نگاهی بر
نام‌پای و سای‌پای

نوشته‌ی ایلای برسرت

ترجمه: امیرحسین نمدچی

نگاهی بر نام‌پای و سای‌پای

نوشته‌ی ایلای برِسِرت

ترجمه‌ی امیرحسین نمدچی

Amir.Hossein.Namadchi@Gmail.com
linkedin.com/in/amir-hossein-namadchi
facebook.com/amirhossein.namadchi
twitter.com/AmirNamadchi

فهرست

آغاز سخن

پایتون یک زبانِ برنامه‌نویسیِ سطحِ بالا، بسیار انعطاف پذیر و خوانا است. این ویژگی‌ها، پایتون را زبانی ایده‌آل برای یادگیری و استفاده می‌سازد. در زمینه‌ی علوم، پژوهش و توسعه بسته‌های اندکی برای گسترش و رسیدن به اهدافِ موردنظر در کمترین زمان ممکن وجود دارند. در میانِ بهترینِ این بسته‌ها، *SciPy* و *NumPy* خودنمایی می‌کنند. این کتاب، نگاهی کوتاه بر ابزارهایِ گوناگونِ موجود در این دو بسته‌ی علمی خواهد داشت که می‌تواند سرآغازی برای بهره‌جویی از آنها در پروژه‌های پژوهشی خواننده باشد.

NumPy و *SciPy* بسته‌های اصلی و بنیادی پایتون برای کار با آرایه‌های عددی و تحلیل پیشرفته‌ی داده‌ها به شمار می‌روند. از این رو، شناختِ ابزارهایِ موجود در این دو بسته و چگونگی بهره‌جویی از آنها، زندگیِ برنامه‌نویس را لذت‌بخش‌تر می‌کند. این کتاب، کاربردِ بسته‌های مزبور را پوشش می‌دهد؛ از خلقِ یک آرایه‌ی ساده گرفته تا یادگیریِ ماشین.

مخاطب‌ها

اگر آشنایی ابتدایی (و بالاتر) با زبان برنامه‌نویسی پایتون دارید، این کتاب مناسب شما است. اگرچه ابزارهای موجود در *SciPy* و *NumPy* نسبتن پیشرفته هستند، استفاده از آنها ساده بوده و می‌تواند برنامه‌نویسان تازه‌کار را نیز خشنود کند.

محتویات کتاب

این کتاب به موارد پایه‌ای و اصلی *NumPy* و *SciPy* اشاره می‌کند. فصلِ نخست، به معرفی بسته‌های *SciPy* و *NumPy*، چگونگی دسترسی و نصب آنها بر روی رایانه‌تان می‌پردازد. فصل دوم، مواردِ ابتدایی *NumPy* مانند تولید آرایه را در بر می‌گیرد. فصل سوم که بیشترین بخش کتاب را تشکیل می‌دهد، شامل مثال‌هایی از ابزارهایِ پرشمارِ بسته‌ی *SciPy* است. این بخش دربرگیرنده‌ی مباحث و مثال‌هایی پیرامونِ تابع‌اولیه‌گیری، بهینه‌سازی، درون‌یابی و... است. فصل پنج در مورد دو بسته‌ی شناخته شده از *scikit* سخن می‌گوید؛ یعنی *scikit-image* و *scikit-learn*. این فصل مباحث پیشرفته‌ی اضافه‌تری را نیز فراهم می‌آورد

که قابل به‌کارگیری در مسائل دنیای واقعی هستند. در فصل آخر، در مورد گام‌های بعدی پیرامونِ مباحثِ پیشرفته‌تر سخن خـواهیم گفت.

پیشگفتار مترجم

کتاب کنونی ترجمه‌ای از *SciPy and NumPy: An Overview for Developers* نوشته‌ی *ایلای برسِرت* است که در سال ۲۰۱۲ توسط شرکت اورایلی منتشر شد. این کتاب به دو بسته‌ی فراگیر و پرکاربرد پایتون در محاسبات علمی، یعنی ***SciPy*** و ***NumPy*** می‌پردازد. بد نیست بدانید که زبان برنامه‌نویسی پایتون در ابتدا برای محاسبات عددی و علمی طراحی نشده بود؛ اما به تدریج توجه جامعه‌ی مهندسین و دانشمندان را به خود جلب کرد به گونه‌ای که در سال ۱۹۹۵ گروه ***matrix-sig*** با هدف تولید بسته‌ای برای محاسبات آرایه‌ای در پایتون، پایه‌ریزی شد. در میان اعضای این گروه، *خودو فَن روسوم* نویسنده‌ی زبان پایتون نیز حضور داشت تا محاسبات آرایه‌ای را هر چه بیشتر به زبان پایتون نزدیک کند. پس از گذشت سال‌ها و تکامل تدریجی بسته‌ی محاسبات آرایه‌ای پایتون، در سال ۲۰۰۵ *تراویس اولیفنت* با هدف یکپارچه‌سازی دستاوردهای گروه، اولین نسخه‌ی ***NumPy*** را در سال ۲۰۰۶ منتشر کرد. از سوی دیگر، نسخه‌ی نخستین ***SciPy*** در سال ۲۰۰۱ منتشر شد. ***SciPy*** که بر پایه‌ی آرایه‌های ***NumPy*** استوار است، مجموعه‌ای کامل و قدرتمند برای حل مسائل مربوط به بهینه‌سازی، تابع‌اولیه‌گیری، جبر خطی، درون‌یابی، حل معادلات دیفرانسیل، تحلیل سریع فوریه و... می‌باشد.

این کتاب به مواردی ساده و ابتدایی از دو بسته‌ی مزبور اشاره می‌کند و می‌تواند نقطه‌ی آغازی برای آشنایی با دنیای محاسبات عددی و علمی در پایتون به شمار رود. متن‌باز بودن و رایگان بودن پایتون و بسته‌های آن به همراه سادگی شگفت‌انگیز و نزدیکی به زبان انسان، موجب تمایل بسیاری از دانشمندان و پژوهش‌گران به این زبان شده است. این رشد چشم‌گیر تقریبن در همه‌ی زمینه‌های پژوهشی و مهندسی قابل مشاهده است. یک جستجوی ساده در گوگل، این حقیقت را آشکار خواهد کرد.

با توجه به تاریخِ انتشارِ کتابِ اصلی (سال ۲۰۱۲)، همه‌ی کدهای موجود در آن با نسخه‌ی ۲ پایتون نوشته شده بودند. حال آنکه سایت پایتون، سال ۲۰۲۰ را پایان عمر این نسخه معرفی کرده و کاربران را تشویق به استفاده از نسخه‌های جدیدتر می‌کند. نسخه‌ی ۳,۵ پایتون در حال حاضر نسخه‌ی پیشنهادی سایت پایتون می‌باشد. از این رو، همه‌ی کدهای کتاب در محیط پایتون ۳,۵ دوباره ارزیابی و تبدیل شده‌اند.

اگرچه نویسنده‌ی کتاب شماری از توزیع‌ها را برای بهره‌جویی از پایتون و بسته‌هایش معرفی کرده است، برای ویندوز، پیشنهاد مترجم استفاده از توزیع *Anaconda* است. توجه داشته باشید که با نصب این بسته، زبان برنامه‌نویسی پایتون و بسیاری از بسته‌های پرکاربرد از جمله ***NumPy*** و ***SciPy*** بطور همزمان بر روی رایانه‌ی شما نصب خواهد شد.

این کتاب کاملن رایگان بوده و استفاده‌ی تجاری از آن غیراخلاقی است. شما می‌توانید پیشنهادات و نظرات ارزنده‌ی خود را به آدرس *Amir.Hossein.Namadchi@Gmail.com* ارسال فرمایید.

امیرحسین نمدچی – پاییز ۹۵

فصل یکم

پیشگفتار

اگر ویژگی‌هایی مانند قابلِ حمل بودن، انعطاف‌پذیری، ساختارِ نحوی، سَبک و قابلیتِ توسعه در نظر گرفته شوند، پایتون یک زبانِ توان‌مندِ برنامه‌نویسی خواهد بود. این زبان توسط *خودو فَن روسوم* و با ساختارِ نحویِ خوانا نوشته شد. برای تعریفِ یک تابع و یا آغاز یک حلقه‌ی تکرار، بجای کروشه از تورفتگی استفاده می‌شود. در نتیجه، یک برنامه‌نویس پایتون می‌تواند با نگاه به قطعه کُد، به سرعت هدف و کارکردِ آن را درک کند.

زبان‌هایی مانند *فُرترن* و *سی* که از همگردان برایِ اجرایِ کُد بهره می‌گیرند، بطور ذاتی از پایتون سریع‌تر هستند؛ اما الزامن اینگونه نیست. استفاده از بسته‌هایی مانند ***Cython***، پایتون را قادر می‌سازد تا با کدهای *سی* تعامل داشته باشد و از طریق حافظه، اطلاعات را از برنامه‌ی *سی* به پایتون و برعکس، انتقال دهد. بر این اساس، پایتون می‌تواند با زبان‌های سریع‌تر برابری و رقابت کرده و از کدهای موروثی[1] استفاده کند (مانند ***FFTW***). این ویژگی، موجبِ گرایشِ شمارِ زیادی از دانشمندان و افراد دیگر به پایتون شده است. بطور ویژه، دو بسته‌ی ***NumPy*** و ***SciPy*** هسته‌ی اصلی پایتون در کاربردهای علمی و محاسباتی را تشکیل می‌دهند. افزون بر این، این دو بسته به سادگی امکان یکپارچه کردن کدهای موروثی را نیز فراهم می‌کند.

۱-۱- چرا ***SciPy*** و ***NumPy***؟

عملیات پایه‌ای موردِ استفاده در برنامه‌نویسی علمی، آرایه‌ها، ماتریس‌ها، تابع‌اولیه‌گیری، حل‌گرهای معادله‌های دیفرانسیل، آمار و... را در بر می‌گیرد. بصورت پیش‌فرض، به جز شماری از عمل‌گرهای ساده‌ی ریاضیاتی که تنها بر روی یک متغیر، و نه آرایه و ماتریس، قابل استفاده هستند، پایتون چنین قابلیت‌هایی را بطور ذاتی در اختیار ندارد. با این وجود، ***NumPy*** و ***SciPy*** دو بسته‌ی کارا و توان‌مندِ پایتونی هستند که این زبان را برای اهدافِ محاسباتی آماده می‌کنند.

NumPy مختصِ پردازش‌های عددی بوسیله‌ی آرایه‌های چندبُعدی `ndarrays` است که در آن آرایه‌ها می‌توانند محاسباتِ درایه به درایه را انجام دهند. در صورتِ نیاز، می‌توان از فرمالیسمِ جبرِ خطی بهره گرفت؛ بدون اینکه نیاز به تغییر و اصلاح آرایه‌های ***NumPy*** باشد. افزون بر این، اندازه‌ی آرایه‌ها را می‌توان به صورت پویا تغییر داد. این ویژگی، نگرانی‌هایی که

---

[1] کد موروثی کد منبعی است که دیگر مورد پشتیبانی یا استفاده‌ی سیستم‌عامل یا بقیه‌ی فن‌آوری‌های رایانه‌ای نیست – مترجم

برنامه‌نویسیِ سریع در دیگر زبان‌ها را دشوار می‌سازد، کاهش می‌دهد. هنگامی که هدف، دسترسی به چند درایه از آرایه است، به جای خلق یک آرایه‌ی جدید، می‌توان بر روی آن پوشانه اعمال کرد.

