که مسیرها مطابق با $x(t) = F^t\big(x(0)\big)$ تکامل می‌یابند. عملگر کوپمن، $K^t: \mathcal{G}(\mathcal{M}) \to \mathcal{G}(\mathcal{M})$ تابع اندازه گیری $g \in \mathcal{G}(\mathcal{M})$ ارزیابی شده در نقطه $x(t_0)$ را به همان تابع اندازه‌گیری ارزیابی شده در نقطه $x(t_0 + t)$ ترسیم می‌کند:

$$K^t g(x) = g\big(F^t(x)\big)$$

که در آن $\mathcal{G}(\mathcal{M})$ مجموعه ای از توابع اندازه گیری $\mathcal{G}: \mathcal{M} \to \mathbb{C}$ است. مولد بینهایت کوچک $\mathcal{L}$ عملگر $\text{time}-t$ کوپمن به عنوان عملگر Lie شناخته می‌شود، زیرا مشتق Lie از g در امتداد میدان برداری f(x) است که دینامیک با معادله اول داده می‌شود. این از اعمال قانون زنجیره‌ای به مشتق زمانی $g(x)$ به دست می‌آید:

$$\frac{d}{dt}g\big(x(t)\big) = \nabla g.\dot{x}(t) = \nabla g.f\big(x(t)\big) = \mathcal{L}g\big(x(t)\big)$$

در زمان پیوسته، عملگر Lie تابع ویژه $\varphi(x)$ برآورده می‌شود.

$$\frac{d}{dt}\varphi(x) = \mathcal{L}\varphi(x) = \mu\varphi(x)$$

یک تابع ویژه φ از L با مقدار ویژه μ، سپس یک تابع ویژه از K^t با مقدار ویژه $\lambda^t = \exp(\mu t)$ است. با این حال، ما اغلب چندین اندازه‌گیری از یک سیستم انجام می‌دهیم که آنها را در بردار g ترتیب می‌دهیم:

$$g(x) = \begin{bmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_p(x) \end{bmatrix}$$

بردار مشاهده‌پذیرها، g، را می‌توان بر حسب مبنای توابع ویژه $\varphi_j(x)$ گسترش داد:

$$K^t g(x) = \sum_{j=1}^{\infty} \lambda^t \varphi_j(x) v_j$$

جایی که $v_j := \left[\left\langle \varphi_j, g_1 \right\rangle, \left\langle \varphi_j, g_2 \right\rangle, \cdots, \left\langle \varphi_j, g_p \right\rangle\right]$ j-امین مود کوپمن است که با تابع ویژه φ_j مرتبط است.

برای یک سیستم زمان گسسته:

$$x_{k+1} = F(x_k)$$

که در آن $x_k = x(t_k) = x(k\Delta t)$، عملگر کوپمن K، تکامل یک مرحله‌ای تابع اندازه‌گیری g را کنترل می‌کند،

$$Kg(x_k) = g\big(F(x_k)\big) = g(x_{k+1})$$

در این مورد، یک تابع ویژه کوپمن، $\varphi(x)$ مربوط به مقدار ویژه λ را برآورده می‌کند.

$$\varphi(x_{k+1}) = K\varphi(x_k) = \lambda\varphi(x_k)$$

۲-۲-۳- نظریه کوپمن برای سیستم‌های کنترل شده

دینامیک زمان پیوسته برای یک سیستم کنترل شده توسط

$$\frac{d}{dt}x(t) = f(x(t)u(t))$$

به جای حالت معمول x، توابع اندازه‌گیری را در حالت توسعه یافته $\tilde{x} = (x, u)$ در نظر می‌گیریم، که در آن تبدیل جریان مربوطه $\tilde{F}^t(x,u) = [F^t(x,u), \Theta^t(u)]$ و $\Theta^t(u)$ تبدیل تغییر زمان t واحد است به طوری که $\Theta^t(u)(s) = u(s+t)$ می‌باشد.

به طور خلاصه، اپراتور کوپمن در سیستم کنترل شده، تابع اندازه‌گیری حالت توسعه یافته را کنترل می‌کند.

$$K^t g(x,u) = g\left(\tilde{F}^t(x,u)\right)$$

تجزیه مود کوپمن مربوطه برای بردار قابل مشاهده،

$$g(x,u) = \begin{bmatrix} g_1(x,u) \\ g_2(x,u) \\ \vdots \\ g_p(x,u) \end{bmatrix}$$

می‌توان به صورت نوشتاری،

$$K^t g(x,u) = \sum_{j=1}^{\infty} \lambda^t \varphi_j(x,u) v_j$$

که در آن تابع ویژه کوپمن به صورت زیر است

$$\varphi(x,u,t) = K^t\varphi(x,u) = \lambda\varphi(x,u)$$

اگر سیستم کنترل شده با زمان پیوسته کنترلی باشد،

$$f\big(x(t),u(t)\big) = f_0(x) + \sum_{i=1}^{q} f_i(x)u_i$$

که در آن u_i i-امین مولفه ورودی u است، سپس عملگر Lie (در امتداد میدان برداری f) در تابع اندازه‌گیری $g(x)$ تبدیل می‌شود،

$$\mathcal{L}g(x) = \nabla_x g(x).\dot{x} = \nabla_x g(x).f_0(x) + \nabla_x g(x).\sum_{i=1}^{q} f_i(x)u_i$$

به طور مشابه، پس از اینکه عملگر Lie را در امتداد مسیر برداری f_0 به صورت $\mathcal{A}$ و آن را در امتداد f_i به صورت $\mathcal{B}_i$ تعریف کردیم، دوخطی‌سازی را برای سیستم کنترلی وابسته خواهیم داشت.

$$\frac{d}{dt}g(x) = \mathcal{A}g(x) + \sum_{i=1}^{q} u_i\mathcal{B}_i$$

با فرض اینکه φ یک تابع ویژه از $\mathcal{A}$ باشد، داریم

$$\frac{d}{dt}\varphi(x) = \mu\varphi(x) + \nabla_x\varphi(x).\sum_{i=1}^{q} f_i(x)u_i$$

بعلاوه، اگر فضای برداری که توسط D پوشانده شده باشد، چنین توابع ویژه $\{\varphi_i\}_{i=1}^{D}$ تحت $\mathcal{B}_1, \dots, \mathcal{B}_q$ ثابت باشد، داریم:

$$\forall i = 1, \dots, q,\ \mathcal{B}_i\varphi = B_i\varphi$$

که $\varphi = [\varphi_1 \quad \cdots \quad \varphi_D]^T$ می‌باشد.

ادغام کردن این دو معادله، ما فرم دوخطی معروف کوپمن را برای سیستم‌های کنترلی وابسته داریم،

$$\frac{d}{dt}\varphi(x) = \Lambda_c\varphi(x) + \sum_{i=1}^{q} u_i B_i\varphi$$

برای سیستم کلی زمان گسسته

$$x_{k+1} = F(x_k, u_k)$$

که در آن $x_k = x(t_k) = x(k\Delta t)$، عملگر کوپمن تکامل یک مرحله‌ای تابع اندازه‌گیری g را در حالت توسعه یافته $\tilde{x} = (x, u)$ کنترل می‌کند،

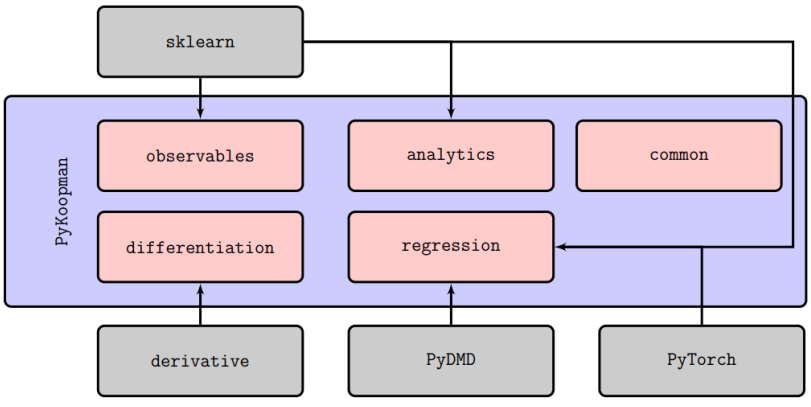
$$Kg(x_k, u_k) = g\big(F(x_k, u_k)\big) = g(x_{k+1}, u_{k+1})$$

یک تابع ویژه کوپمن، $\varphi(x)$ مربوط به مقدار ویژه λ را برآورده می‌کند.

$$\varphi(x_{k+1}, u_{k+1}) = K\varphi(x_k, u_k) = \lambda\varphi(x_k, u_k)$$

۲-۳- ویژگی‌ها

جزء اصلی بسته PyKoopman کلاس مدل Koopman است. برای اینکه این بسته برای یک پایگاه کاربر گسترده‌تر قابل دسترسی باشد، این کلاس به عنوان یک تخمین‌گر scikit-learn پیاده‌سازی می‌شود. وابستگی به بسته‌های خارجی در شکل ۲-۲ نشان داده شده است. علاوه‌براین، کاربران می‌توانند خطوط مسیر پیچیده‌ای را برای تنظیم هایپرپارامتر و انتخاب مدل با ادغام pykoopman با scikit-learn ایجاد کنند.



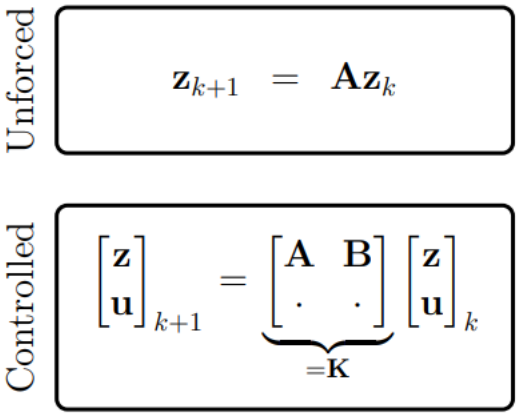
شکل ۲-۲ وابستگی به بسته‌های خارجی PyKoopman.

همانطور که در شکل ۳-۲ نشان داده شده است، PyKoopman برای بالا بردن دینامیک غیر خطی به یک سیستم خطی با تحریک خطی طراحی شده است. به طور خاص، پیاده سازی PyKoopman ما شامل دو مرحله اصلی است:

۱. observable: مشاهده پذیرهای غیرخطی که برای بالا بردن x به z و بازسازی x از z استفاده می‌شوند.

۲. regression: رگرسیون مورد استفاده برای یافتن بهترین عملگر دینامیک A.

علاوه بر این، ما یک ماژول differentiation داریم که مشتق زمان از یک مسیر و یک ماژول analytics برای تقریب‌های دلخواه عملگر کوپمن را ارزیابی می‌کند.



شکل ۳-۲ دسته‌بندی گسترده انواع مدل‌هایی که با PyKoopman فعلی قابل شناسایی هستند. در حالی که قطعات نقطه‌دار (که با «·» مشخص شده‌اند) می‌توانند به طور همزمان در چارچوب کشف شوند، معمولاً برای اهداف کنترلی نادیده گرفته می‌شوند.