SciPy بر پایه‌ی چارچوب آرایه‌های *NumPy* استوار است. این بسته، با فراهم کردنِ توابعِ پیشرفته‌ی ریاضی مانند تابع‌اولیه‌گیری، حل‌گرهای معادلات دیفرانسیل، توابع ویژه، بهینه‌سازی و مانند اینها، برنامه‌نویسی علمی را به سطحی کاملن نوین ارتقاء می‌دهد. نام بردن و لیست کردن همه‌ی توابع به‌کاررفته در بسته‌ی *SciPy*، دست‌کم نیازمند چندین صفحه است. هنگامی که به فزونی ابزارهای *SciPy* می‌نگریم، برخی اوقات تصمیم درباره‌ی اینکه کدام تابع برای استفاده بهترین است می‌تواند دلهره‌آور باشد! دلیل نوشته شدنِ این کتاب همین است. ما در این کتاب ابزارهای اصلی و فراگیر در بسته‌های *NumPy* و *SciPy* را مورد بررسی و واکاوای قرار می‌دهیم. در نتیجه، خواننده قادر خواهد بود که به سرعت نتایج مطلوبش را بدست آورده و در مورد آنچه که برای رویارویی با مسائل فراتر از این کتاب مورد نیاز است، تصمیم‌گیری نماید.

۱-۲- نصب *NumPy* و *SciPy*

اکنون، احتمالن قانع شده‌اید و می‌پرسید «فوق‌العادست! اما از کجا می‌توانم این بسته‌ها را نصب کنم؟». چندین راه برای این کار وجود دارد و ما نخست ساده‌ترین آن را برای *OS X*، *Linux* و *Windows* بیان می‌کنیم.

دو بسته‌ی شناخته‌شده، فراگیر و پیش‌همگردانی‌شده‌ی پایتونی وجود دارند که *NumPy* و *SciPy* را در بر می‌گیرند و بر روی هر سه زیرساخت (*OS X*، *Linux* و *Windows*) قابل اجرا هستند: *the Entought Python Distribution (EPD)* و *ActivePython (AP)*. اگر نسخه‌ی رایگان این بسته‌ها را می‌خواهید باید *EPD Free* یا نسخه‌ی *AP Community* را بارگذاری کنید. اگر به پشتیبانی نیازمند هستید، همواره می‌توانید از این دو منبع، بسته‌های بیشتر و گسترده‌تری را انتخاب کنید.

اگر کاربر *MacPorts* هستید، ممکن است بخواهید بسته‌های *NumPy* و *SciPy* را با استفاده از *Package Manager* نصب کنید. برای اینکار از دستور *MacPorts* مطابق زیر برای نصب بسته‌های پایتونی استفاده کنید. توجه داشته باشید که نصب بسته‌های *NumPy* و *SciPy* با استفاده از *MacPorts* زمان‌بر خواهد بود؛ بویژه برای بسته‌ی *SciPy*. بنابراین، ایده‌ی خوبی است که نخست فرآیند نصب را آغاز کرده و در این میان، یک فنجان قهوه نوشِ جان کنید.

```
sudo port install py35-numpy py35-scipy py35-ipython
```

MacPorts از نسخه‌های گوناگون پایتون پشتیبانی می‌کند (برای نمونه ۲,٦ و ۲,۷). از این رو، اگر چه در بالا از *py35* استفاده شده است، در صورتی‌که بخواهید از پایتون ۲,٦ با *SciPy* و *NumPy* استفاده کنید، به سادگی می‌توانید *py35* را به *py26* یا *py27* تغییر دهید.

اگر از توزیع‌های لینوکسی مبتنی بر *Debian* مانند *Ubunto* یا *Linux Mint* استفاده می‌کنید، از دستور `apt-get` برای نصب بسته‌ها بهره جویید:

```
sudo apt-get install python-numpy python-scipy
```

در سامانه‌های مبتنی بر *RPM* مانند *Fedora* یا *OpenSUSE*، می‌توانید بسته‌های پایتون را با استفاده از دستور `yum` نصب کنید.

```
sudo yum install numpy scipy
```

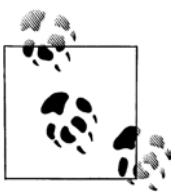
برپایی و نصب *NumPy* و *SciPy* بر روی ویندوز پیچیده‌تر از سیستم‌های مبتنی بر لینوکس است؛ چرا که همگردانیِ کد در آن دشوارتر است. خوشبختانه، یک برنامه‌ی نصبِ همگردانی‌شده‌ی باینری به نام *python(x,y)* وجود دارد که هر دو بسته‌ی *NumPy* و *SciPy* را دارا بوده و مختص ویندوز است.

افرادی که ترجیح می‌دهند از منبع اصلی، *NumPy* و *SciPy* را نصب و برپا کنند، سایت *www.scipy.org* آخرین نسخه این بسته را در اختیار قرار می‌دهد. راهکارِ دیگر، بارگذاری دو بسته‌ی مزبور از انباشت‌گاهِ کدِ *scipy* و *numpy* در سایت *github* می‌باشد. اگر کاربری حرفه‌ای در برپایی بسته‌ها از کد منبع نیستید و به دنبال چالش نمی‌گردید، توصیه می‌کنیم از همان بسته‌های پیش‌همگردانی‌شده‌ای که به آنها اشاره شده استفاده کنید.

۱-۳- کار کردن با *NumPy* و *SciPy*

شما می‌توانید از دو راه مختلف با برنامه‌های پایتون کار کنید: بصورت تعاملی یا از طریقِ اسکریپت. برخی برنامه‌نویس‌ها بر این عقیده‌اند که بهترین راه، نوشتن کد در اسکریپت است. در نتیجه در صورتِ لزوم، نیازی به تکرار کارهای خسته کننده‌ی پیشین نخواهد بود. برخی دیگر می‌گویند برنامه‌نویسی تعاملی بهترین راه است؛ چرا که می‌توانید قابلیت‌های برنامه را واکاوی و زیر و رو کنید. ما هر دو مورد را تایید می‌کنیم. اگر یک پایانه که در آن محیطِ پایتون باز است و یک ویرایش‌گرِ متن برای نوشتنِ اسکریپت داشته باشید، آنگاه بهترین‌های هر دو دنیا را در اختیار دارید!

در مورد بخش تعاملی، ما شدیدن استفاده از *IPython* را توصیه می‌کنیم. آی‌پایتون از بهترین‌های محیطِ *Bash* (مانند: استفاده از کلید *tab* برای کامل کردن دستور و تغییر دایرکتوری) بهره گرفته و آن را با محیطِ پایتون در هم می‌آمیزد. آی‌پایتون کارهایی فراتر از این را نیز انجام می‌دهد، اما برای مثال‌های به‌کاررفته در این کتاب، تنها کافیست که برپا و اجرا شود.



اشکال‌ها و باگ در برنامه‌ها، حقیقت‌های زندگی هستند و هیچ راهی برای جلوگیری از آنها وجود ندارد. توانایی یافتن اشکال‌ها و بر طرف کردنِ آسان و سریع آن‌ها، بخش بزرگی از برنامه‌نویسیِ موفق را تشکیل می‌دهد. آی‌پایتون دارای یک ویژگی است که با آن می‌توان برنامه‌های اشکال‌دار پایتون را خطایابی کرد. این کار را می‌توان با تایپِ عبارت `debug` پس از اجرای اسکریپت، انجام داد. برای جزییات بیشتر به اینجا مراجعه کنید.

فصل دوم

NumPy

۲-۱- آرایه‌های *NumPy*

NumPy یک بسته‌ی بنیادی پایتونی برای محاسبات علمی است. این بسته قابلیت‌های آرایه‌های *N*بُعدی، عملیات درایه به درایه، عملیات ریاضی اصلی مانند جبر خطی، و توانایی فراخوانی کدهای *Fortran*، *C* و *C++* را گردِ هم می‌آورد. ما در این فصل بیشتر این موارد را پوشش می‌دهیم. نخست، آرایه‌های *NumPy* را معرفی کرده و سپس از برتری‌های آن در مقایسه با لیست‌ها و دیکشنری‌های پایتون سخن خواهیم گفت.

پایتون، داده را به شیوه‌های گوناگونی انبار می‌کند؛ اما متداول‌ترین آن‌ها لیست‌ها و دیکشنری‌ها هستند. شیءِ لیست در پایتون می‌تواند تقریبن هر نوع از اشیاء پایتون را به صورت یک عضو ذخیره کند. اما عملیات بر روی عضوها در یک لیست تنها از طریق حلقه‌ها امکان‌پذیر است که در پایتون از دیدگاه محاسباتی، ناکارا و زمان‌بر است. بسته‌ی *NumPy* کابران را قادر می‌سازد تا بر این کاستیِ لیست‌هایِ پایتون چیره شوند. این کار با بهره‌گیری از یک شیءِ انبارشِ داده به نام `ndarray` انجام می‌شود.

شیِ `ndarray` مشابه لیست است؛ اما به‌جای انعطاف‌پذیری بالا در انبار کردنِ انواعِ گوناگونِ اشیاء در یک لیست، تنها نوعِ یکسانی از عضو می‌تواند در هر ستون انبار شود. برای نمونه، با یک لیستِ پایتون، شما می‌توانید اولین درایه را یک لیست و دومین درایه را یک لیستِ دیگر یا دیکشنری قرار دهید. با آرایه‌های *NumPy* شما تنها می‌توانید نوع یکسانی از عضو (برای نمونه همه‌ی درایه‌ها یا عضوها باید اعداد اعشاری، صحیح و یا رشته باشند) را انبار کنید. علیرغم این محدودیت، هنگامی‌که سخن از زمان عملیات و محاسبات به میان می‌آید، `ndarray` به سادگی پیروز می‌شود؛ چرا که سرعت عملیات به طور چشمگیری افزایش می‌یابد. با استفاده از دستورِ جادوییِ `%timeit` در آی‌پایتون، توانایی آرایه‌های *NumPy* را در برابر لیست‌های پایتون، از دیدگاه سرعت، مقایسه می‌کنیم.

```
import numpy as np
# Create an array with 10^7 elements.
arr = np.arange(1e7)
# Converting ndarray to list
larr = arr.tolist()
# Lists cannot by default broadcast,
# so a function is coded to emulate
# what an ndarray can do.
def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist
# Using IPython's magic timeit command
timeit arr * 1.1
>>> 1 loops, best of 3: 76.9 ms per loop
timeit list_times(larr, 1.1)

>>> 1 loops, best of 3: 2.03 s per loop
```

در این مثال، عملیات با `ndarray` تقریبن ۲۵ برابر تُندتر از حلقه‌ی پایتون است. آیا قانع شدید که آرایه‌های *NumPy* بهترین انتخاب هستند؟ از این پس و در ادامه، هر زمان که ممکن باشد به جای استفاده از لیست‌ها، با اشیاءِ آرایه کار خواهیم کرد.

اگر نیازمند به عملیات جبر خطی باشیم، می‌توانیم از شیء `matrix` استفاده کنیم. گفتنی است که این نوع داده از ویژگی پیش‌فرض عملیاتِ پخش[۲] که در `ndarray` تعریف شده است، سود نمی‌برد. برای نمونه، وقتی دو آرایه‌ی `ndarray`، با نام‌های A و B و با اندازه‌ی برابر در هم ضرب می‌شوند، درایه‌ی n_{ij}ام از A در درایه‌ی n_{ij}ام از B ضرب می‌شود؛ حال‌آنکه ضرب دو شیء ماتریسی، عملیاتِ ضربِ ماتریسی معمول را نتیجه می‌دهد.

بر خلاف اشیاء `ndarray`، اشیاء ماتریسی یا `matrix`، تنها می‌توانند دو بُعد داشته باشند. این بدین معناست که ساخت بُعدِ سوم و بالاتر ممکن نمی‌باشد. نمونه‌ی زیر را ببینید.

```
import numpy as np
# Creating a 3D numpy array
arr = np.zeros((3,3,3))
# Trying to convert array to a matrix, which will not work
mat = np.matrix(arr)

# "ValueError: shape too large to be a matrix."
```

اگر با ماتریس‌ها کار می‌کنید، این نکته را بخاطر بسپارید.

[۲] *Broadcasting* ابزاری را فراهم می‌آورد که با آن، عملیات بر روی آرایه‌ها بصورت برداری انجام می‌شود؛ به‌گونه‌ای که حلقه‌ها به جای پایتون در C اجرا می‌شوند؛ آشکار است که این ویژگی سرعت و کارایی محاسبات را افزایش می‌دهد – مترجم

۲-۱-۱- آفرینشِ آرایه و نوع داده

راه‌های گوناگونی برای آفرینش یک آرایه در NumPy وجود دارد که در اینجا ما در مورد آنهایی که سودمندتر هستند، بحث خواهیم کرد.

```
import numpy as np
# First we create a list and then
# wrap it with the np.array() function.
alist = [1, 2, 3]
arr = np.array(alist)
# Creating an array of zeros with five elements
arr = np.zeros(5)
# What if we want to create an array going from 0 to 100?
arr = np.arange(100)
# Or 10 to 100?
arr = np.arange(10,100)
# If you want 100 steps from 0 to 1...
arr = np.linspace(0, 1, 100)
# Or if you want to generate an array from 1 to 10
# in log10 space in 100 steps...
arr = np.logspace(0, 1, 100, base=10.0)
# Creating a 5x5 array of zeros (an image)
image = np.zeros((5,5))
# Creating a 5x5x5 cube of 1's
# The astype() method sets the array with integer elements.
cube = np.zeros((5,5,5)).astype(int) + 1
# Or even simpler with 16-bit floating-point precision...
cube = np.ones((5, 5, 5)).astype(np.float16)
```

هنگامیکه که آرایه‌ها خلق می‌شوند، NumPy مقدار پیش‌فرض برای عمقِ بیت را بر اساس محیط پایتون برمی‌گزیند. اگر با پایتون ٦٤بیتی کار می‌کنید، آنگاه دقت درایه‌های آرایه بصورت پیش‌فرض ٦٤بیتی خواهد بود. این دقت، قطعه‌ی نسبتن بزرگی از حافظه را اشغال می‌کند و معمولن همیشه نیاز به این میزان دقت نیست. در هنگام خلق یک آرایه، شما می‌توانید عمقِ بیت را با استفاده از پارامتر نوع داده (dtype) به int، numpy.float16، numpy.float32 یا numpy.float64 تغییر دهید. در این مثال، چگونگی انجام این کار نشان داده شده است.

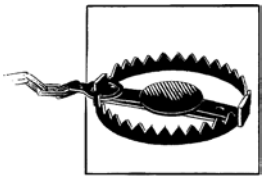
```
# Array of zero integers
arr = np.zeros(2, dtype=int)
# Array of zero floats
arr = np.zeros(2, dtype=np.float32)
```

اکنون که آرایه‌ها را خلق کردیم، می‌توانیم از راه‌های مختلفی اندازه‌ی آن‌ها را تغییر دهیم. اگر یک آرایه‌ی ۲۵ عضوی داشته باشیم، می‌توانیم آن را به یک آرایه‌ی ۵×۵ و یا یک آرایه‌ی ۳ بُعدی تبدیل کنیم.

```
# Creating an array with elements from 0 to 999
arr1d = np.arange(1000)
# Now reshaping the array to a 10x10x10 3D array
arr3d = arr1d.reshape((10,10,10))
# The reshape command can alternatively be called this way
arr3d = np.reshape(arr1d, (10, 10, 10))
```

```
# Inversely, we can flatten arrays
arr4d = np.zeros((10, 10, 10, 10))
arr1d = arr4d.ravel()
print(arr1d.shape)
```

بازآرایی ساختار آرایه‌های به روش‌های زیادی و به سادگی قابل انجام است.



به خاطر داشته باشید که آرایه‌های تغییرشکل یافته در بالا فقط یک نما از همان داده‌ها در حافظه هستند. به سخن دیگر، اگر شما یکی از این آرایه‌ها را اصلاح کنید، دیگر آرایه‌ها نیز دست‌خوش تغییر می‌شوند. برای نمونه، اگر یکمین درایه از `arr1d` در مثال بالا را به ۱ تغییر دهید، یکمین درایه از `arr3d` نیز ۱ خواهد شد. اگر نمی‌خواهید این اتفاق رخ دهد، از تابع `numpy.copy` برای جدا کردن آرایه از حافظه استفاده کنید.

۲-۱-۲- آرایه‌های رکوردی

آرایه‌ها عمومن مجموعه‌ای از اعداد صحیح یا اعشاری هستند. اما برخی اوقات لازم است که داده‌هایی با ساختار پیچیده‌تری و با انواع مختلف در هر ستون از آرایه انبار شوند. در مجلاتِ پژوهشی، جدول‌ها معمولن به گونه‌ای هستند که شامل چند ستون با متغیر رشته‌ای برای مشخصات و ستون‌های دیگر با متغیرهای اعشاری برای کمیت‌های عددی هستند. توانایی در انبار کردنِ این گونه از اطلاعات می‌تواند بسیار سودمند باشد. در *NumPy* این کار با `numpy.recarray` انجام می‌شود. ساختن یک `recarray` برای بار نخست می‌تواند اندکی گیج‌کننده باشد؛ از این رو در زیر مفاهیم مقدماتی را مرور می‌کنیم.

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))
toadd = [(1,2.,'Hello'),(2,3.,"World")]
recarr[:] = toadd
```

آرگومانِ اختیاری `dtype` نوعِ داده‌ی در نظر گرفته شده برای ستون‌های یکم تا سوم را مشخص می‌کند. `i4` نشان‌دهنده‌ی عدد صحیح ۳۲بیتی، `f4` نمایش‌گرِ عدد اعشاری ۳۲بیتی و `a10` برابر با یک رشته با طول ۱۰ کاراکتر است. جزییاتِ بیشتر در مورد چگونگی تعریفِ انواعِ داده‌های بیشتر را می‌توان در راهنمای *NumPy* پیدا کرد. مثال بالا، چگونگی تعریف یک `recarray` را نشان می‌دهد؛ برای برپایی این‌گونه از آرایه‌ها به روشی ساده، می‌توان از تابع عمومی پایتون، `zip`، استفاده کرد. تابع مزبور، لیستی از چندتایی‌ها مطابق آنچه که در مثالِ بالا برای شیء `toadd` تعریف شده است، می‌آفریند. در ادامه نشان می‌دهیم که چگونه می‌توان با دستور `zip` این کار را انجام داد.

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))
# Now creating the columns we want to put
# in the recarray
col1 = np.arange(2) + 1
col2 = np.arange(2, dtype=np.float32)
col3 = ['Hello', 'World']
# Here we create a list of tuples that is
# identical to the previous toadd list.
toadd = zip(col1, col2, col3)
# Assigning values to recarr
recarr[:] = list(toadd)
```

```
# Assigning names to each column, which
# are now by default called 'f0', 'f1', and 'f2'.
recarr.dtype.names = ('Integers' , 'Floats', 'Strings')
# If we want to access one of the columns by its name, we
# can do the following.
recarr['Integers']
# array([1, 2], dtype=int32)
```

کار کردن با ساختار `recarray` ممکن است اندکی کسل‌کننده به نظر برسد؛ اما در ادامه و در بخش خواندن و نوشتن، هنگامی که چگونگی خواندنِ داده‌های پیچیده با *NumPy* را بررسی می‌کنیم، اهمیت بیشتری خواهد یافت.

اگر در زمینه‌ی ستاره‌شناسی یا فیزیکِ نجوم پژوهش می‌کنید و عمومن با جداول داده‌ها سر و کار دارید، یک بسته‌ی سطحِ بالا با عنوان *ATpy* می‌تواند برایتان جالب باشد. این بسته به کاربر اجازه‌ی خواندن، نوشتن و تبدیل جداول داده‌ها با فرمت *FITS*، *ASCII*، *HDF5* و *SQL* را می‌دهد.

۲-۱-۳-اندیس‌گذاری و برش

اندیسِ لیست‌ها در پایتون از صفر شروع می‌شود و آرایه‌های *NumPy* نیز از همین قاعده پیروی می‌کنند. برای اندیس‌گذاریِ یک شیء ۲×۲، بطور معمول در پایتون بصورت زیر عمل می‌کنیم.

```
alist=[[1,2],[3,4]]
# To return the (0,1) element we must index as shown below.
alist[0][1]
```

اگر بخواهیم به ستونِ سمتِ راستِ آرایه‌ی بالا دسترسی داشته باشیم، راهکاری ساده برای انجام این کار با لیست‌های پایتون وجود ندارد. اما در *NumPy*، اندیس‌گذاری از قاعده‌ی ساده‌تری پیروی می‌کند.

```
# Converting the list defined above into an array
arr = np.array(alist)
# To return the (0,1) element we use ...
arr[0,1]
# Now to access the last column, we simply use ...
arr[:,1]
# Accessing the columns is achieved in the same way,
# which is the bottom row.
arr[1,:]
```

در برخی موارد، شیوه‌هایِ پیچیده‌تری برای اندیس‌گذاری مورد نیاز خواهد بود؛ مانند اندیس‌گذاری شرطی. متداول‌ترینِ این نوع، دستور `numpy.where()` است. با استفاده از این تابع می‌توانید بر اساسِ پاره‌ای از شرط‌ها، اندیس‌های دلخواه خود را صرف‌نظر از بُعدِ آرایه، استخراج کنید.

```
# Creating an array
arr = np.arange(5)
# Creating the index array
index = np.where(arr > 2)
print(index)
```

```
#(array([3, 4]),)
# Creating the desired array
new_arr = arr[index]
```

ممکن است بخواهید برخی از اندیس‌های خاص را پاک کنید. برای این کار می‌توانید از دستور numpy.delete() استفاده کنید. در این صورت، متغیرهای ورودی این تابع، آرایه‌ها و اندیس‌هایی هستند که می‌خواهید پاک شوند.

```
# We use the previous array
new_arr = np.delete(arr, index)
```

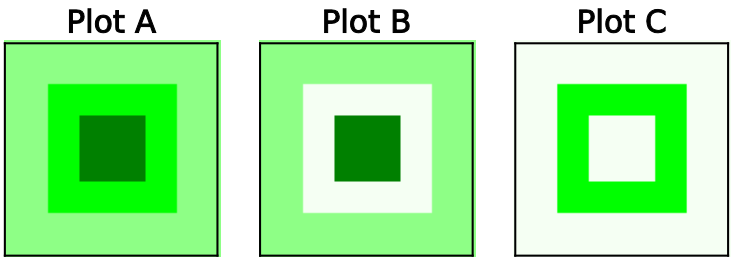
به جای استفاده از تابع numpy.delete می‌توانید از یک آرایه‌ی ساده‌ی بولی برای استخراج درایه‌های مورد نظر نیز استفاده کنید.

```
index = arr > 2
print(index)
    [False False True True True]
new_arr = arr[index]
```

کدام روش بهتر است و چه زمانی از کدام یک استفاده کنیم؟ اگر سرعت برایتان مهم است، اندیس‌گذاری بولی برای حالتی که با شمار زیادی از درایه‌های سروکار دارید، تُندتر است. افرون بر این، می‌توانید با استفاده از عمل‌گر ~، شیءهای True و False در یک آرایه را معکوس کنید. این فن، سریع‌تر از تابع numpy.where است.