در زمان نگارش، ویژگی‌های زیر پیاده سازی شده است:

- کتابخانه observable برای بالا بردن حالت x به فضای قابل مشاهده
  - شناسایی (برای DMD/DMDc یا در صورتی که کاربران بخواهند خود مشاهده‌پذیرها را محاسبه کنند): Identity
  - چندجمله‌ای‌های چند متغیره: Polynomial
  - مختصات تاخیر زمانی: TimeDelay
  - توابع پایه شعاعی: RadialBasisFunctions
  - ویژگی‌های تصادفی فوریه: RandomFourierFeatures
  - کتابخانه سفارشی (تعریف شده توسط توابع ارائه شده توسط کاربر): CustomObservables
  - الحاق مشاهده‌پذیرها: ConcatObservables
- روش شناسایی سیستم برای انجام رگرسیون
  - تجزیه حالت دینامیک: PyDMDRegressor
  - تجزیه حالت دیتامیک با کنترل: DMDc
  - تجزیه حالت دینامیک تعمیم یافته: EDMD
  - تجزیه حالت دینامیک تعمیم یافته با کنترل: EDMDc
  - تجزیه حالت دینامیک کرنل: KDMD
  - هنکل DMD: HDMD
  - هنکل DMD با کنترل: HDMDc

- DMD شبکه عصبی: NNDMD
- ساخت پراکنده زیرفضای ثابت کوپمن
  - یادگیری چند وظیفه ای بر اساس سازگاری خطی: ModesSelectionPAD21
- مشتق‌گیری عددی برای محاسبه $\dot{X}$ از X
  - مشتق محدود: FiniteDifference
  - مشتق محدود مرکزی مرتبه چهارم: Derivative(kind='finite_difference')
  - Savitzky-Golay با چند جمله‌ای مکعبی: Derivative(kind='savitzky-golay')
  - مشتق طیفی: Derivative(kind='spectral')
  - مشتق Spline: Derivative(kind='spline')
  - مشتق تغییرات کلی منظم: Derivative(kind='trend_filtered')
- سیستم‌های دینامیکی معیار مشترک
  - مدل فضای حالت زمان گسسته تصادفی، پایدار وخطی: drss
  - اسیلاتور Van del Pol: vdp-osc
  - سیستم لورنز: lorenz
  - دینامیک خطی دو بعدی: Linear2Ddynamics
  - دینامیک خطی روی چنبره: torus_dynamics
  - اسیلاتور دافینگ تحت نیرو: forced_duffing
  - معادله Cubic-quintic Ginzburg-Landau: cqgle
  - معادله Kuramoto-Sivashinsky: ks
  - معادله غیرخطی شرودینگر: nls
  - معادله ویسکوز برگرز: vbe
- روال‌های اعتبارسنجی برای بررسی‌های سازگاری

تقریباً تمام اشیاء PyKoopman از فرمت داده یک قدم جلوتر پشتیبانی می‌کنند، به جز زمانی که به صراحت به تاخیر زمانی نیاز است، مانند HAVOK. علاوه بر این، NNDMD نه تنها از فرمت استاندارد یک مرحله‌ای پشتیبانی می‌کند، بلکه داده‌ها را با مسیرهای چند مرحله‌ای نیز در خود جای می‌دهد.

۲-٤- نکات عملی

در این بخش، ما راهنمایی‌های عملی را برای استفاده مؤثر از Pykoopman ارائه می‌دهیم.

۲-٤-۱- انتخاب Observables

استفاده از observable غیرخطی، تقریب اپراتور کوپمن را اساساً با DMD متفاوت می‌کند. با این حال، انتخاب observable در عمل می‌تواند یک کار بسیار غیرمنطقی باشد. ویژگی‌های چندجمله‌ای برای سیستم‌های عملی در روباتیک یا پویایی سیال قابل مقیاس نیست. به عنوان یک نمونه قانون شست در عمل، می‌توان عملکرد پایه شعاعی صفحه نازک را به عنوان اولین انتخاب امتحان کرد. اگر تعداد نمونه‌برداری فوری داده در زمان فقط چند صدم باشد (به عنوان مثال، مانند دینامیک سیال)، می‌توان کرنل DMD را انتخاب کرد، اما تنظیم HyperParameters در

عملکرد کرنل می‌تواند بسیار مهم باشد. اگر تعداد نقاط داده بیش از چند هزار (به عنوان مثال، چندین مسیر از سیستم های روباتیک شبیه‌سازی شده) باشد، می‌توان روش کرنل را با ویژگی های فوریه تصادفی در observables.RandomFourierFeatures به عنوان Observables انتخاب کرد.

یکی دیگر از رویکردهای مفید، observable تأخیر در زمان است که می‌توان با استفاده از عملکرد تبدیل جریان معکوس به صورت بازگشتی به عنوان observable تفسیر کرد. با این حال ، خود شروع به کار نمی کند. درست مانند مدل‌های خودکار، تعداد تأخیرها تعیین‌کننده حداکثر تعداد حالت‌های خطی مناسب است که مدل می‌تواند ثبت کند. تعداد تأخیرها نیز تأثیر شگفت انگیزی بر شرایط عددی دارد.

بعلاوه، ممکن است استفاده از observable سفارشی شده توسط معادله حاکم بر سیستم خودکار با فراخوانی observables.CustomObservables با توابع لامبدا مفید باشد. اگر همه روش‌های فوق با شکست مواجه شوند، ممکن است از شبکه عصبی برای جستجوی observables استفاده شود. این رویکرد معمولاً گویاتر است اما از نظر محاسباتی نیز گران‌تر است.

۲-٤-۲- بهینه‌ساز

هنگامی که observable انتخاب شدند، مرحله بهینه‌سازی بهترین عملگر خطی را پیدا می‌کند که observable در مرحله زمانی فعلی را به مرحله زمانی بعدی نگاشت می‌کند. اگرچه اغلب اوقات رگرسیون حداقل مربعات استاندارد یا شبه معکوس کافی است، می توان از هر رگرسیون PyDMD استفاده کرد. علاوه بر این، می توان از NNDMD برای جستجوی همزمان برای observable و آموزش خطی بهینه استفاده کرد.

با توجه به NNDMD، ما متوجه شده‌ایم که استفاده از تلفات مکرر منجر به عملکرد مدل دقیق‌تر و قوی‌تر از تلفات استاندارد یک مرحله‌ای می‌شود که در الگوریتم‌های سنتی‌تر اتخاذ شده است. به لطف نمودار پویا در PyTorch، NNDMD می‌تواند تلفات مکرر را به تدریج به حداقل برساند، از کمینه کردن تنها یک مرحله آتی از دست دادن تا چندین مرحله در آینده. علاوه بر این، ما دریافتیم که استفاده از الگوریتم‌های بهینه‌سازی مرتبه دوم، مانند L-BFGS، به طور قابل‌توجهی آموزش را در مقایسه با بهینه‌ساز Adam تسریع می‌کند. با این حال، گاهی اوقات L-BFGS استاندارد می‌تواند واگرا شود، به خصوص زمانی که در یک دوره زمانی طولانی آموزش داده شود. با PyTorch.Lightning، NNDMD می‌تواند به راحتی از قدرت محاسباتی پلتفرم‌های سخت‌افزاری مختلف استفاده کند.

۲-۵- ویژگی‌های اضافی

در این بخش، افزونه‌ها و پیشرفت‌های بالقوه پیاده‌سازی PyKoopman را فهرست شده است.

- Bilinearization: اگرچه در حالت ایده‌آل ما می‌خواهیم یک سیستم ورودی-خروجی خطی استاندارد در مختصات تبدیل شده داشته باشیم، این می‌تواند منجر به ناسازگاری با سیستم اصلی شود. شایان ذکر است که Bilinearization در بسته پایتون دیگری به نام pykoop گنجانده شده است.
- طیف پیوسته: اکثر الگوریتم های موجود طیفی گسسته و نقطه ای را در داده ها منعکس می کنند. در نتیجه، این الگوریتم‌ها ممکن است با سیستم‌های پر هرج و مرج که حاوی یک طیف پیوسته هستند، دست و پنجه نرم کنند. چندین روش برای مدیریت طیف های پیوسته وجود دارد، از جمله استفاده از مختصات تاخیر

زمانی. رویکردهای اخیر از جمله resDMD، MPEDMD، و DMD با اطلاعات فیزیک، همگی نویدبخش دینامیک طیف پیوسته هستند.

- کتابخانه‌های توسعه یافته: سیستم خطی شناسایی شده در فضای ابعاد بیشتر را می‌توان برای تسهیل طراحی کنترل بهینه برای سیستم‌های غیرخطی بیشتر مورد بهره برداری قرار داد. به عنوان مثال، LQR کلاسیک به سیستم‌های غیرخطی گسترش یافته است. علاوه بر این، MPC غیرخطی را می‌توان با استفاده از سیستم خطی شناسایی شده از عملگر Koopman به MPC خطی تبدیل کرد، که مسئله بهینه‌سازی غیرمحدب اصلی را به یک مسئله بهینه‌سازی محدب تبدیل می‌کند. در آینده، می‌توان این انتظار را داشت که کتابخانه‌های منبع باز برای ترکیب کنترل مبتنی بر Koopman که با PyKoopman یکپارچه شده‌اند، به طور گسترده توسط جامعه استفاده خواهند شد[۱].

۳- مثال‌ها

۳-۱- **مثال اول: Extended DMD for slow manifold**

برای داده‌های خود، سیستم غیر خطی ساده را با یک نقطه ثابت و یک slow manifold در نظر می‌گیریم:

$$\dot{x}_1 = \mu x_1$$
$$\dot{x}_2 = \lambda(x_2 - x_1^2)$$

برای $\lambda < \mu < 0$ سیستم دارای یک slow attracting manifold در راستای $x_2 = x_1^2$ و نقطه ثابت $(0,0)$ می‌باشد.

ابتدا سیستم دینامیکی زیر را در نظر بگیرید.

$$\dot{x}_1 = -0.05x_1$$
$$\dot{x}_2 = -x_2 + x_1^2$$

برای پیاده‌سازی در پایتون در ابتدا باید بسته PyKoopman را نصب کنیم.

```
pip install pykoopman
```

کتابخانه‌های مورد نیاز را فراخوانی می‌کنیم.

```
import numpy as np
from scipy . integrate import odeint
import matplotlib.pyplot as plt
import random

from pykoopman import Koopman
from pykoopman.observables import Polynomial
from pykoopman.regression import EDMD
```

در پایتون معادله‌ی بالا به صورت زیر تعریف می‌گردد.

```
def slow_manifold(x, t):
   return [
      -0.05 * x[0],
     -x[1] + x[0]**2
   ]
```

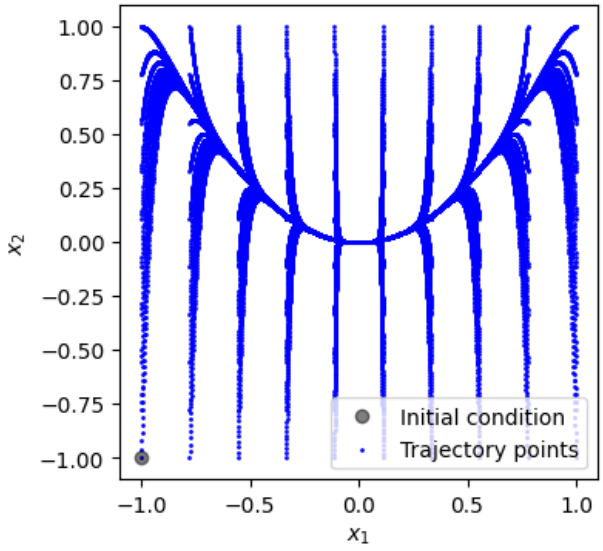
برای تهیه داده‌های آموزشی، ۱۰۰ عدد تصادفی را در $[-1,1]$ به عنوان شرایط اولیه ترسیم می‌کنیم و سپس با استفاده از معادله سیستم رو به جلو در زمان، مسیرهای مربوطه را جمع آوری می‌کنیم:

```
X = []
Xnext = []
for x0_0 in np.linspace (-1, 1, 10):
  for x0_1 in np.linspace (-1, 1, 10):
    x0 = np.array ([x0_0 , x0_1])
    x_tmp = odeint(slow_manifold, x0, t)
    X.append (x_tmp [:-1,:] )
    Xnext.append ( x_tmp [ 1:,:])

X = np.vstack (X)
Xnext = np.vstack (Xnext)
```

توجه داشته باشید که X و Xnext با X و $\acute{X}$ مطابقت دارند. برای نمایش داده‌های خروجی سیستم موردنظر از کد زیر استفاده شده است (شکل ۱-۳).

```
plt.figure(figsize=(4, 4))
plt.plot(X[0, 0], X[0, 1], 'o', color='black', label="Initial condition", alpha=0.5)
```

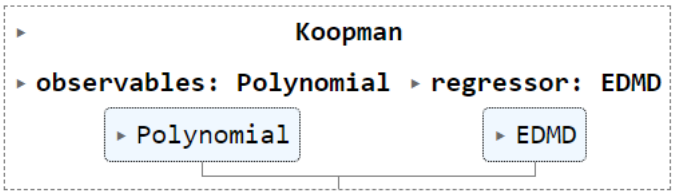
```
plt.scatter(X[:, 0], X[:, 1], s=1, color='blue', label='Trajectory points')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend(loc='best')
plt.show()
```



شکل ۱-۳ داده‌های اندازه‌گیری با استفاده از slow_manifold شبیه‌سازی شده است.

بسته PyKoopman حول کلاس Koopman ساخته شده است که عملگر کوپمن زمان گسسته را از داده‌ها تقریب می‌زند. برای شروع، می‌توانیم یک تابع observable و یک رگرسیون مناسب ایجاد کنیم. سپس این دو شی به عنوان ورودی برای کلاس Koopman عمل خواهند کرد. برای مثال، می‌توانیم از EDMD برای تقریب دینامیک سیستم تعریف شده، استفاده کنیم.

```
model = Koopman (observables=Polynomial(2), regressor=EDMD())
model.fit (X, Xnext)
```

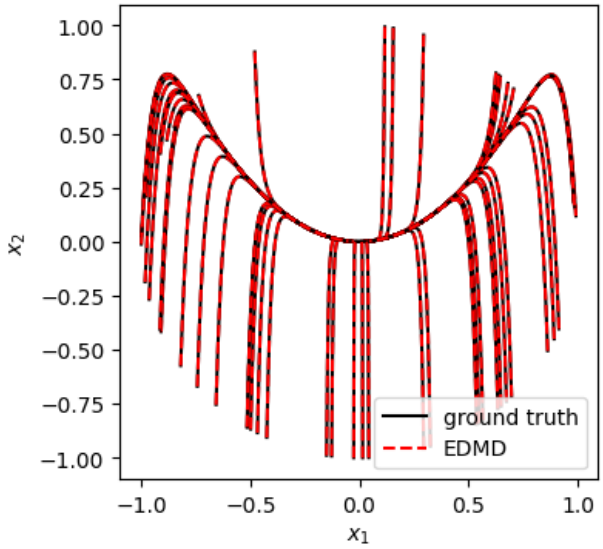


شکل ۲-۳ عملگر کوپمن زمان گسسته

هنگامی که شی Koopman آموزش دید، می‌توانیم از روش model.simulate برای پیش‌بینی در یک افق زمانی دلخواه استفاده کنیم. برای مثال، کد زیر استفاده از model.simulate را برای پیش‌بینی ۵۰ شرایط اولیه دیده نشده نمونه‌برداری شده در دایره واحد نشان می‌دهد. تطابق عالی بین ground truth و پیش‌بینی EDMD را از مدل کوپمن فوق‌الذکر بر روی داده‌های آزمایشی دیده نشده به‌طور تصادفی نشان می‌دهد.

```
random.seed(76)

plt.figure(figsize=(4, 4))
theta = (np.random.rand(1 ,50))*2*np.pi
x0_test_array = np.stack ((np.cos(theta) ,np.sin(theta)) ,axis=2).reshape(-1, 2)
```

```
for x0_test in x0_test_array:
  xtest_true = odeint(slow_manifold, x0_test.flatten(), t)
  xtest_pred = model.simulate(x0_test ,n_steps=t.size-1)
  xtest_pred = np.vstack([ xtest_true[0] ,xtest_pred])

  plt.plot (xtest_true [: ,0] , xtest_true [: ,1] , 'k')
  plt.plot (xtest_pred [: ,0] , xtest_pred [: ,1] , 'r--')

plt.xlabel ( r'$x_1$')
plt.ylabel ( r'$x_2$')
plt.plot (xtest_true [0 ,0] , xtest_true [0 ,1] , 'k',label='ground truth')
plt.plot (xtest_pred [0 ,0] , xtest_pred [0 ,1] , 'r--',label='EDMD')
plt.legend(loc='best')
```



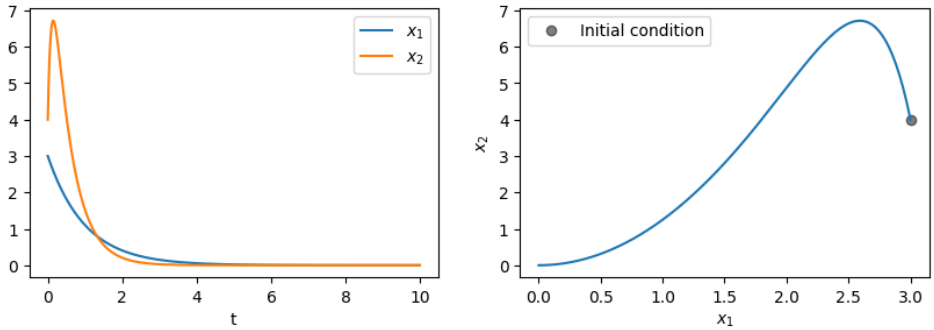
شکل ۳-۳ مسیرهای ground truth و پیش بینی های از EDMD که در PyKoopman با توجه به شرایط اولیه نادیده اجرا شده است.

۳-۲- مثال دوم: Different observables

ما از یک سیستم slow manifold استفاده می‌کنیم که یک سیستم غیرخطی دو بعدی را تشکیل می‌دهد. سپس ما observable را از Identity، چند جمله‌ای، تاخیرهای زمانی، ویژگی‌های فوریه تصادفی (کرنل گاوسی همسان‌گرد) با و بدون حالت‌ها، و observable سفارشی‌سازی شده با تعریف دستی توابع لامبدا می‌سازیم. در نهایت، ما حتی می‌توانیم به طور دلخواه آن موارد observable را با هم ترکیب کنیم.

```
mu = -1
lam = -10

def ode(z, t):
   return [
      mu * z[0],
      lam * (z[1] - z[0] ** 2)
   ]

dt = 0.01
t_train = np.arange(0, 10, dt)
x0_train = [3, 4]
```

```
x_train = odeint(ode, x0_train, t_train)

input_features = ["x1", "x2"]
```



شکل ۴-۳ نمایش متغیرهای حالت

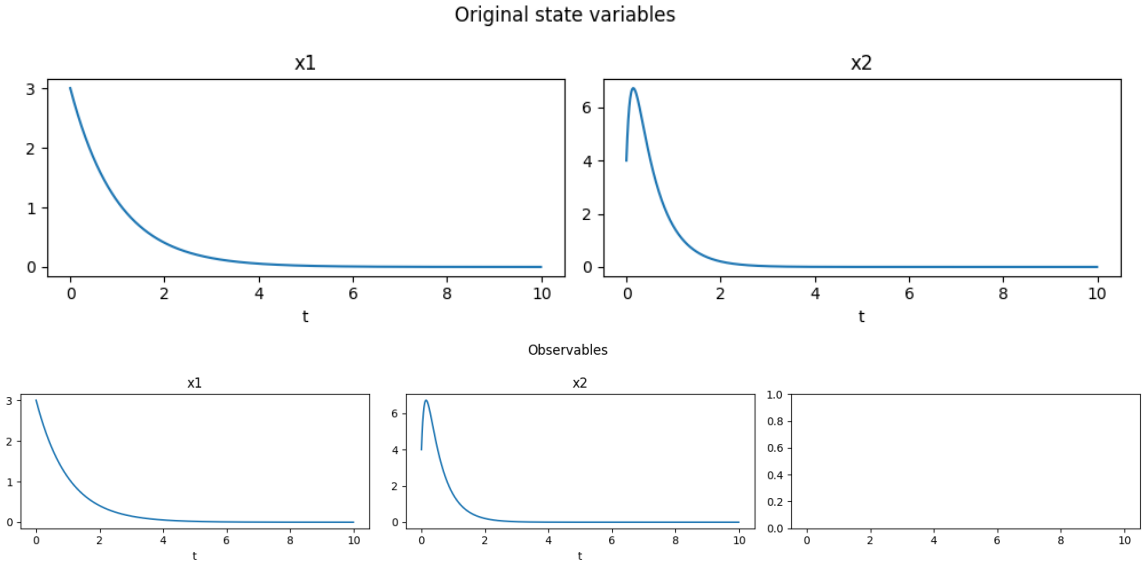
سپس یک تابع برای نمایش خروجی مربوط به observable به همراه متغیر حالت اصلی تعریف می‌کنیم.

```
def plot_observables(observables, x, t, input_features=None, t_delay=None):
    '''Generate plots of state variables before and after being transformed into new observables.'''
    n_features = x.shape[1]
    if input_features is None:
        input_features = [f'x{i}' for i in range(n_features)]
    if t_delay is None:
        t_delay = t

    # Plot input features (state variables)
    fig, axs = plt.subplots(1, n_features, figsize=(n_features * 5, 3))

    for ax, k, feat_name in zip(axs, range(n_features), input_features):
        ax.plot(t, x[:, k])
        ax.set(xlabel='t', title=feat_name)
    fig.suptitle('Original state variables')
    fig.tight_layout()
    # fig.show()

    # Plot output features
    y = observables.fit_transform(x)

    n_output_features = observables.n_output_features_
    feature_names = observables.get_feature_names(input_features)

    n_rows = (n_output_features // 3) + (n_output_features % 3 > 0)
    fig, axs = plt.subplots(n_rows, 3, figsize=(15, 3 * n_rows), sharex=True)

    for ax, k, feat_name in zip(axs.flatten(), range(n_output_features), feature_names):
        ax.plot(t_delay, y[:, k])
        ax.set(xlabel='t', title=feat_name)

    fig.suptitle('Observables')
    fig.tight_layout()
    # fig.show()
    return
```

۳-۲-۱- Identity Observable

Identity Observable به سادگی متغیرهای حالت را بدون تغییر نمایش می‌دهند.

```
obs = pk.observables.Identity()
plot_observables(obs, x_train, t_train, input_features=input_features)
```