۲-۲- گزاره‌های بولی و آرایه‌های *NumPy*

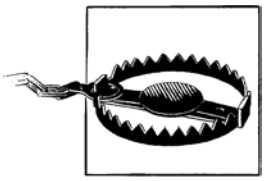
گزاره‌های بولی معمولن در ترکیب با عمل‌گرهای and و or استفاده می‌شوند. این عمل‌گرها هنگامی که مقادیر بولی با یکدیگر مقایسه می‌شوند، سودمند هستند. در هنگام استفاده از آرایه‌های *NumPy*، می‌توانید از عمل‌گرهای & و | استفاده کنید. عمل‌گرهای مزبور مقادیر بولی را با سرعت بیشتری مقایسه می‌کنند. اگر با منطقِ صوری آشنا باشید، خواهید دید کاری که ما در *NumPy* می‌توانیم انجام دهیم در واقع تعمیمی از کاری است که با آرایه‌ها انجام می‌دادیم. در زیر یک مثال از اندیس‌گذاری با استفاده از گزاره‌های بولی مرکب که در شکل ۲-۱ نمایش داده شده‌اند، آورده شده است.



شکل ۲-۱: سه نمودار که چگونگی اندیس‌گذاری در پایتون را نشان می‌دهند

```
# Creating an image
img1 = np.zeros((20, 20)) + 3
img1[4:-4, 4:-4] = 6
```

```
img1[7:-7, 7:-7] = 9
# See Plot A
# Let's filter out all values larger than 2 and less than 6.
index1 = img1 > 2
index2 = img1 < 6
compound_index = index1 & index2
# The compound statement can alternatively be written as
compound_index = (img1 > 3) & (img1 < 7)
img2 = np.copy(img1)
img2[compound_index] = 0
# See Plot B.
# Making the boolean arrays even more complex
index3 = img1 == 9
index4 = (index1 & index2) | index3
img3 = np.copy(img1)
img3[index4] = 0
# See Plot C.
```



در هنگام برپایی آرگومان‌های پیچیده‌ی بولی، مهم است که از پرانتز استفاده شود. درست همانند تقدمِ عمل‌گرها در ریاضی، باید آرگومان‌های بولی را به گونه‌ای ساماندهی کنید که در نهایت یک گذاره‌ی منطقیِ درست ساخته شود.

در حالتی خاص، ممکن است بخواهید روی درایه‌های خاصی از آرایه عملیات انجام دهید که به سادگی انجام‌شدنی است.

```
import numpy as np
import numpy.random as rand
# Creating a 100-element array with random values
# from a standard normal distribution or, in other
# words, a Gaussian distribution.
# The sigma is 1 and the mean is 0.
a = rand.randn(100)
# Here we generate an index for filtering
# out undesired elements.
index = a > 0.2
b = a[index]
# We execute some operation on the desired elements.
b = b ** 2 - 2
# Then we put the modified elements back into the
# original array.
a[index] = b
```

۲-۳-خواندن و نوشتن

خواندن و نوشتن اطلاعات بر روی فایل‌های داده، چه با فرمت متن باشد و چه دودویی، در محاسباتِ علمی بسیار پراهمیت است. این ابزار توانایی ذخیره، به اشتراک گذاری و خواندن داده که ممکن است خروجی هر زبانی باشد را فراهم می‌آورد. خوشبختانه پایتون در خواندن و نوشتن داده‌ها بسیار توانا است.

۲-۳-۱-فایل‌های متنی

در زمینه‌ی فایل‌های متنی، پایتون یکی از توان‌مندترین زبان‌های برنامه‌نویسی است. تجزیه کردن در این زبان نه تنها قدرتمند و انعطاف‌پذیر است، بلکه در مقایسه با دیگر زبان‌ها مانند *C*، سریع است. در اینجا یک مثال از چگونگی باز کردن و تجزیه‌ی اطلاعات متن توسط پایتون آورده شده است.

```
# Opening the text file with the 'r' option,
# which only allows reading capability
f = open('somefile.txt', 'r')
# Parsing the file and splitting each line,
# which creates a list where each element of
# it is one line
alist = f.readlines()
# Closing file
f.close()

# After a few operations, we open a new text file
# to write the data with the 'w' option. If there
# was data already existing in the file, it will be overwritten.
f = open('newtextfile.txt', 'w')
# Writing data to file
f.writelines(newdata)
# Closing file
f.close()
```

دسترسی و ثبت داده به روش نشان داده شده می‌تواند بسیار سریع و انعطاف‌پذیر باشد، اما یک کاستی دارد: اگر حجم فایل بزرگ باشد، آنگاه دسترسی به داده‌ها و سامان‌بخشیِ آنها دشوار و کُند انجام خواهد شد. انتقالِ مستقیمِ داده‌ها به آرایه‌های *NumPy* بهترین گزینه خواهد بود. این کار را می‌توانیم با یکی از توابع *NumPy* به نام `loadtxt` انجام دهیم. اگر داده‌ها با سطر و ستون ساخت یافته باشد، آنگاه دستور `loadtxt` بسیار خوب عمل خواهد کرد؛ البته تا زمانی که همه‌ی داده‌ها از یک نوع باشند؛ مثلن اعداد صحیح یا اعشاری. می‌توانیم با استفاده از `numpy.savetxt`، به همان سادگیِ تابع `loadtxt`، داده‌ها را ذخیره کنیم.

```
import numpy as np
arr = np.loadtxt('somefile.txt')
np.savetxt('somenewfile.txt')
```

اگر فرمتِ هر ستون متفاوت باشد، دستورِ `loadtxt` باز هم می‌تواند داده‌ها را بخواند، اما نوع ستون‌ها باید از پیش تعریف شود. در این صورت، داده‌هایی که خوانده می‌شوند از نوع `recarray` خواهند بود. در این مثال، چگونگی برخورد پایتون با این ساختمان داده‌ی پیچیده را خواهیم دید.

```
# example.txt file looks like the following
#
# XR21 32.789 1
# XR22 33.091 2
table = np.loadtxt('example.txt',
dtype={'names': ('ID', 'Result', 'Type'),'formats': ('S4', 'f4', 'i2')})
# array([('XR21', 32.78900146484375, 1),
# ('XR22', 33.090999603271484, 2)],
# dtype=[('ID', '|S4'), ('Result', '<f4'), ('Type', '<i2')])
```

بر پایه‌ی آن‌چه که در مورد اشیاء `recarry` بیان شد، می‌توانیم با استفاده از نام هر ستون به آن دسترسی داشته باشیم؛ مثلن `table['Result']`. دسترسی به هر ستون مشابه همان فرآیندی است در آرایه‌های معمولی `numpy.array` انجام می‌شد.

اشیاء `recarray` یک کاستی دارند: در نسخه‌ی ۱٫۸ بسته‌ی *NumPy*، روشی خودکار و قابل اطمینان برای ذخیره‌ی ساختمان داده‌ی `numpy.recarry` به فرمت متنی وجود ندارد[۳]. اگر این ویژگی برایتان مهم است، بهترین راه استفاده از ابزارهای بسته‌ی *matplotlib.mlab* می‌باشد.

یک بسته‌ی عمومی و بسیار سریع برای خواندن و نوشتن فایل‌های متنی به نام *Asciitable* وجود دارد. اگر خواندن و نوشتن فایل‌هایی با فرمت *ASCII* در کارتان زیاد استفاده می‌شود، این بسته را باید در کنار *NumPy* داشته باشید.

۲-۳-۲- فایل‌های دودویی

فایل‌های متنی به دلیل قابلیت حمل و کاربرپسند بودن، گزینه‌ای عالی برای خواندن، انتقال و انبارِ داده‌ها هستند. از سوی دیگر، کار کردن با فایل‌های دودویی دشوارتر بوده و قالب‌بندی، خوانایی و قابلیتِ حملِ آن سخت‌تر است. با این وجود فایل‌های دودویی، دو برتریِ چشم‌گیر در مقایسه با فایل‌های متنی دارند: حجم فایل و سرعت خواندن یا نوشتن. این ویژگی، مخصوصن در کار کردن با فایل‌های حجیم می‌تواند پراهمیت واقع شود.

در *NumPy*، می‌توان به فایل‌های دودویی با استفاده از توابع `numpy.save` و `numpy.load` دسترسی داشت. محدودیتِ اصلی این است که فرمتِ دودویی تنها در سیستم‌هایی که از *NumPy* استفاده می‌کنند قابل خواندن است. اگر می‌خواهید خواندن و نوشتن فایل‌ها را به فرمتی قابلِ‌حمل‌تر انجام دهید، تابع `scipy.io` این کار را برایتان انجام خواهد داد که در فصل بعد به آن خواهیم پرداخت. در حال حاضر، اجازه دهید قابلیت‌های *NumPy* را مرور کنیم.

```
import numpy as np
# Creating a large array
data = np.empty((1000, 1000))
# Saving the array with numpy.save
np.save('test.npy', data)
```

[۳] این ویژگی در آخرین نسخه‌ی بسته‌ی *NumPy* به آن اضافه شده است. دستور `numpy.recarray.tofile` این کار را انجام می‌دهد.

```
# If space is an issue for large files, then
# use numpy.savez instead. It is slower than
# numpy.save because it compresses the binary
# file.
np.savez('test.npz', data)
# Loading the data array
newdata = np.load('test.npy')
```

خوشبختانه، توابع numpy.save و numpy.savez مشکلی با ذخیره‌ی اشیاء numpy.recarray ندارند. از این رو اگر قابلیتِ حمل فراتر از محیط پایتون اهمیتی نداشته باشد، کار کردن با آرایه‌های پیچیده و ساخت‌یافته آسان خواهد بود.

۲-۴-ریاضیات

پایتون دارای ماژول math است که بر روی اشیاء اختصاصی پایتون قابل اعمال است. متاسفانه، اگر سعی کنید که تابع math.cos را بر روی یک آرایه‌ی NumPy اعمال کنید، با خطا روبرو خواهید شد؛ چرا که توابعِ ماژول math به‌گونه‌ای طراحی شده‌اند که بر روی عضوها عمل کنند و نه لیست‌ها یا آرایه‌ها. از این رو، NumPy ابزارهایِ ریاضیِ مخصوصِ خودش را دارد. این ابزارها به‌گونه‌ای بهینه شده‌اند که بر روی آرایه‌های NumPy قابل اعمال باشند و همچنین، عملیات را با سرعت بالا انجام دهند. وقتی که بسته‌ی NumPy را وارد پایتون می‌کنید، بیشترِ ابزارهایِ ریاضی به صورت خودکار وارد می‌شوند؛ از توابع ساده‌ی مثلثاتی و لگاریتمی گرفته تا توابعی پیچیده‌تر مانند تبدیل سریع فوریه (FFT) و عملیات جبر خطی.

۲-۴-۱- جبر خطی

آرایه‌های NumPy بصورتِ پیش‌فرض مانند ماتریس‌ها در جبر خطی رفتار نمی‌کنند. در عوض، عملیات از هر عضو از یک آرایه به بَعدی نگاشت می‌شود. این ویژگیِ بسیار سودمندی است؛ چرا که در این صورت می‌توان از برای کارایی بیشتر، حلقه‌ها را کنار گذاشت. اما هنگامی که ترانهاده یا ضرب نقطه‌ای مورد نیاز باشد، چه می‌توان کرد؟ بدون اینکه کلاسِ دیگری فراخوانی شود، می‌توانید از توابعِ داخلی numpy.transpose یا numpy.dot برای انجام این عملیات استفاده کنید. ریاضی‌گراها می‌توانند به جای آرایه از شیء numpy.matrix بهره جویند. در ادامه هر دو رویکرد را بررسی خواهیم کرد تا تفاوت‌ها و شباهت‌های آن‌ها آشکار گردد. از آن مهم‌تر، پاره‌ای از برتری‌ها و کاستی‌های دو شیءِ numpy.array و numpy.matrix را مقایسه خواهیم کرد.