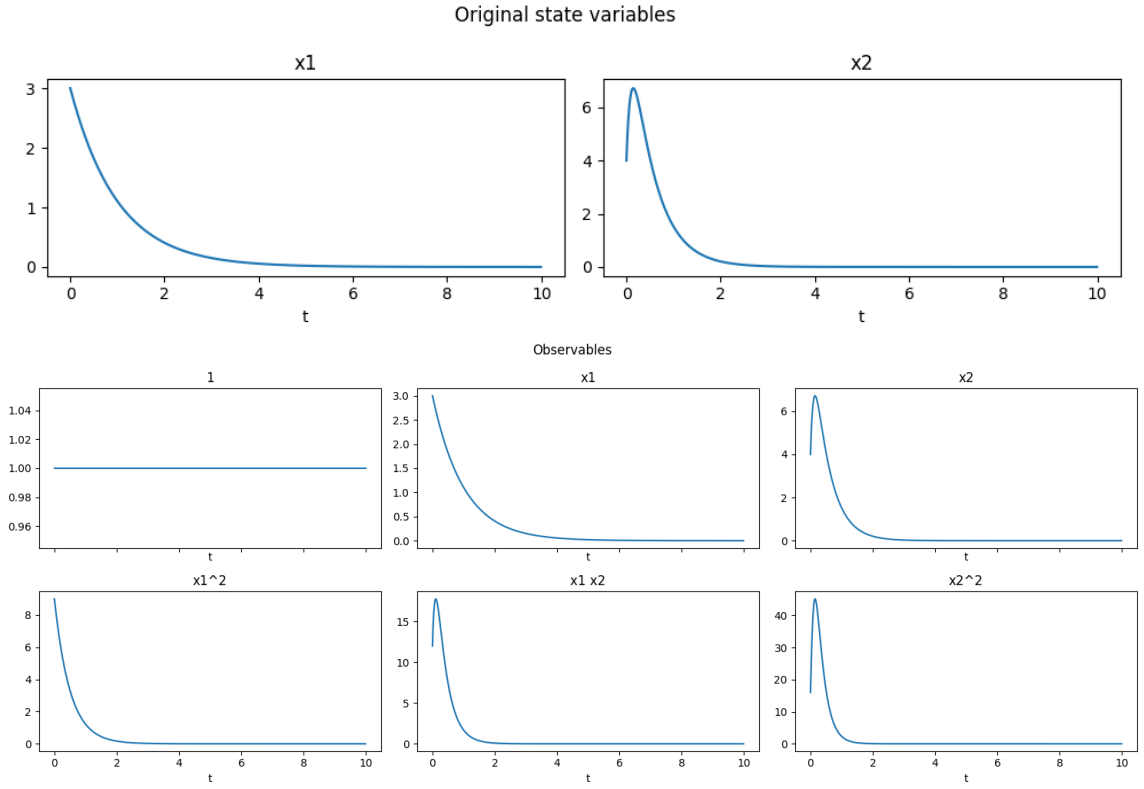


شکل ۵-۳ نمایش متغیرهای حالت اصلی - به کمک Identity observable

۳-۲-۲- چند جمله‌ای

Observables چند جمله‌ای توابع چند جمله‌ای متغیرهای حالت را محاسبه می‌کنند.

```
obs = pk.observables.Polynomial(degree=2)
plot_observables(obs, x_train, t_train, input_features=input_features)
```



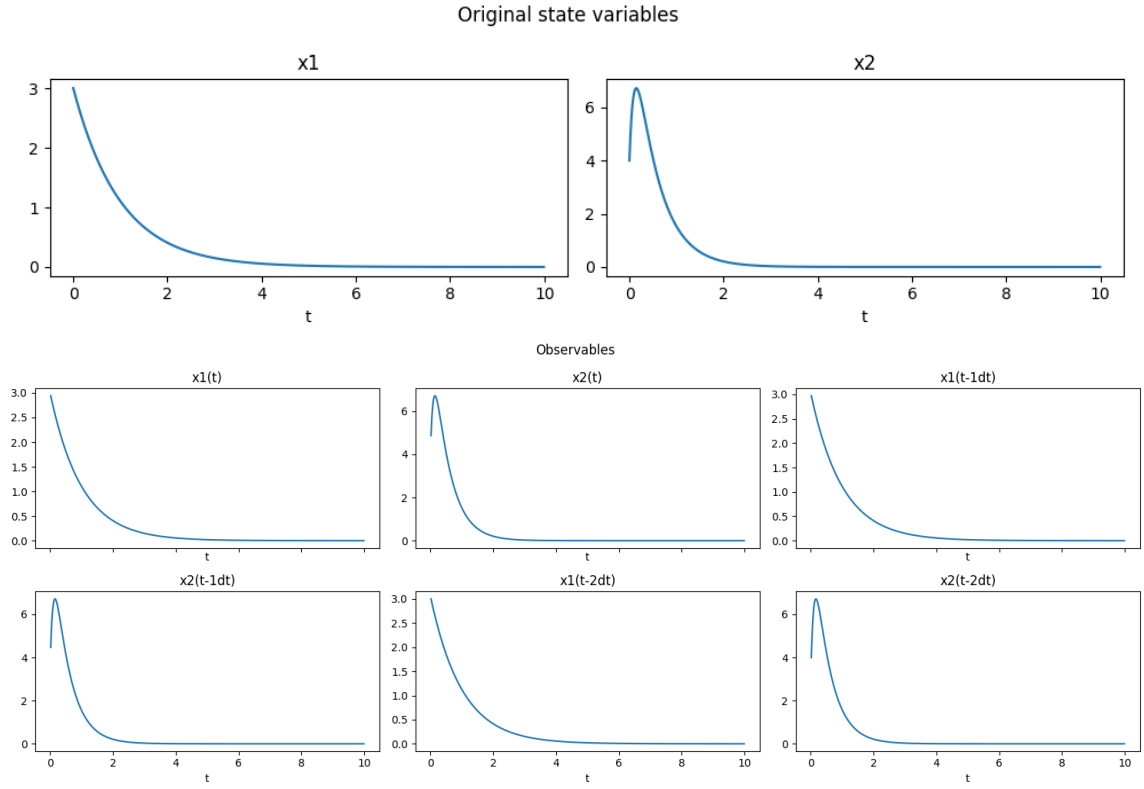
شکل ۶-۳ نمایش متغیرهای حالت اصلی - به کمک observable چندجمله‌ای

۳-۲-۳- تأخیر زمانی

اغلب استفاده از نسخه‌های با تأخیر زمانی متغیرهای حالت مفید است. به عنوان مثال برای مورد delay=1 و n_delays=2 مانند جایگزینی $[x(t), x(t-\Delta t), x(t-2\Delta t)]$ می‌باشد.

برای حالت سیستم با ارزش برداری، تأخیر زمانی بردار در نظر گرفته می‌شود، که در آن مرتبه به هر حالت در آخرین نمونه زمانی داده می‌شود، سپس تمام حالات در نمونه زمانی با تأخیر قبلی و غیره. کلاس TimeDelay برای کمک به ساخت چنین observable طراحی شده است. توجه داشته باشید که چند مشاهدات حالت اول (ردیف‌های x_train) را حذف می‌کند، زیرا این ردیف‌ها تاریخچه زمانی کافی برای ایجاد تأخیرها را ندارند. اطلاعات در واقع از بین نمی‌روند زیرا از آن برای تشکیل نسخه تاخیری متغیر حالت متناظر آن استفاده می‌شود.

```
delay = 1 # dt
n_delays = 2
obs = pk.observables.TimeDelay(delay=delay, n_delays=n_delays)

t_delay = t_train[delay * n_delays:]
plot_observables(obs, x_train, t_train, input_features=input_features, t_delay=t_delay)
```



شکل ۳-۷ نمایش متغیرهای حالت اصلی – به کمک observable تاخیر زمانی – delay=1

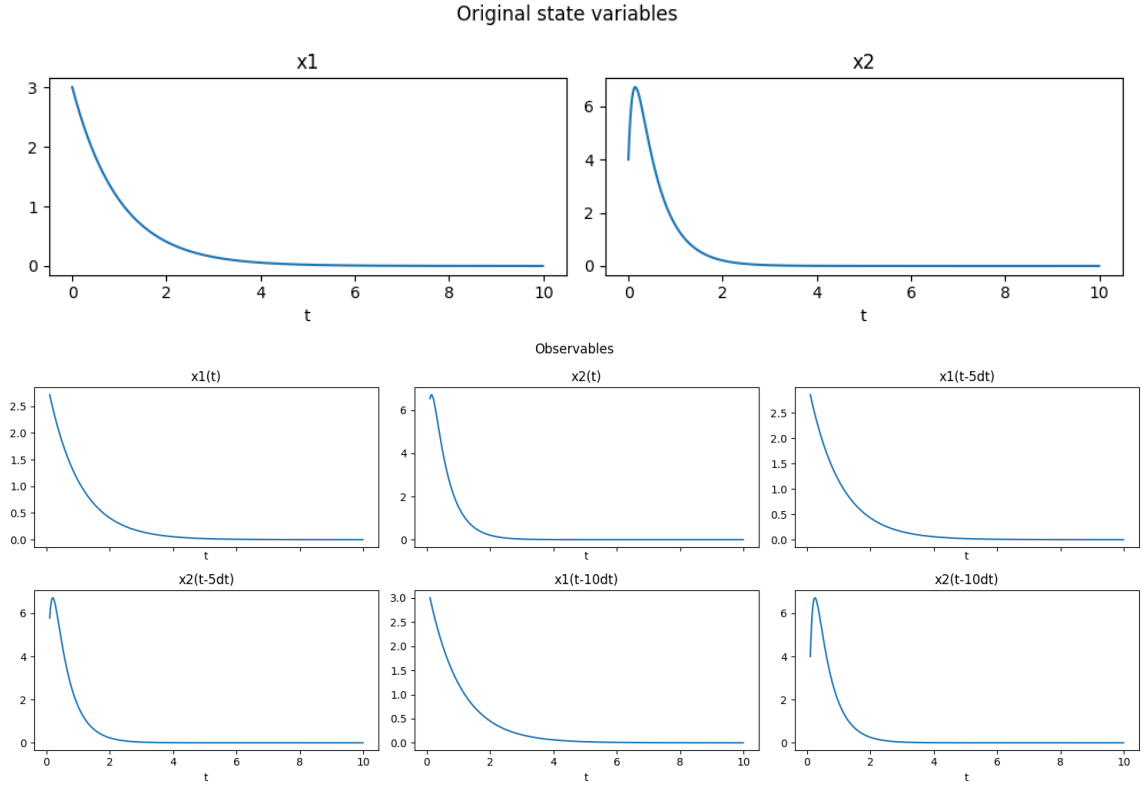
همانطور که انتظار می‌رود، برای چنین موردی در مجموع ۶ ویژگی خروجی داریم. با کمک دستور زیر این موضوع را به همراه تعداد ابعاد داده‌ها و اسامی ویژگی‌ها نمایش می‌دهیم.

```
print("Number of output features: ",obs.n_output_features_)
print("Shape of data: ",obs.fit_transform(x_train).shape)
print("Features Name: ",obs.get_feature_names())
```

```
Number of output features:  6
Shape of data:  (998, 6)
Features NAme:  ['x0(t)', 'x1(t)', 'x0(t-1dt)', 'x1(t-1dt)', 'x0(t-2dt)', 'x1(t-2dt)']
```

شکل ۸-۳ نمایش خروجی‌های کلاس تاخیر زمانی – delay=1

برای بررسی بیشتر delay=5 را نیز بررسی می‌کنیم.



شکل ۹-۳ نمایش متغیرهای حالت اصلی – به کمک observable تاخیر زمانی – delay=5

```
Number of output features:  6
Shape of data:  (990, 6)
Features Name:  ['x0(t)', 'x1(t)', 'x0(t-5dt)', 'x1(t-5dt)', 'x0(t-10dt)', 'x1(t-10dt)']
```

شکل ۱۰-۳ نمایش خروجی‌های کلاس تاخیر زمانی – delay=5

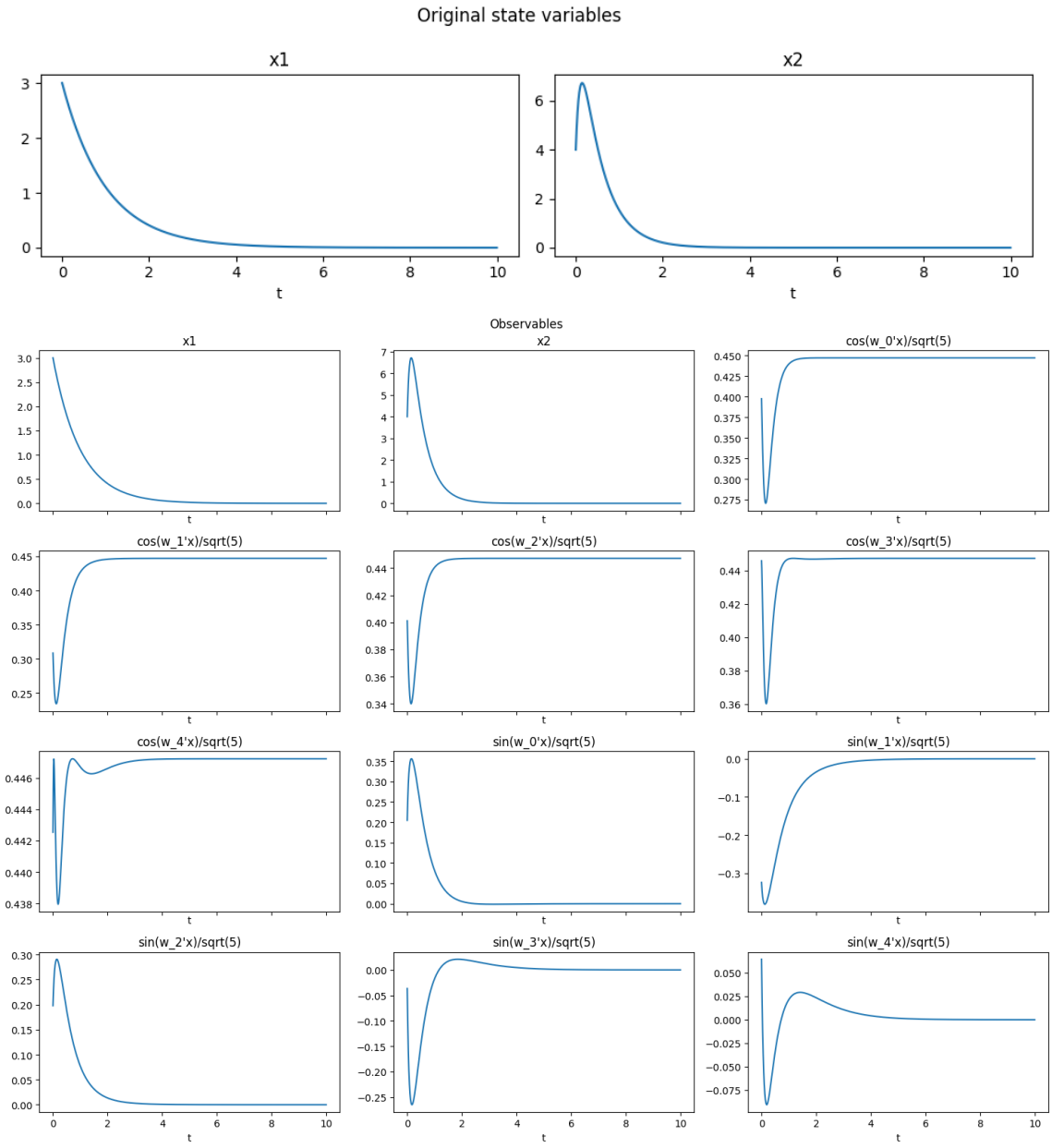
۳-۲-٤- ویژگی‌های فوریه تصادفی

در ریاضیات و پردازش سیگنال، کرنل گاوسی ایزوتروپیک (همسان‌گرد) به تابعی اطلاق می‌شود که برای هموارسازی داده‌ها یا تصویر استفاده می‌شود و در تمام جهات به صورت یکسان عمل می‌کند. این کرنل در بسیاری از زمینه‌ها مانند تحلیل تصاویر و فیلترهای دیجیتال کاربرد دارد. تابع گاوسی ایزوتروپیک معمولاً به شکل زیر تعریف می‌گردد.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

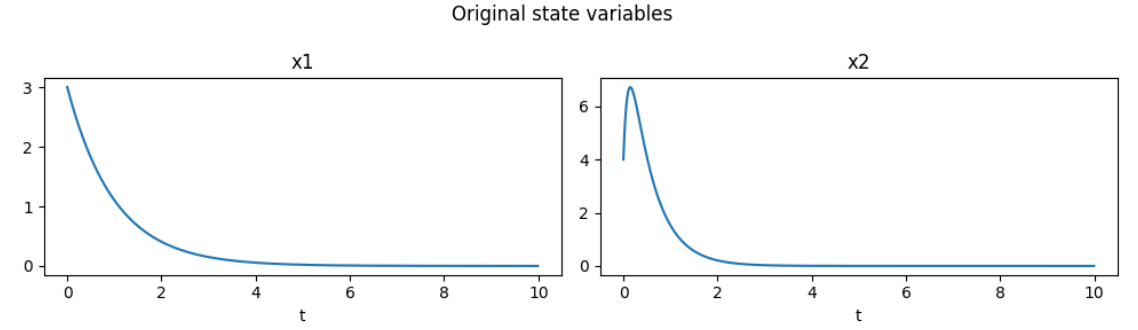
که در آن σ انحراف معیار است که میزان هموارسازی را تعیین می‌کند. این تابع در تمامی جهات به طور یکسان کاهش می‌یابد و از این رو به آن ایزوتروپیک گفته می‌شود.

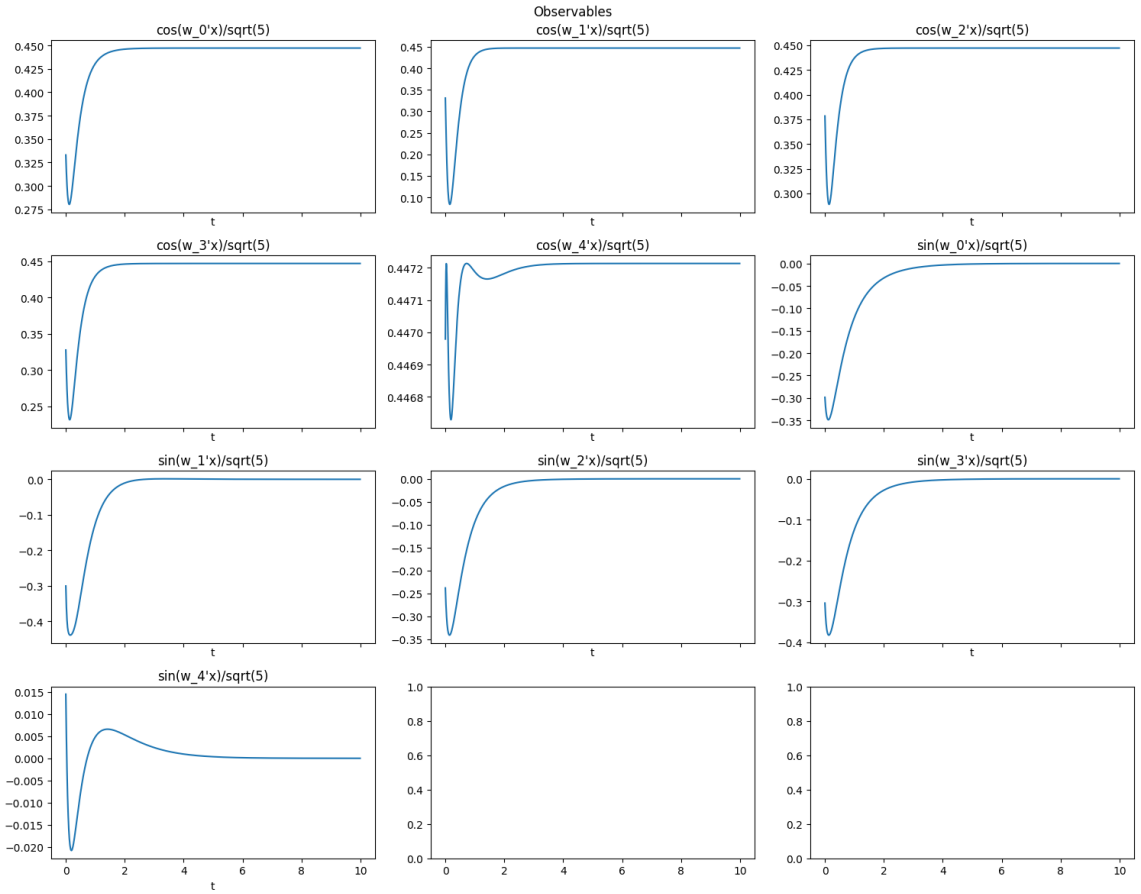
```
# with state
obs = pk.observables.RandomFourierFeatures(include_state=True,gamma=0.01,D=5)
plot_observables(obs, x_train, t_train, input_features=input_features)
```



شکل ۱۱-۳ نمایش متغیرهای حالت اصلی - به کمک observable ویژگی‌های فوریه تصادفی – با متغیر حالت

```
# without state
obs = pk.observables.RandomFourierFeatures(include_state=False,gamma=0.01,D=5)
plot_observables(obs, x_train, t_train, input_features=input_features)
```