برخی عملیات در جبر خطی به آسانی و سرعت انجام می‌شوند. یک مثال شناخته‌شده، حل یک دستگاه معادله است که می‌توانیم آن را به فرم ماتریسی بیان کنیم:

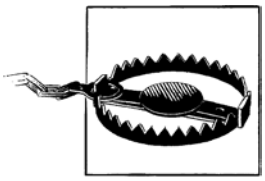
$$\begin{aligned} 3x + 6y - 5z &= 12 \\ x - 3y + 2z &= -2 \\ 5x - y + 4z &= 10 \end{aligned} \qquad (۱-۲)$$

$$\begin{bmatrix} 3 & 6 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix} \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} 12 \\ -2 \\ 10 \end{Bmatrix} \qquad (۲-۲)$$

اکنون، اجازه دهید تا دستگاه ماتریسی را بصورت **AX=B** بیان کرده و آن را حل کنیم. این بدین معناست که می‌خواهیم حاصل **$X=A^{-1}B$** را بدست آوریم. چگونگی انجام این کار با *NumPy* را در زیر نشان دادیم.

```
import numpy as np
# Defining the matrices
A = np.matrix([[3, 6, -5],
[1, -3, 2],
[5, -1, 4]])
B = np.matrix([[12],
[-2],
[10]])
# Solving for the variables, where we invert A
X = A ** (-1) * B
print(X)
# matrix([[ 1.75],
# [ 1.75],
# [ 0.75]])
```

پاسخِ متغیرهای دستگاه برابر $x=1.75$، $y=1.75$ و $z=0.75$ می‌باشد. شما می‌توانید به سادگی با محاسبه‌ی **AX** و مقایسه‌ی آن با درایه‌های بردار **B**، درستی پاسخ‌ها را کنترل کنید. انجام این گونه عملیات با *NumPy* ساده بوده و دستگاه معادلات بالا نیز به آسانی برای ماتریس‌های دوبُعدیِ بزرگتر قابل پیاده‌سازی می‌باشد.



همه‌ی ماتریس‌ها وارون‌پذیر نیستند؛ از این رو این شیوه برای بدست آوردن پاسخ دستگاه همواره عملی نخواهد بود. شما می‌توانید با استفاده از دستور `numpy.linalg.svd` که معمولن در وارون کردن ماتریس‌های بد–وضع خوب کار می‌کند، از کنار این مشکل عبور کنید.

اکنون که فهمیدیم ماتریس‌ها در *NumPy* چگونه کار می‌کنند، می‌توانیم نشان دهیم که چگونه همین عملیات، بدون استفاده از زیرکلاس `numpy.matrix` قابل انجام است. (زیرکلاس `numpy.matrix` در کلاس `numpy.array` قرار دارد؛ بدین معنا که می‌توان مثال بالا را بدون فراخوانی مستقیم کلاس `numpy.matrix` پیاده‌سازی کرد.)

```
import numpy as np
a = np.array([[3, 6, -5],
[1, -3, 2],
[5, -1, 4]])
# Defining the array
b = np.array([12, -2, 10])
# Solving for the variables, where we invert A
x = np.linalg.inv(a).dot(b)
print(x)
# array([ 1.75, 1.75, 0.75])
```

هر دو روش در رویارویی با عملیات جبر خطی، عملی است؛ اما کدام بهترین است؟ از دیدگاه نحوی، روش `numpy.matrix` ساده‌ترین است. با این وجود، `numpy.array` عملی‌تر است. نخست اینکه آرایه‌های *NumPy* تقریبن برای هر نوع محاسبات علمی در محیط پایتون استاندارد هستند؛ بنابراین باگ‌ها و مشکلاتِ مرتبط با عملیات جبر خطی در این حالت کمتر از مورد

`numpy.matrix` می‌باشد. افزون بر این، در مثال‌هایی مشابه آنچه که در بالا نشان داده شد، روش `numpy.array` از دیدگاه محاسباتی سریع‌تر است.

تبادل داده از یک کلاس به کلاس دیگر می‌تواند دشوار باشد و اگر به درستی انجام نشود، ممکن است منجر به نتایج غیرمنتظره گردد. برای نمونه، هنگامی که پاره‌ای از عملیات توسط `numpy.matrix` انجام شده و ادامه‌ی آن به `numpy.array` انتقال داده می‌شود، احتمال بروز این مشکل وجود خواهد داشت. استفاده از یک ساختمان داده بطور کلی نگرانی و مشکلات کمتری را در مقایسه با انتقال داده بین ماتریس‌ها و آرایه‌ها خواهد داشت. بنابراین، توصیه می‌شود که هر زمان ممکن است از `numpy.array` استفاده شود.

فصل سوم

SciPy

با *NumPy* می‌توانیم با کدهای ساده، به سرعت به پاسخ دست پیدا کنیم. اما *SciPy* کجا خودش را نشان می‌دهد؟ *SciPy* بسته‌ای است که از آرایه‌های *NumPy* بهره می‌گیرد تا مسائل استانداردی که دانشمندان و مهندسین عمومن با آن روبرو می‌شوند را حل کند. برای نمونه: تابع‌اولیه‌گیری، تعیین بیشینه‌ها و کمینه‌های یک تابع، یافتن بردارهای ویژه‌ی ماتریس‌های بزرگ و تُنُک، کنترل اینکه آیا دو توزیع یکسان هستند یا خیر و مانند این‌ها. ما در اینجا، با بررسی مثال‌های ساده که در مسائلِ دنیایِ واقعی کاربرد دارند، موارد پایه را پوشش می‌دهیم که در نهایت به شما این توانایی را می‌دهد تا از ویژگی‌های پیچیده‌تر *SciPy* استفاده کنید.

نخست، کارمان را با بهینه‌سازی و برازش داده آغاز می‌کنیم؛ چرا که این موارد بسیار متداول هستند. به دنبال آن، مباحث درون‌یابی، تابع‌اولیه‌گیری، تحلیل فضایی، خوشه‌بندی، پردازش سیگنال و تصویر، ماتریس‌های تُنُک و آمار را پوشش می‌دهیم.

۳-۱- بهینه‌سازی و کمینه سازی

بسته‌ی بهینه‌سازی موجود در *SciPy* به ما توانایی حلِ ساده و سریع مسائل کمینه‌سازی را می‌دهد. اما صبر کنید: کمینه‌سازی چیست و چگونه می‌تواند به کار شما کمک کند؟ مثال‌های شناخته‌شده در این زمنیه شامل رگرسیون خطی، یافتن بیشینه و کمینه‌ی یک تابع و مقادیر آن‌ها، تعیین ریشه‌ی یک تابع و محل قطع دو تابع می‌باشد. در زیر، با یک مثال ساده‌ی رگرسیون خطی کار را آغاز می‌کنیم و سپس برای برازش داده‌های خطی آن را تعمیم می‌دهیم.

ابزارهای بهینه‌سازی و کمینه‌سازی موجود در *NumPy* و *SciPy* فوق‌العاده هستند؛ اما روش‌های مونت کارلوی زنجیر مارکوفی (*MCMC*)، یا به سخن دیگر تحلیل بیزی، در آن‌ها گنجانده نشده است. اما بسته‌های متداول دیگری مانند *PyMC* (یک بسته‌ی ارزشمند با امکانات فراوان) و *emcee* (برای مسایل بزرگ‌مقیاس) وجود دارند.

۳-۱-۱- مدل‌سازی داده و برازش

روش‌های گوناگونی برای برازش داده با رگرسیون خطی وجود دارد. در این بخش ما از تابع `curve_fit` که یک روش مبتنی بر χ^2 (بهترین برازش) است، بهره خواهیم گرفت. در مثال زیر، از یک تابع شناخته شده با نویز، داده‌ها را تولید کرده و سپس

داده‌های نویزی را با curve_fit برازش می‌دهیم. تابعی که ما در این مثال مدل خواهیم کرد یک معادله‌ی ساده‌ی خطی به فرم $f(x)=ax+b$ است.

```
import numpy as np
from scipy.optimize import curve_fit
# Creating a function to model and create data
def func(x, a, b):
    return a * x + b
# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 2)
# Adding noise to the data
yn = y + 0.9 * np.random.normal(size=len(x))
# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)
# popt returns the best fit values for parameters of
# the given model (func).
print(popt)
```

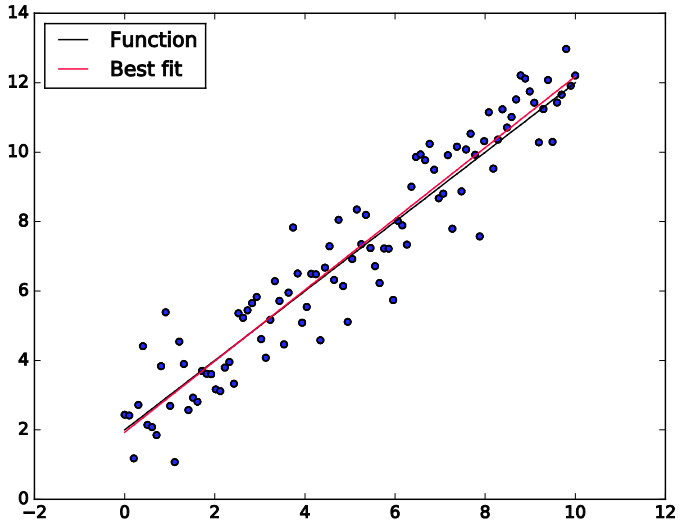
مقادیر popt، اگر برازش خوب باشد، باید نزدیک به مقادیر yباشند. شما می‌توانید کیفیت برازش را با استفاده از متغیر pcov، که اعضای قطری آن واریانس هر پارامتر را نشان می‌دهد، کنترل کنید. شکل ۳-۱ نتایج برازش را نمایش می‌دهد.

در گام بعدی، می‌خواهیم برازش حداقل مربعات را بر منحنی گوسی که یک تابع ناخطی است، اعمال کنیم:

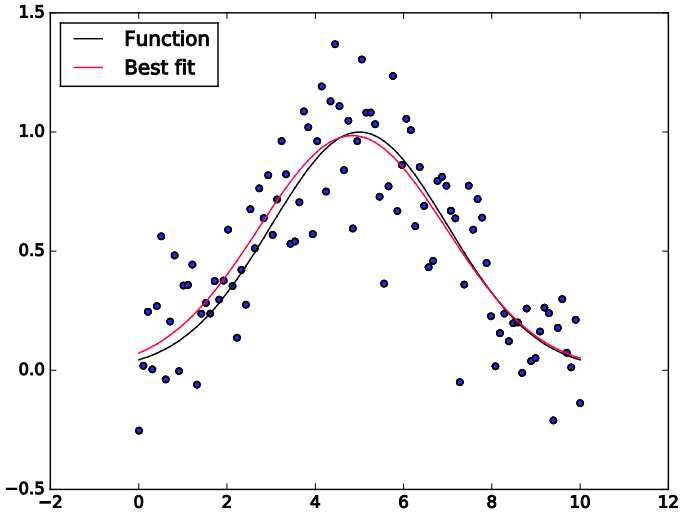
$$a \times \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \qquad (۱-۳)$$

در این رابطه a یک کمیت نرده‌ای، μ میانگین و σ انحراف معیار استاندارد است.

```
# Creating a function to model and create data
def func(x, a, b, c):
    return a*np.exp(-(x-b)**2/(2*c**2))
# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 5, 2)
# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))
# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)
# popt returns the best-fit values for parameters of the given model (func).
print(popt)
```



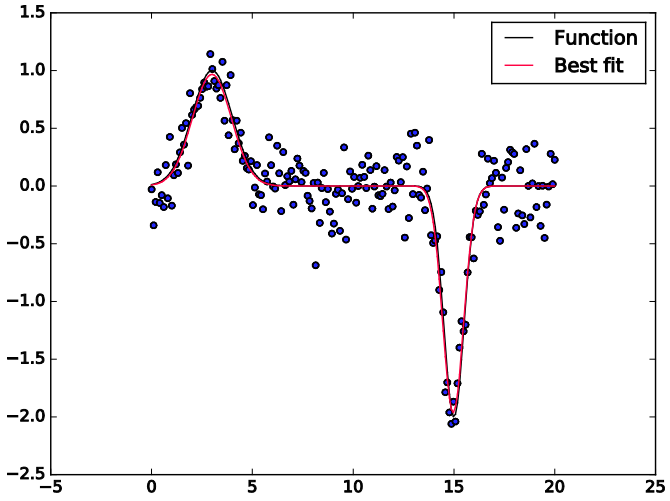
شکل ۳-۱: برازش داده‌های نویزی با یک معادله‌ی خطی



شکل ۳-۲ : برازش داده‌های نویزی با یک معادله‌ی گوسی

همان‌طور که از شکل ۳-۲ پیدا است، نتایج برازش گوسی، پذیرفتنی است.