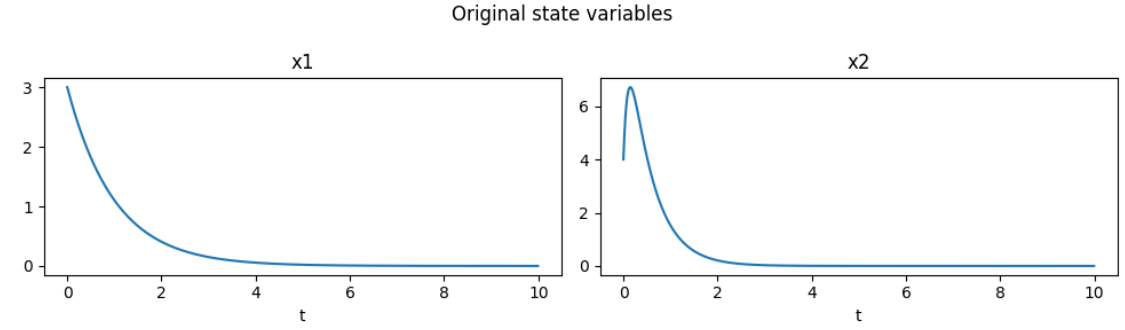


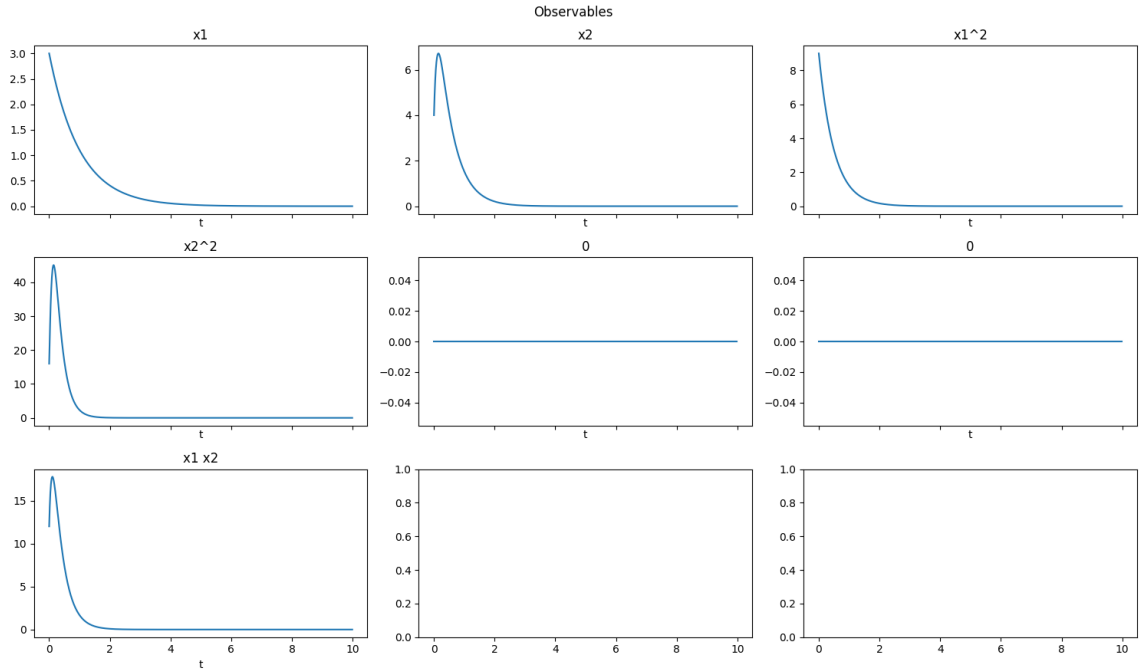
شکل ۱۲-۳ نمایش متغیرهای حالت اصلی - به کمک observable ویژگی‌های فوریه تصادفی – بدون متغیر حالت

۳-۲-۵- Custom Observable

کلاس CustomObservables به فرد اجازه می‌دهد تا به طور مستقیم توابعی را مشخص کند که باید برای متغیرهای حالت اعمال شود. توابع یک متغیر برای هر متغیر حالت و توابع چند متغیره برای هر ترکیب معتبری از متغیرها اعمال خواهد شد. متغیرهای حالت اصلی به طور خودکار گنجانده می‌شوند، حتی زمانی که فرد تابع identity را از مجموعه توابع مشخص شده حذف کند.

```
observables = [lambda x: x ** 2, lambda x: 0 * x, lambda x, y: x * y]
observable_names = [
    lambda s: f"{s}^2",
    lambda s: str(0),
    lambda s, t: f"{s} {t}",
]
obs = pk.observables.CustomObservables(observables, observable_names=observable_names)
plot_observables(obs, x_train, t_train, input_features=input_features)
```

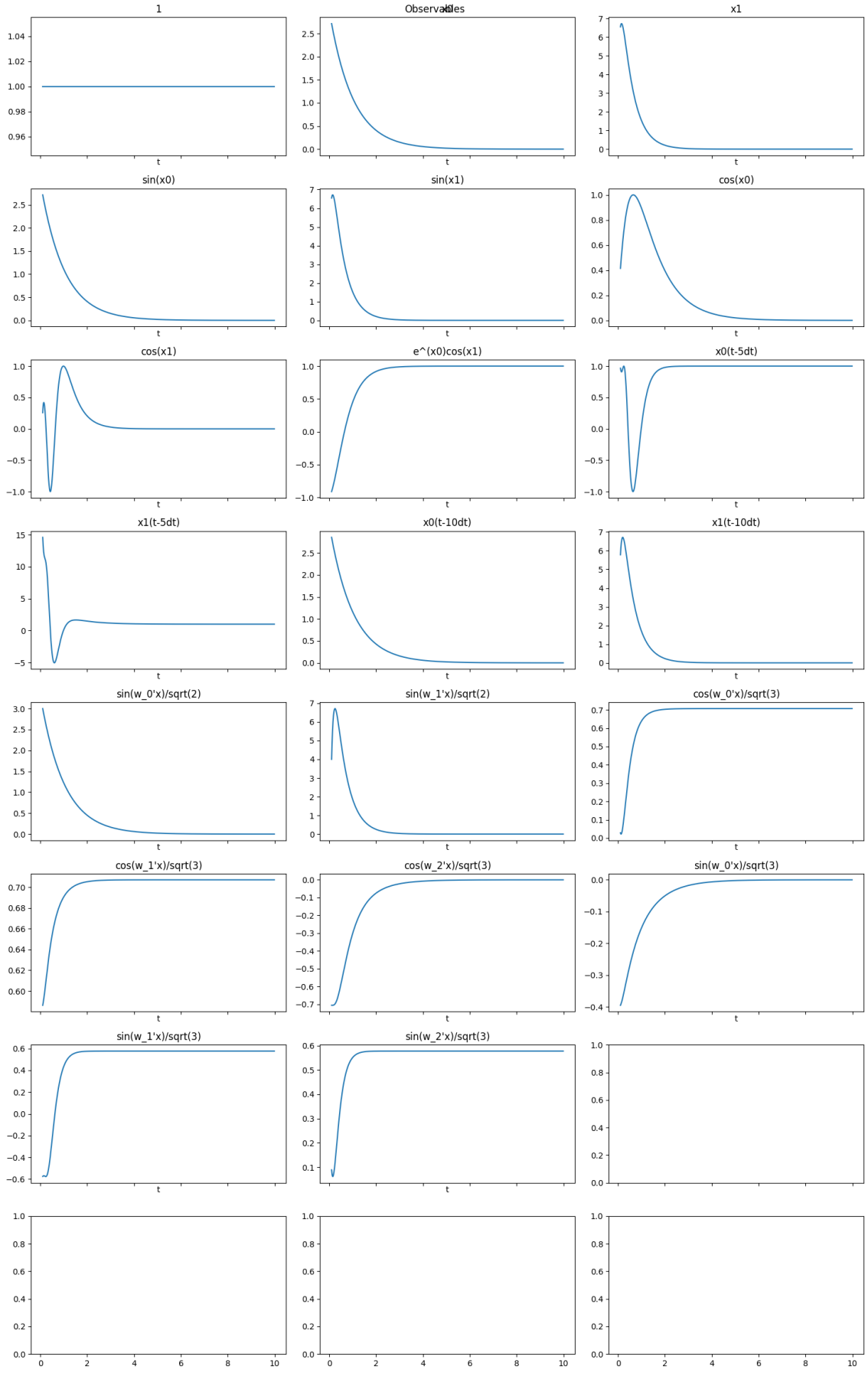




شکل ۱۳-۳ نمایش متغیرهای حالت اصلی - به کمک CustomObservable

۶-۲-۳- **ترکیب** observables

```
#first observable: from Polynomial combinations
ob1 = pk.observables.Polynomial(degree=1)
#second observable: from CustomObservables of univariate functions
observables = [lambda x: np.sin(x), lambda x: np.cos(x), lambda x, y: np.exp(x)*np.cos(y)]
observable_names] =
    lambda s: f"sin({s}),"
    lambda s: f"cos({s}),"
    lambda s, t: f"e^({s})cos({t}),"
[
ob2 = pk.observables.CustomObservables(observables, observable_names=observable_names)
#third observable: a time delay observable
delay = 5 # dt
n_delays = 2
t_delay = t_train[delay * n_delays:]
ob3 = pk.observables.TimeDelay(delay=delay, n_delays=n_delays)
#fourth observable: random fourier feature without state
ob4 = pk.observables.RandomFourierFeatures(include_state=False,gamma=0.01,D=2)


#fifth observable: random fourier feature with state
ob5 = pk.observables.RandomFourierFeatures(include_state=True,gamma=0.1,D=3)
obs = ob1 + ob2 + ob3 + ob4 + ob5
plot_observables(obs, x_train, t_train, input_features=input_features, t_delay=t_delay)
```



شکل ۱۴-۳ نمایش متغیرهای حالت به کمک ترکیب observables

```
print("Number of output features: ",obs.n_output_features_)
print("Shape of data: ",obs.fit_transform(x_train).shape)
```

```
Number of output features:  24
Shape of data:  (990, 24)
```

شکل ۱۵-۳ نمایش خروجی‌های ترکیب کلاس‌ها

۳-۳- **مثال سوم: Neural Network DMD on Slow Manifold**

در اینجا ما در مورد نحوه استفاده از NN-DMD در PyKoopman (pykoopman.regressor.NNDMD) صحبت خواهیم کرد. ما در ادامه سیستم slow manifold را در نظر گرفته و نحوه استفاده مستقیم از رگرسیور NNDMD را به عنوان مدل Koopman نشان خواهیم داد.

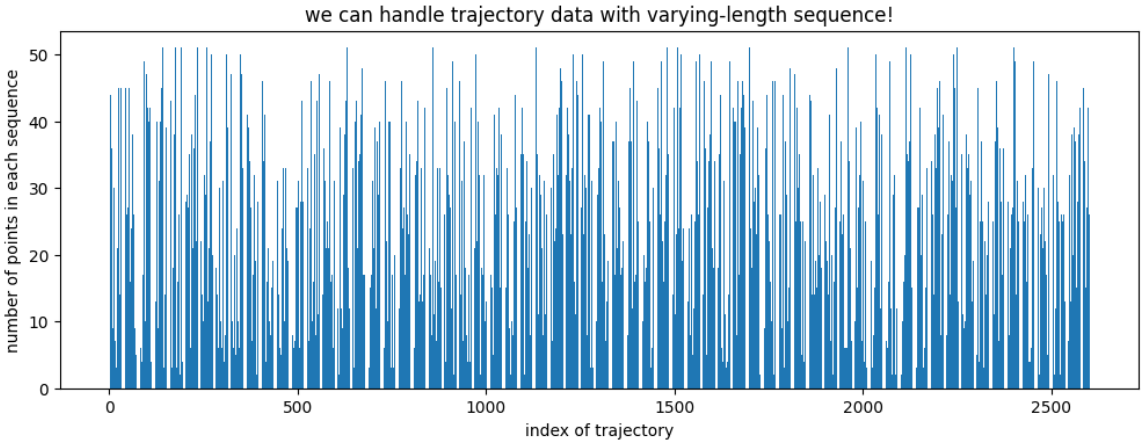
```
nonlinear_sys = slow_manifold(mu=-0.1, la=-1.0, dt=0.1)
```

در اینجا یک شبکه $[-2,2]$ برای نمونه‌برداری از شرایط اولیه ایجاد می‌کنم. از ۵۱ نقطه در هر جهت نمونه برداری می‌کنیم.

```
n_pts = 51
xmin = ymin = -2
xmax = ymax = +2
xx, yy = np.meshgrid(np.linspace(xmin, xmax, n_pts), np.linspace(ymin, ymax, n_pts))
Xdat = np.vstack((xx.flatten(), yy.flatten()))
```

سپس، دنباله‌های زیادی با طول‌های مختلف ایجاد شده است.

```
max_n_int = 51

traj_list = []
for i in range(Xdat.shape[1]):
    X, Y = nonlinear_sys.collect_data_discrete(Xdat[:,[i]], np.random.randint(1, max_n_int))
    tmp = np.hstack([X, Y[:,-1:]]).T
    traj_list.append(tmp)
```



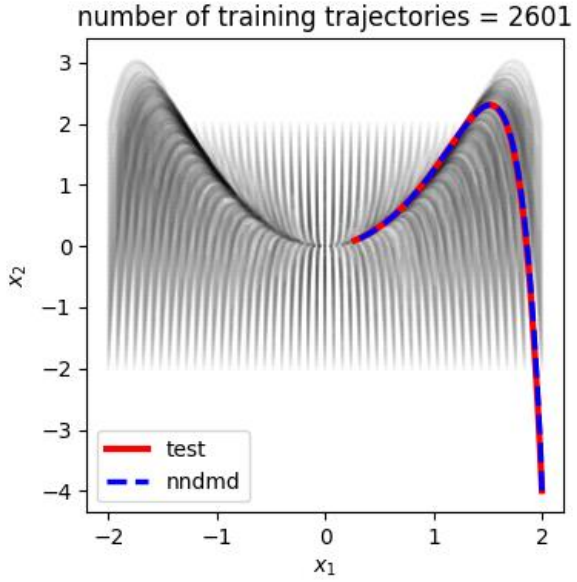
شکل ۱۶-۳ دامنه‌ی مسیرهای ایجاد شده

اکنون، NNDMD به عنوان مدل انتخاب شده است. توجه داشته باشید که ما look_forward=50 را می‌نامیم، به این معنی که مدل ما حداکثر می‌تواند توالی داده‌ها را با ۵۱ نقطه کنترل کند. اگر دنباله داده حاوی بیش از ۵۱ نقطه باشد، آنگاه می‌تواند look_forward=50 یا حتی تعداد کمتری را تنظیم کند. اما نمی‌توان look_forward را بزرگتر از length_of_longest_sequence تنظیم کرد.

```
look_forward = 50
dlk_regressor = pk.regression.NNDMD(dt=nonlinear_sys.dt, look_forward=look_forward,
                        config_encoder=dict(input_size=2,
                                hidden_sizes=[32] * 3,
                                output_size=3,
                                activations="swish"),
                        config_decoder=dict(input_size=3, hidden_sizes=[],
                                output_size=2, activations="linear"),
                        batch_size=512, lbfgs=True,
                        normalize=True, normalize_mode='equal',
                        normalize_std_factor=1.0,
                        trainer_kwargs=dict(max_epochs=3))
dlk_regressor.fit(traj_list)
```

حال برای مسیرهای دیده نشده مدل را امتحان می‌کنیم.

```
x0 = np.array([2, -4])  #np.array([2, -4])
T = 20
t = np.arange(0, T, nonlinear_sys.dt)
Xtest = nonlinear_sys.simulate(x0[:, np.newaxis], len(t)-1).T
Xtest = np.vstack([x0[np.newaxis, :], Xtest])

Xkoop_nn = dlk_regressor.simulate(x0[np.newaxis, :], n_steps=len(t)-1)
```



شکل ۱۷-۳ مسیرهای ground truth و پیش بینی های از NNDMD که در PyKoopman با توجه به شرایط اولیه نادیده اجرا شده است.

۳-٤- مثال چهارم: Extended DMD for Van Der Pol System

در این مثال، یک مدل کوپمن خطی را با استفاده از EDMD برای یک سیستم غیرخطی و دینامیکی آموزش خواهیم داد. این رویکرد برای سیستم زمان گسسته Van der Pol در زمان معکوس نشان داده شده است:

$x_{k+1} = x_k - y_k dt$, $y_{k+1} = y_k + (x_k - y_k + x_k^2 y_k)dt$

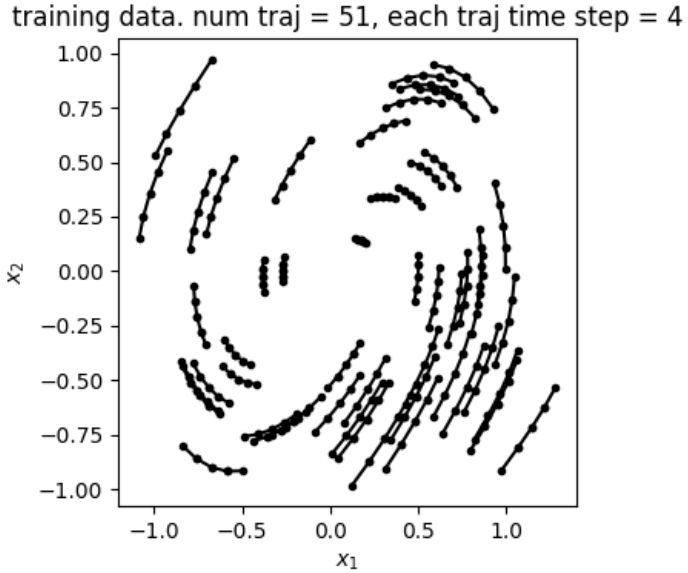
که مقدار $dt = 0.1$ می‌باشد.در ابتدا به مانند قبل بسته‌ی مورد نظر نصب شده است و کتابخانه‌های مور نیازز فراخوانی می‌گردد.

```
import pykoopman as pk
import numpy as np
```

```
import numpy.random as rnd
np.random.seed(42)  # for reproducibility
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches


import warnings
warnings.filterwarnings('ignore')


from pykoopman.common import rev_dvdp  # discrete-time, reverse-time van der Pol
```

داده‌های آموزشی شامل یک جفت داده‌های نمونه برداری شده، که از ۵۱ شرایط اولیه تصادفی توزیع شده یکنواخت گرفته شده است.

```
n_states = 2  # Number of states
dT = 0.1      # Timestep
n_traj = 51   # Number of trajectories
n_int = 4     # Integration length

# Uniform distribution of initial conditions
x = xE = 2*rnd.random([n_states, n_traj])-1

# Init
X = np.zeros((n_states, n_int*n_traj))
Y = np.zeros((n_states, n_int*n_traj))

# Integrate
for step in range(n_int):
    y = rev_dvdp(0, x, 0, dT)
    X[:, (step)*n_traj:(step+1)*n_traj] = x
    Y[:, (step)*n_traj:(step+1)*n_traj] = y
    x = y
```



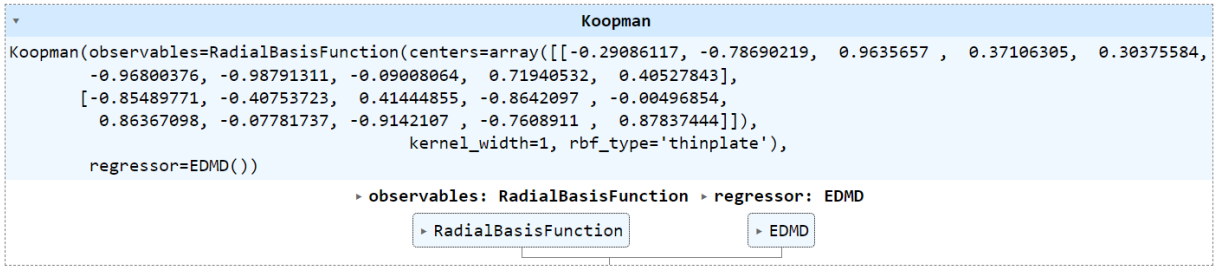
شکل ۱۸-۳ داده‌های اندازه‌گیری با استفاده از van der pol شبیه‌سازی شده است.

توابع پایه شعاعی صفحه نازک (RBFs) نوعی از RBF هستند که در درون‌یابی و هموارسازی فضایی استفاده می‌شوند. آنها به ویژه برای کاربردهایی که به یک سطح صاف و پیوسته نیاز است، مانند سیستم‌های اطلاعات جغرافیایی (GIS)، پردازش تصویر، و بازسازی سطح، مناسب هستند. صفحه نازک RBF به صورت تعریف شده است:

$$\phi(r) = r^2 \log(r)$$

که r فاصله اقلیدسی بین نقاط است. این عملکرد به دلیل صاف بودن و حداقل انرژی خمشی آن شناخته شده است و آن را به گزینه ای بهینه برای درونیابی سطوح در ابعاد دو یا بالاتر تبدیل می کند. به عنوان توابع پایه، توابع پایه شعاعی صفحه نازک را انتخاب می‌کنیم.

```
EDMD = pk.regression.EDMD()
centers = np.random.uniform(-1,1,(2,10))
RBF = pk.observables.RadialBasisFunction(
    rbf_type="thinplate",
    n_centers=centers.shape[1],
    centers=centers,
    kernel_width=1,
    polyharmonic_coeff=1.0,
    include_state=True,
)

model = pk.Koopman(observables=RBF, regressor=EDMD)
model.fit(X.T, y=Y.T)
```

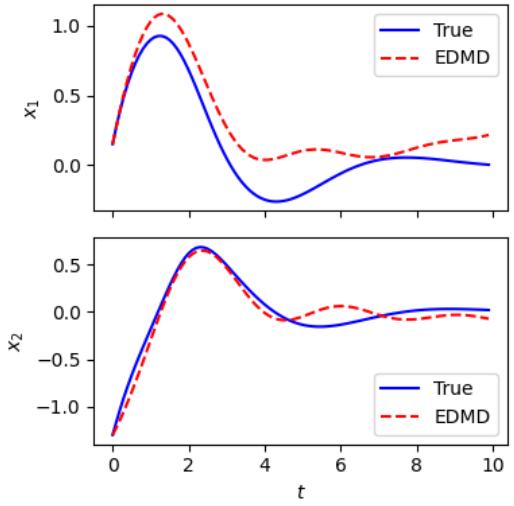


شکل ۱۹-۳ عملگر کوپمن

ابتدا، ما از مدل EDMD آموزش دیده شده برای پیش‌بینی ارزیابی برای یک شرایط اولیه استفاده می‌کنیم. تبدیل معکوس، یعنی از observables به حالت، با استفاده از رگرسیون حداقل مربعات در رگرسیون کوپمن تخمین زده می‌شود.

```
x0 = np.array([۱.۳- ,۰.۱۵])
T = 10
t = np.arange(0, T, dT)


 #Integrate
Xtrue = np.zeros((len(t), n_states))
Xtrue[0, :] = x0
for step in range(len(t)-1):
    y = rev_dvdp(0, Xtrue[step, :][:, np.newaxis], 0, dT)
    Xtrue[step+1, :] = y.ravel()
```

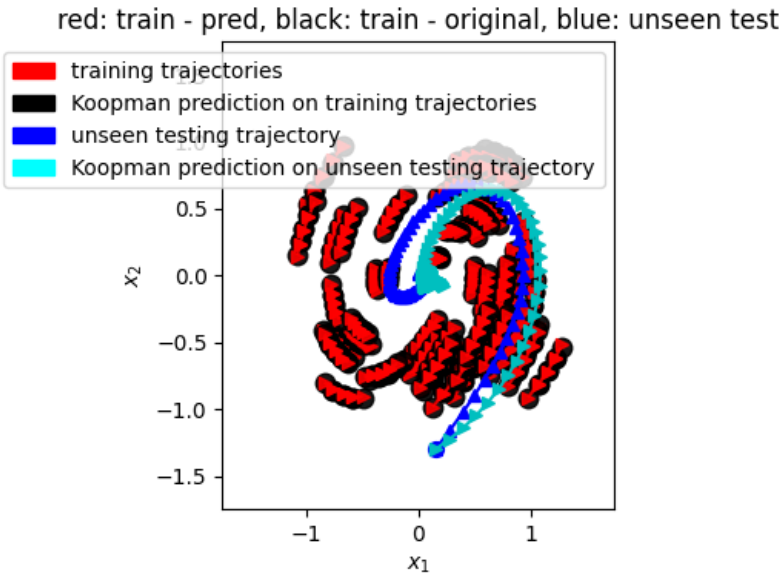
```
#Simulate (multi-step prediction) Koopman model
Xkoop = model.simulate(x0, n_steps=len(t)-1)
Xkoop = np.vstack([x0[np.newaxis,:], Xkoop])
```



شکل ۲۰-۳ مسیرهای ground truth و پیش بینی های از EDMD که در PyKoopman با توجه به شرایط اولیه نادیده اجرا شده است.