در گام بعد، می‌توانیم مجموعه داده‌ی یک بُعدی را بر چند منحنی گوسی برازش دهیم. تابع func اکنون به گونه‌ای بیان می‌شود که دو معادله‌ی گوسی با متغیرهای ورودی متفاوت را شامل گردد. این مثال، حالتِ شناخته‌شده‌یِ برازشِ طیفِ خط می‌باشد (شکل ۳-۳ را ببینید).



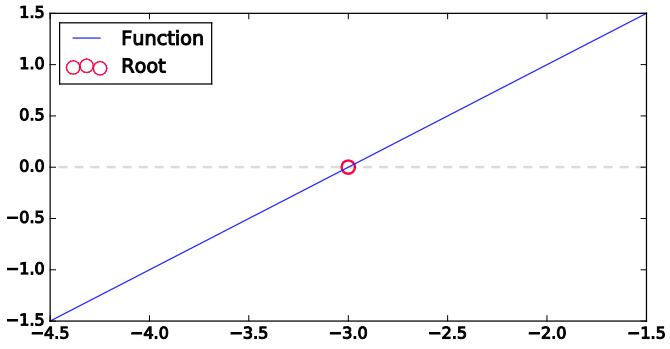
شکل ۳-۳: برازش داده‌های نویزی با چند معادله‌ی گوسی

```
# Two-Gaussian model
def func(x, a0, b0, c0, a1, b1,c1):
    return a0*np.exp(-(x - b0) ** 2/(2 * c0 ** 2))\
    + a1 * np.exp(-(x - b1) ** 2/(2 * c1 ** 2))
# Generating clean data
x = np.linspace(0, 20, 200)
y = func(x, 1, 3, 1, -2, 15, 0.5)
# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))
# Since we are fitting a more complex function,
# providing guesses for the fitting will lead to
# better results.
guesses = [1, 3, 1, 1, 15, 1]
# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn, p0=guesses)
```

۳-۱-۲- جواب‌های توابع

پس از آشنایی با مدل‌سازی داده و برازش، اکنون می‌توانیم کار را با یافتن جواب‌ها ادامه دهیم؛ مانند اینکه ریشه‌ی یک تابع چیست یا اینکه دو تابع کجا یکدیگر را قطع می‌کنند. *SciPy* مجموعه‌ای از این ابزارها را در ماژول `optimize` گردِ هم آورده است. پاره‌ای از موارد اصلی را در این بخش ارائه خواهیم کرد.

برای شروع، ریشه‌ی یک معادله را بدست می‌آوریم (شکل ۳-٤ را ببینید). در اینجا، از تابع `scipy.optimize.fsolve` استفاده خواهیم کرد.



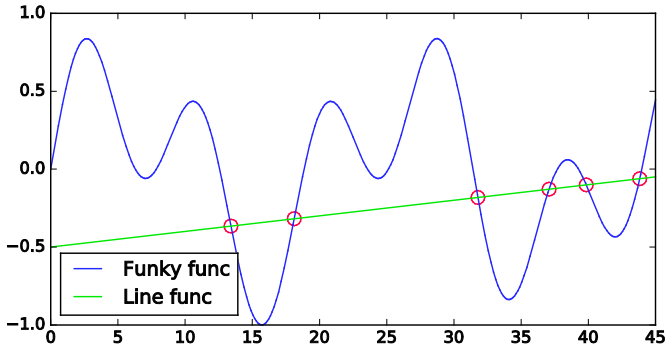
شکل ۳-٤: ریشه‌ی تقریبی یک تابع خطی در *y=0*

```
from scipy.optimize import fsolve
import numpy as np
line = lambda x: x + 3
solution = fsolve(line, -2)
print(solution)
```

یافتن محل تقاطع دو معادله نیز به سادگی انجام می‌شود.

```
from scipy.optimize import fsolve
import numpy as np
# Defining function to simplify intersection solution
def findIntersection(func1, func2, x0):
    return fsolve(lambda x : func1(x) - func2(x), x0)
# Defining functions that will intersect
funky = lambda x : np.cos(x / 5) * np.sin(x / 2)
line = lambda x : 0.01 * x - 0.5
# Defining range and getting solutions on intersection points
x = np.linspace(0,45,10000)
result = findIntersection(funky, line, [15, 20, 30, 35, 40, 45])
# Printing out results for x and y
print(result, line(result))
```

همان‌طور که در شکل ۳-۵ مشاهده می‌شود، نقاط قطع به خوبی شناسایی شده‌اند. به خاطر داشته باشید که پنداشت‌های نخستین در مورد این‌که دو تابع در کجا یکدیگر را قطع می‌کنند، مهم هستند. اگر این مقادیر درست نباشند، ممکن است منجر به نتایجِ به ظاهر درست شوند.



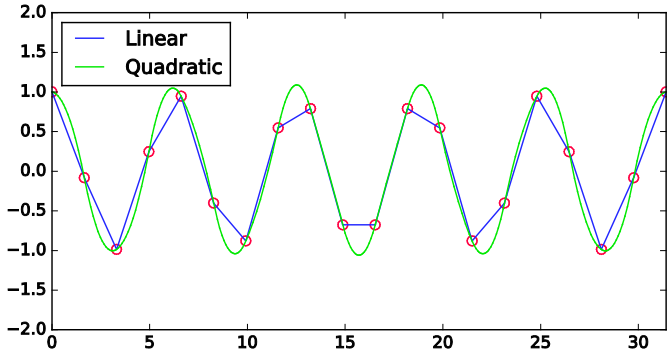
شکل ۳-۵: بافتن محل‌های تقاطع دو تابع

۳-۲-درون‌یابی

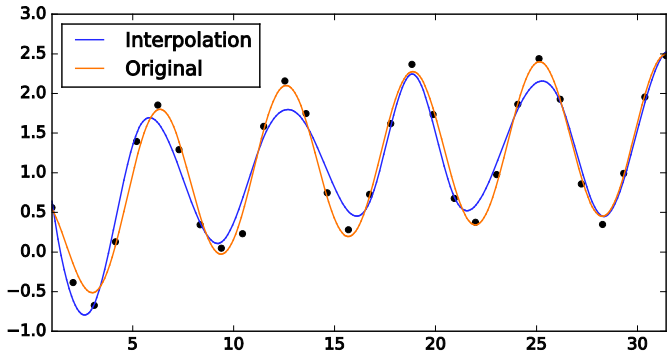
داده‌هایی که شاملِ اطلاعات هستند معمولن به فرم تابع بوده و ما بعنوانِ تحلیل‌گر، می‌خواهیم آن را مدل کنیم. با داشتن مجموعه‌ای از داده‌های نمونه، بدست آوردنِ مقادیرِ میانیِ بینِ دو نقطه مفید خواهد بود؛ زیرا در این صورت می‌توان مقادیر داده در دامنه‌ی نمونه‌برداری‌نشده را تخمین زد. *SciPy* ده‌ها تابع گوناگون برای درون‌یابی را در اختیار قرار می‌دهد؛ از حالت‌های تک‌متغیره‌ی ساده گرفته تا مواردِ پیچیده‌ی چندمتغیره. درون‌یابی تک‌متغیره هنگامی استفاده می‌شود که داده‌های نمونه‌برداری شده، احتمالن، تنها از یک متغیر مستقل پیروی می‌کنند. این در حالی است که در درون‌یابی چندمتغیره فرض بر این است داده‌ها از بیش از یک متغیر مستقل تبعیت می‌کنند.

دو راه اساسی برای درون‌یابی وجود دارد: ۱- برازش یک تابع بر مجموعه‌ی داده‌ها یا ۲- برازش چندین تابع به بخش‌های گوناگون مجموعه‌ی داده به گونه‌ای به نَرمی به یکدیگر پیوسته شده باشند. نوع دوم با نامِ درون‌یابیِ اسپیلاین شناخته می‌شود. این‌گونه درون‌یابی هنگامی که داده‌ها فرمِ تابعیِ پیچیده‌ای داشته باشند، می‌تواند ابزار بسیار توانمندی باشد. نخست نشان خواهیم داد که چگونه یک تابع ساده را درون‌یابی کنیم و سپس به حالت پیچیده‌تری خواهیم پرداخت. مثال زیر یک تابع سینوسی (شکل ۳-۶) را با استفاده از تابع `scipy.interpolate.interp1d` با پارامترهای برازش مختلف، درون‌یابی می‌کند. پارامتر اول برازش خطی و پارامتر دوم برازش درجه ۲ است.

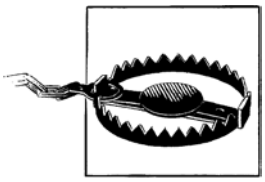
```
import numpy as np
from scipy.interpolate import interp1d
# Setting up fake data
x = np.linspace(0, 10 * np.pi, 20)
y = np.cos(x)
# Interpolating data
fl = interp1d(x, y, kind='linear')
fq = interp1d(x, y, kind='quadratic')
# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yintl = fl(xint)
yintq = fq(xint)
```



شکل ۳-٦: درون‌یابی داده‌های ساخته‌شده (دایره‌های قرمز) با پارامترهای خطی و درجه ۲



شکل ۳-۷: درونیابی داده‌های ساخته‌شده‌ی نویزی



شکل ۳-۶ نشان می‌دهد که در این حالت، برازش درجه ۲ خیلی بهتر است. این مثال، اهمیت انتخاب مناسب پارامترها را در هنگام درون‌یابی را نشان می‌دهد.