در آخر نیز، عملکرد پیش‌بینی داده‌های آموزشی را نشان خواهیم داد.

```
# Init
Xk = np.zeros((n_states, n_int*n_traj))
Yk = np.zeros((n_states, n_int*n_traj))
# 1-step prediction using Koopman model
x = xE.T
for k in range(n_int):
    # print(k)
    yT = model.predict(x)
    y = yT.T
    Xk[:, k*n_traj:(k+1)*n_traj] = x.T
    Yk[:, k*n_traj:(k+1)*n_traj] = y
    x = y.T
```



شکل ۲۱-۳ تخمین رفتار آینده سیستم براساس ورودی‌های جدید و مدل یادگرفته شده.

۳-۵- **مثال پنجم: Extended DMD with control for Van der Pol oscillator**

اسیلاتور تحت نیرو کلاسیک Van der Pol با دینامیک ارائه شده توسط زیر را در این بخش در نظر می‌گیریم.

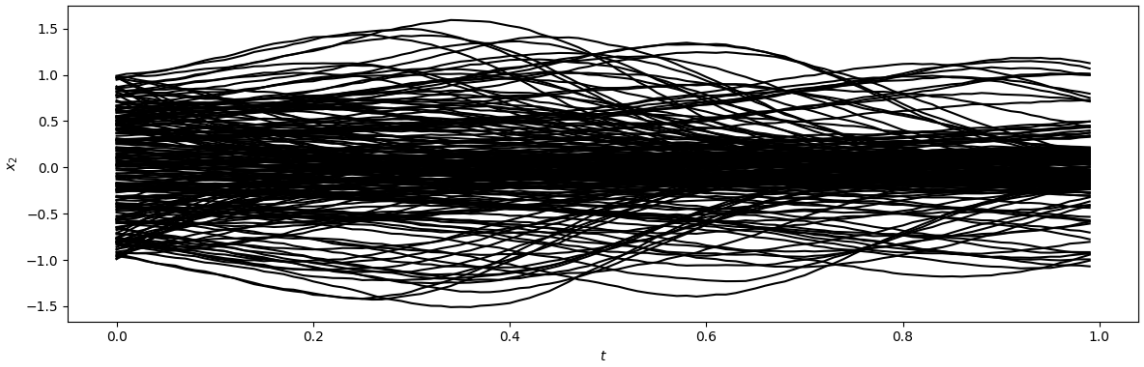
$$\dot{x}_1 = 2x_2 u$$
$$\dot{x}_2 = -0.8x_1 + 2x_2 - 10x_1^2 x_2 + u$$

در ابتدا کتابخانه‌های مورد نیاز را فراخوانی می‌کنیم.

```
%matplotlib inline
import pykoopman as pk
from pykoopman.common.examples import vdp_osc, rk4, square_wave  # required for example system
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd
np.random.seed(76)  # for reproducibility

import warnings
warnings.filterwarnings('ignore')
```

یک مجموعه داده آموزشی متشکل از ۲۰۰ مسیر ایجاد می‌شود، هر مسیر برای ۱۰۰۰ گام ادغام می‌شود و با یک تحریک تصادفی در محدوده $[-1,1]$ تحریک می‌شود. هر مسیر در یک شرایط اولیه تصادفی در جعبه واحد $[-1,1]$ شروع می شود.

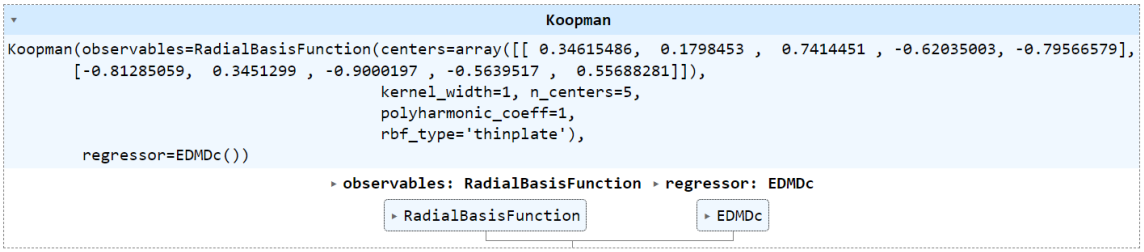
```
n_states = 2 # Number of states
n_inputs = 1 # Number of control inputs
dT = 0.01    # Timestep
n_traj = 200  # Number of trajectories
n_int = 1000  # Integration length

# Time vector
t = np.arange(0, n_int*dT, dT)

# Uniform random distributed forcing in [-1, 1]
u = 2*rnd.random([n_int, n_traj])-1

# Uniform distribution of initial conditions
x = 2*rnd.random([n_states, n_traj])-1

# Init
X = np.zeros((n_states, n_int*n_traj))
Y = np.zeros((n_states, n_int*n_traj))
U = np.zeros((n_inputs, n_int*n_traj))

# Integrate
for step in range(n_int):
    y = rk4(0, x, u[step, :], dT, vdp_osc)
    X[:, (step)*n_traj:(step+1)*n_traj] = x
    Y[:, (step)*n_traj:(step+1)*n_traj] = y
    U[:, (step)*n_traj:(step+1)*n_traj] = u[step, :]
    x = y
```



شکل ۲۲-۳ مسیرهای ایجاد شده برای سیستم

Observables (یا توابع بالابر) برای مدل کوپمن به‌صورت خود حالت ($\psi_1 = x_1 , \psi_2 = x_2$) (با تنظیم include_states=True در زیر که پیش‌فرض نیز هست) و توابع پایه شعاعی صفحه نازک با مراکز به‌طور تصادفی انتخاب می‌شوند.

```
EDMDc = pk.regression.EDMDc()
centers = np.random.uniform(-1,1,(2,5))
RBF = pk.observables.RadialBasisFunction(
    rbf_type="thinplate",
    n_centers=centers.shape[1],
    centers=centers,
    kernel_width=1,
    polyharmonic_coeff=1,
    include_state=True,
)

model = pk.Koopman(observables=RBF, regressor=EDMDc)
model.fit(X.T, y=Y.T, u=U.T)
```

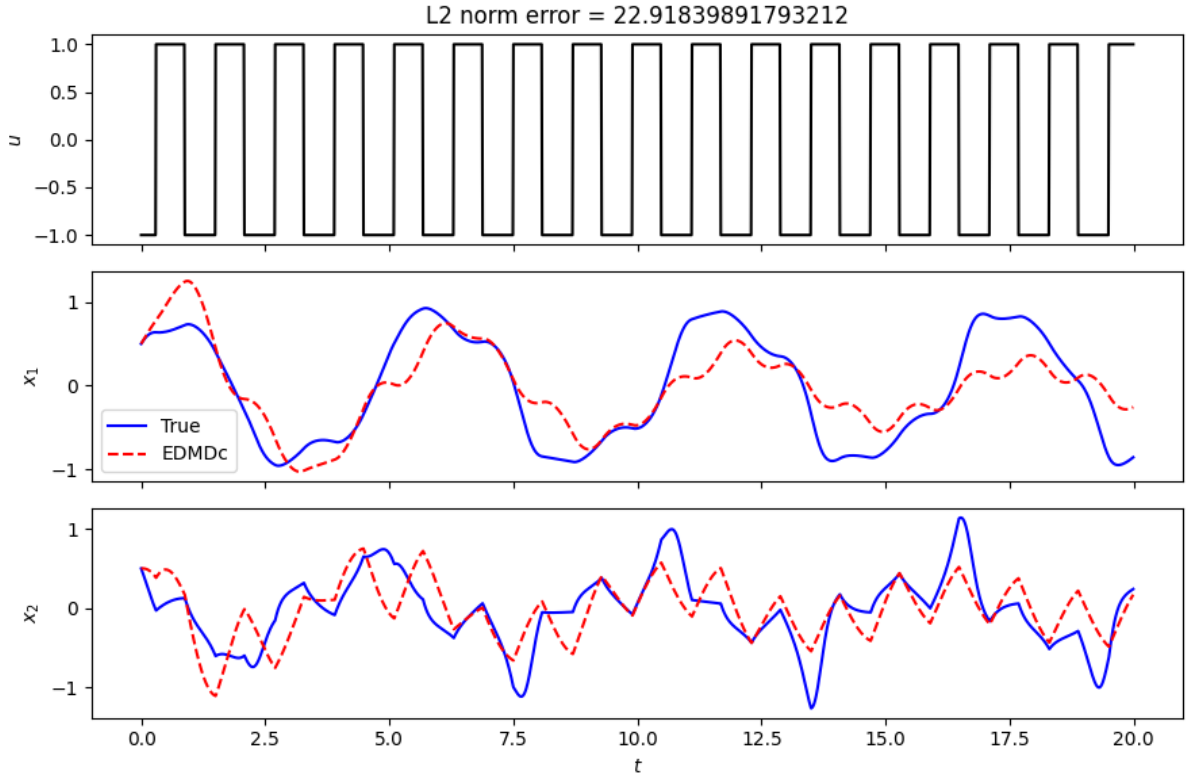


شکل ۲۳-۳ عملگر کوپمن

در ادامه، از مدل آموزش‌دیده برای انجام یک پیش‌بینی چند مرحله‌ای از یک شرایط اولیه معین استفاده می‌شود. هنگام ادغام سیستم غیرخطی، پیش‌بینی با مسیر واقعی مقایسه می‌شود.

```
n_int = 2000  # Integration length
t = np.arange(0, n_int*dT, dT)
u = np.array([-square_wave(step//2+1) for step in range(n_int)])
x = np.array([0.5, 0.5])

# Integrate nonlinear system
Xtrue = np.zeros((n_states, n_int))
Xtrue[:, 0] = x
for step in range(1, n_int, 1):
```

```
    y = rk4(0, Xtrue[:, step-1].reshape(n_states,1), u[np.newaxis, step-1], dT, vdp_osc)
    Xtrue[:, step] = y.reshape(n_states,)
# Multi-step prediction with Koopman/EDMDc model
Xkoop = model.simulate(x, u[:, np.newaxis], n_steps=n_int-1)
Xkoop = np.vstack([x[np.newaxis,:], Xkoop]) # add initial condition to simulated data for comparison
below
```



شکل ۲۴-۳ مقایسه پیش‌بینی با مقدار واقعی

سیستم افزایش بعد یافته نیز دارای ابعاد به صورت زیر است.

$$Z^{+} = Az + Bu$$
$$x = Cz$$

```
print('Shape of:')
print('A: ',model.A.shape,)
print('B: ',model.B.shape,)
print('C: ',model.C.shape,)
print('W: ',model.W.shape,)
```

```
Shape of:
A:  (7, 7)
B:  (7, 1)
C:  (2, 7)
W:  (2, 7)
```

شکل ۲۵-۳ نمایش ابعاد سیستم افزایش بعد یافته

منابع

[۱] S. Pan, E. Kaiser, B. M. de Silva, J. N. Kutz, and S. L. Brunton, "PyKoopman: a python package for data-driven approximation of the Koopman operator," *arXiv preprint arXiv:2306.12962*, 2023.