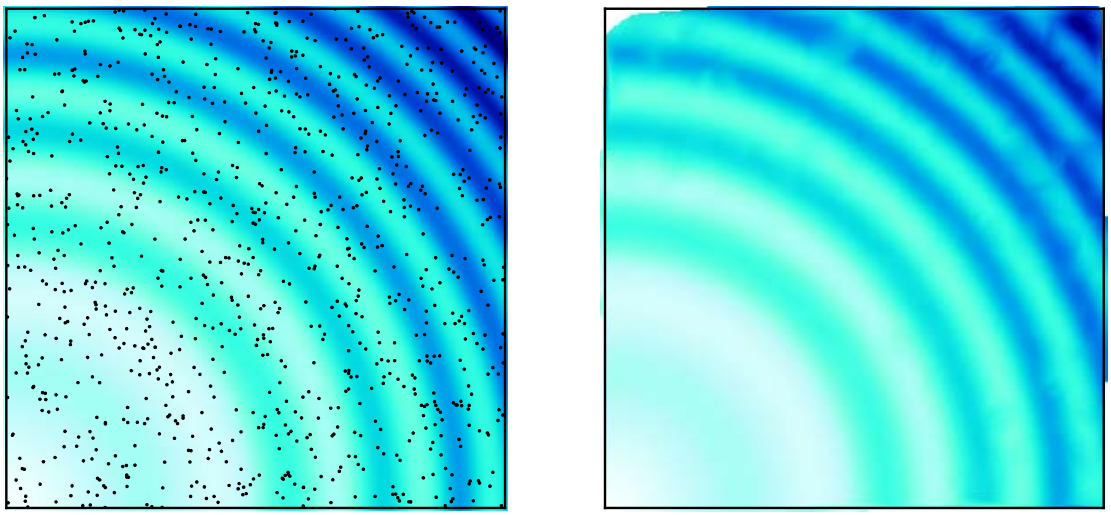
آیا می‌توانیم داده‌های نویزی را درون‌یابی کنیم؟ بله، این کار بسیار ساده است اگر از یک تابع برازش اسپیلاین به نام `scipy.inetrpolate.UnivariateSpline` استفاده کنید. (نتایج در شکل ۳-۷ نشان داده شده است.)

```
import numpy as np
import matplotlib.pyplot as mpl
from scipy.interpolate import UnivariateSpline
# Setting up fake data with artificial noise
sample = 30
x = np.linspace(1, 10 * np.pi, sample)
y = np.cos(x) + np.log10(x) + np.random.randn(sample) / 10
# Interpolating the data
f = UnivariateSpline(x, y, s=1)
# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
```

```
xint = np.linspace(x.min(), x.max(), 1000)
yint = f(xint)
```

آرگومان s عامل نَرمی بوده که در هنگام برازش داده‌های نویزی باید از آن استفاده شود. اگر s=0 باشد درون‌یابی از همه‌ی نقاط عبور کرده و از نویزها چشم‌پوشی می‌کند.

اکنون، یک مثال چندمتغیره را بررسی می‌کنیم. در این حالت، می‌خواهیم یک تصویر را بازتولید کنیم. از تابع scipy.interpolate.griddata به دلیل توانایی در کار کردن با داده‌های چندبُعدیِ بدونِ ساختار، بهره خواهیم گرفت. مثلن، اگر یک تصویر ۱۰۰۰×۱۰۰۰ پیکسل و ۱۰۰۰ نقطه‌ی تصادفی روی آن داشته باشید، آیا به خوبی می‌توانید این تصویر را بازتولید کنید؟ به شکل ۳-۸ رجوع کنید تا ببینید که تابع scipy.interpolate.griddata چقدر خوب کار می‌کند.



شکل ۳-۸: تصویر اصلی با نمونه‌ی تصادفی (نقاط مشکی، سمت چپ) و تصویر درون‌یابی شده (سمت راست)

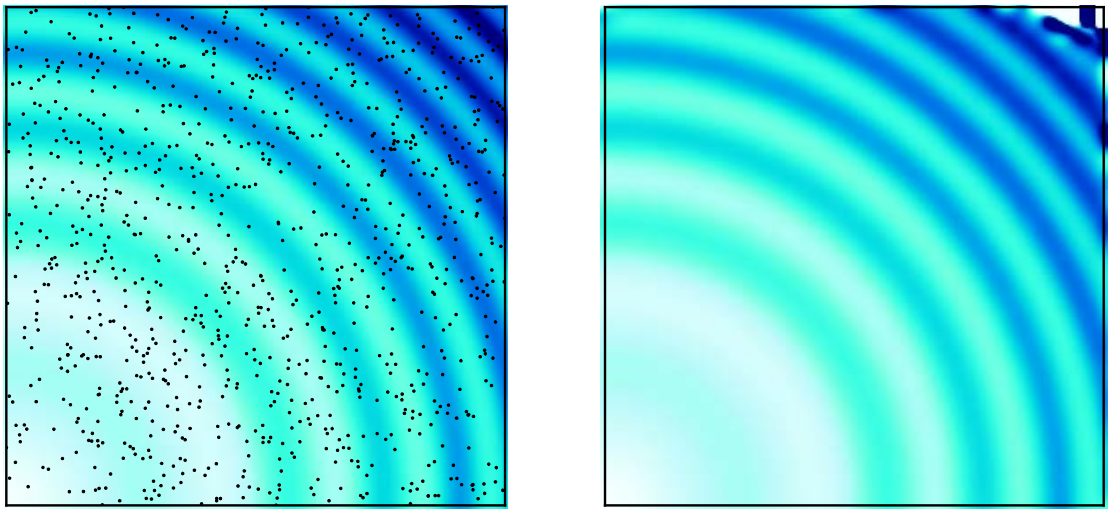
سمت چپ شکل ۳-۸ تصویر اصلی بوده و نقاط مشکی موقعیت‌های نمونه‌ی تصادفی هستند. در سمت راست، تصویر درون‌یابی شده دیده می‌شود. خطاهای اندکی را می‌توان در تصویر درون‌یابی شده مشاهده کرد که ناشی از کم بودنِ شمارِ نقطه‌های نمونه است. بر این اساس، تنها راه برای دست‌یابی به یک درون‌یابی بهتر، افزایشِ تعدادِ نمونه‌ها می‌باشد.

اما اگر از یک درون‌یابِ چندمتغیره‌ی اسپیلاینِ دیگر استفاده کنیم، نتایج چه تفاوتی خواهند کرد؟ در اینجا از تابع scipy.interpolate.SmoothBivariateSpline بهره گرفته می‌شود که کُدی مشابه مثال پیشین خواهد داشت.

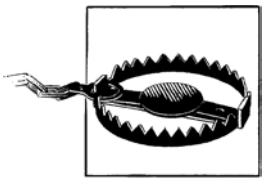
```
import numpy as np
from scipy.interpolate import SmoothBivariateSpline as SBS
# Defining a function
ripple = lambda x, y: np.sqrt(x**2 + y**2)+np.sin(x**2 + y**2)
# Generating sample that interpolation function will see
xy= np.random.rand(1000, 2)
x, y = xy[:,0], xy[:,1]
sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)
# Interpolating data
```

```
fit = SBS(x * 5, y * 5, sample, s=0.01, kx=4, ky=4)
interp = fit(np.linspace(0, 5, 1000), np.linspace(0, 5, 1000))
```

مطابق شکل ۳-۹، نتایجی مشابه با مثال پیشین خواهیم داشت. بخش سمت چپ، تصویرِ اصلی به همـراه نقطـه‌هـای نمونـه‌ی تصادفی را نشان داده و در سمت راست، تصویر درون‌یابی شده دیده می‌شود. به استثناء گوشه‌ی بالای سمت راست تصویر، به نظر می‌رسد که تابع `SmoothBivariateSpline` اندکی بهتر از `griddata` کار می‌کند.



شکل ۳-۹: تصویر اصلی با نمونه‌ی تصادفی (نقاط مشکی، سمت چپ) و تصویر درون‌یابی شده (سمت راست)



اگر چه مطابق شکل، به نظر می‌رسد که `SmoothBivariateSpline` بهتر عمل می‌کند، اما کافی اسـت کُـد را چندین بار اجرا کنید تا ببینید چه اتفاقی رخ می‌دهد. تابع `SmoothBivariateSpline` نسبت بـه داده‌هـای نمونه‌اش بسیار حساس است و درون‌یابی‌هایش می‌تواند با خطا همراه باشـد. `girddata` تـابعی قـوی‌تـر بـوده و صرف‌نظر از داده‌های ورودی‌اش، می‌تواند درون‌یابی معقول‌تری را ارائه دهد.

۳-۳- تابع‌اولیه‌گیری

تابع‌اولیه‌گیری ابزاری مهم در ریاضیات و علوم به شمار می‌رود. با داشتنِ منحنی حاصل از یک تابع و یا مجموعه‌ای از داده‌هـا، می‌توانیم مساحت زیر آن را محاسبه کنیم. در فضای پژوهشی، داده‌ها به ندرت به فرم تابع هستند و محاسبه‌ی تابع‌اولیـه، بـرخلاف آن چه که در کلاس‌های درسی انجام می‌گرفت (محاسبه‌ی تحلیلی تابع‌اولیه)، نیازمند یک فرآیند تقریبی خواهد بود.

هدف اصلی در تابع‌اولیه‌گیری با *SciPy*، بدست آوردنِ پاسخِ عددی است. اگـر بـه دنبـال محاسـبه‌ی تـابع‌اولیـه‌ی نامعین هستید، به بسته‌ی *SymPy* نیم‌نگاهی داشته باشید. این بسته مسائل گوناگون ریاضی را بصورت سـمبولیک مورد بررسی قرار می‌دهد.

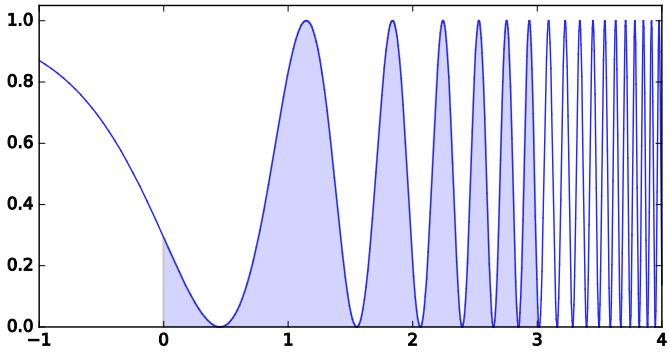
SciPy مجموعه‌ای از توابعِ مختلف را برای تابع‌اولیه‌گیری از معادله‌ها و داده‌ها در اختیار دارد. نخست، این توابع را مرور خواهیم کرد و سپس تابع‌اولیه‌گیری از داده‌های گسسته را مورد بررسی قرار خواهیم داد. به دنبال آن، از ابزار برازشِ داده که پیش‌تر از آن استفاده کرده بودیم، برای محاسبه‌ی تابع‌اولیه‌ی معین بهره خواهیم گرفت.

۳-۳-۱-تابع‌اولیه‌گیری تحلیلی

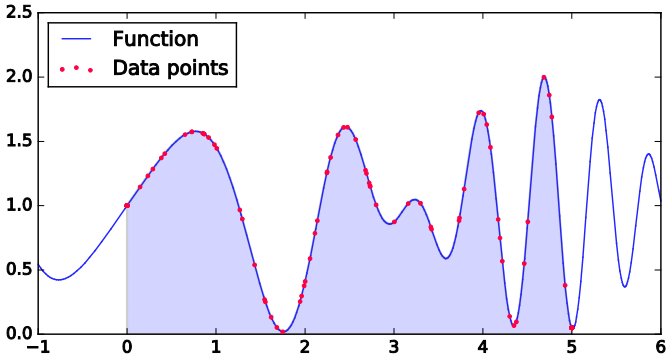
کارِ خود را با تابعِ بیان‌شده در زیر آغاز خواهیم کرد. تابع‌اولیه‌گیری از این معادله سرراست بوده و خطایِ تقریبِ پاسخِ آن، ناچیز است. در شکل ۳-۱۰، منحنی تابع و مساحت زیر آن مشخص شده است.

$$\int_0^3 \cos^2\left(e^x\right)dx \qquad (۲-۳)$$

```
import numpy as np
from scipy.integrate import quad
# Defining function to integrate
func = lambda x: np.cos(np.exp(x)) ** 2
# Integrating function with upper and lower
# limits of 0 and 3, respectively
solution = quad(func, 0, 3)
print(solution)
# The first element is the desired value
# and the second is the error.
# (1.296467785724373, 1.397797186265988e-09)
```



شکل ۳-۱۰: تابع‌اولیه‌ی معین (ناحیه‌ی سایه زده شده) یک تابع



شکل ۳-۱۱: تابع‌اولیه‌ی معین (ناحیه‌ی سایه زده شده) یک تابع. تابع اصلی با خط و داده‌های تصادفی با نقطه‌های قرمز نشان داده شده‌اند

۳-۳-۲-تابع‌اولیه‌گیری عددی

در این جا به مساله‌ای می‌پردازیم که در آن به جای یک معادله، مجموعه‌ای از داده‌ها در دسترس بوده و بنابراین، تابع‌اولیه‌گیری عددی موردنیاز می‌باشد. شکل ۳-۱۱ نشان می‌دهد که از چه نوع داده‌ی نمونه برای تقریبِ تابع‌اولیه‌ی معین استفاده شده است.

```
import numpy as np
from scipy.integrate import quad, trapz
# Setting up fake data
x = np.sort(np.random.randn(150) * 4 + 4).clip(0,5)
func = lambda x: np.sin(x) * np.cos(x ** 2) + 1
y = func(x)
# Integrating function with upper and lower
# limits of 0 and 5, respectively
fsolution = quad(func, 0, 5)
dsolution = trapz(y, x=x)
print('fsolution = ' + str(fsolution[0]))
print('dsolution = ' + str(dsolution))
print('The difference is ' + str(np.abs(fsolution[0] - dsolution)))
# fsolution = 5.10034506754
# dsolution = 5.04201628314
# The difference is 0.0583287843989.
```

تابع `quad` تنها با توابعی که قابلِ فراخوانی باشند کار می‌کند. این در حالی است که `trapz` از داده‌های گسسته برای محاسبه‌ی عددی تابع‌اولیه بهره می‌گیرد.

۳-۴-آمار

در *NumPy*، توابع پایه‌ای آماری مانند `mean`، `std`، `median`، `argmax` و `argmin` وجود دارد. افزون بر این، `numpy.arrays` شامل متدهایی هستند که توانایی استفاده‌ی آسان از توابعِ آماری *NumPy* را به ما می‌دهند.

```
import numpy as np
# Constructing a random array with 1000 elements
x = np.random.randn(1000)
# Calculating several of the built-in methods
# that numpy.array has
```

```
mean = x.mean()
std = x.std()
var = x.var()
```

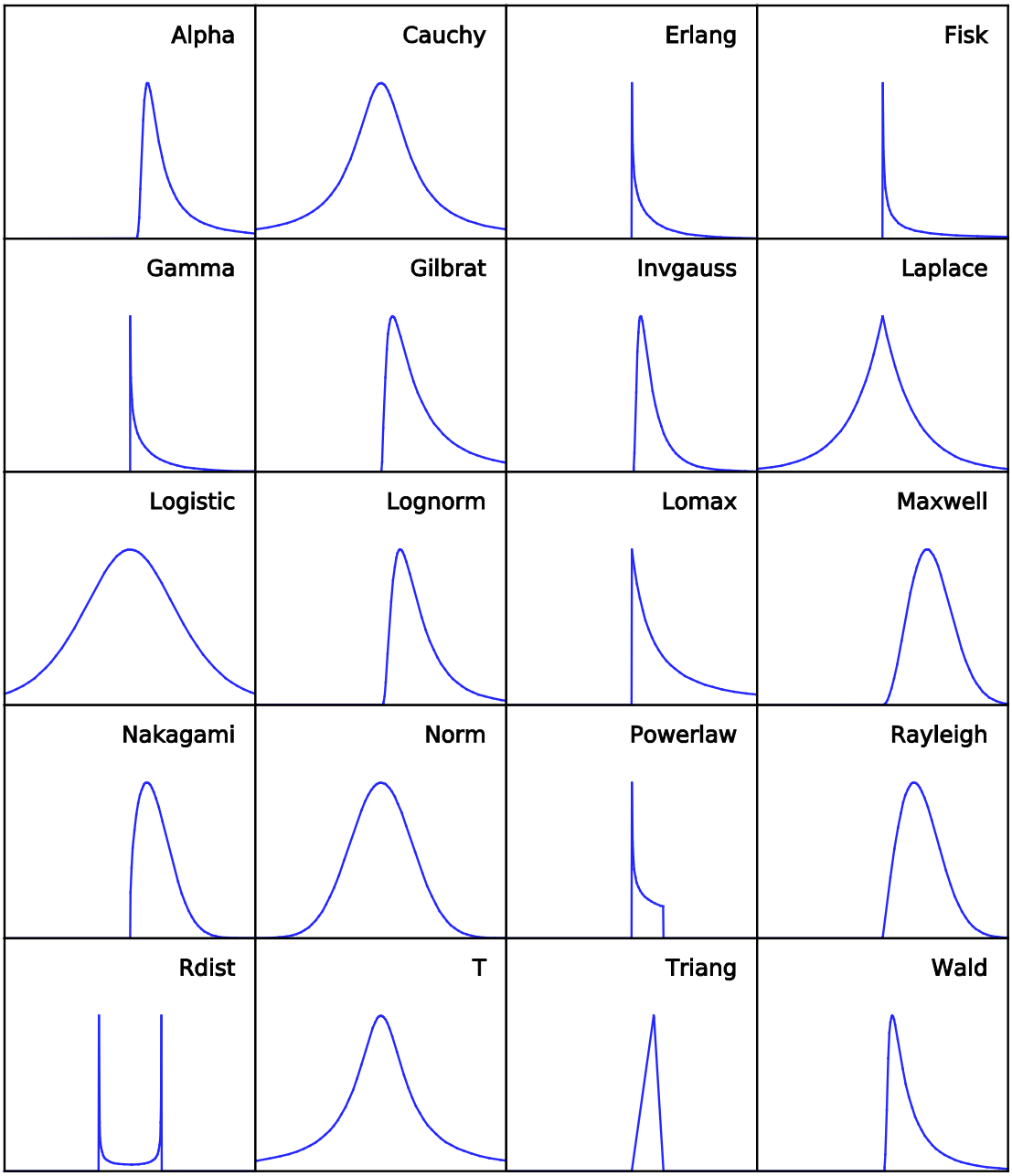
برای محاسبات سریع، متدهای بالا سودمند هستند؛ اما برای پـژوهش‌هـای کمّـی، بـه بـیش از ایـن نیـاز خواهـد بـود. *SciPy* مجموعه‌ی گسترده‌ای از ابزارهای آماری مانند توزیع‌ها (پیوسته و گسسته) و توابـع را در اختیـار مـی‌گـذارد. نخسـت، چگـونگی برون‌یابی انواعِ مختلفِ توزیع‌ها را بیان خواهیم کرد. سپس، متداول‌ترین توابع آماری *SciPy* که در رشته‌های مختلف کـاربرد دارد را مورد بررسی قرار می‌دهیم.

۳-۴-۱- توزیع‌های پیوسته و گسسته

حـدود ۸۰ توزیـع پیوسـته و بـیش از ۱۰ توزیـع گسسـته وجـود دارد. ۲۰تـا از توزیـع‌هـای پیوسـته‌ی موجـود در بسـته‌ی `scipy.stats` را بصورتِ تابع‌هایِ چگالیِ احتمال در شکل ۳-۱۲ نشان داده‌ایم. این توزیع‌ها برای تولید اعـداد تصـادفی، مشـابه توابعی که در `numpy.random` وجود دارند، سودمند هستند. با این وجود، بر خلاف توابع `numpy.random` که تنهـا محـدود بـه توزیع‌های یکنواخت و گوسی هستند، *SciPy* مجموعه‌ای ارزشمند و متنوع از توزیع‌ها را در اختیار می‌گذارد.

وقتی که از بسته‌ی `scipy.stats` یک توزیع را فراخوانی می‌کنیم، از روش‌های مختلفی می‌توانیم اطلاعـات آن را اسـتخراج کنیم: تابعِ چگالیِ احتمال (*PDF*)، تابعِ توزیعِ تجمعی (*CDF*)، نمونه‌های متغیر تصادفی (*RVSs*)، تابعِ توزیـعِ معکـوس (*PPFs*) و مانند این‌ها. اکنون چگونه می‌توان از *SciPy* این توزیع‌ها را فراخوانی کرد؟ با استفاده از تابعِ نرمالِ کلاسیک، نشـان مـی‌دهـیم کـه چگونه می‌توان به یک توزیع دسترسی داشت:

$$\mathrm{PDF} = e^{\left(-x^2/2\right)/\sqrt{2\pi}} \qquad (۳-۳)$$



شکل ۳-۱۲: نمونه‌ای از ۲۰ توزیع پیوسته در ***SciPy***

```
import numpy as np
from scipy.stats import norm
# Set up the sample range
x = np.linspace(-5,5,1000)
# Here set up the parameters for the normal distribution,
# where loc is the mean and scale is the standard deviation.
dist = norm(loc=0, scale=1)
# Retrieving norm's PDF and CDF
pdf = dist.pdf(x)
cdf = dist.cdf(x)
```

```
# Here we draw out 500 random values from the norm.
sample = dist.rvs(500)
```

مطابق مثال، با استفاده از دو متغیر `loc` و `scale` می‌توان میانگین و انحراف معیارِ توزیع را تنظیم کرد. فرآیند مزبور به سادگی برای دیگر توزیع‌ها نیز قابل انجام است و در صورت لزوم می‌توان به راهنمای بسته مراجعه کرد.

در موارد دیگر، ممکن است به توزیع‌های گسسته مانند پواسن، دوجمله‌ای یا هندسی نیاز باشد. بر خلاف توزیع‌های پیوسته، توزیع‌های گسسته برای مسائلی که در آن تعداد معلومی از پیشامدها در یک بازه‌ی مشخصِ فضا یا زمان رخ می‌دهند، مناسب هستند. این پیشامدها با یک نرخ میانگین معلوم رخ داده و هر پیشامد مستقل از پیشامد قبلی است.

معادله‌ی (۳-٤) تابعِ جرمِ احتمالِ توزیع هندسی (*PMF*) را نشان می‌دهد.

$$\mathrm{PDF} = (1-p)^{k-1}\,p \qquad (٤\text{-}٣)$$

```
import numpy as np
from scipy.stats import geom
# Here set up the parameters for the geometric distribution.
p = 0.5
dist = geom(p)
# Set up the sample range.
x = np.linspace(0, 5, 1000)
# Retrieving geom's PMF and CDF
pmf = dist.pmf(x)
cdf = dist.cdf(x)
# Here we draw out 500 random values.
sample = dist.rvs(500)
```

۳-۴-۲- توابع

بیش از ٦۰ تابع آماری در *SciPy* وجود دارد. تابع‌های آماری معمولن نمونه‌ها را توصیف یا آزمایش می‌کنند؛ مثلن، فراوانی مقادیر مشخص یا آزمون اسمیرنف-کولموگروف.

از آنجا که *SciPy* گستره‌ی وسیعی از توزیع‌ها را پوشش می‌دهد، بهتر است از همان‌هایی که پیش‌تر اشاره شد، بهره جوییم. در بسته‌ی `stats` شماری از توابع مانند `kstest` و `normaltest` وجود دارند که نمونه‌ها را می‌آزمایند. این آزمون‌های توزیع می‌توانند در تعیین اینکه آیا یک نمونه متعلق به توزیع خاصی است یا خیر، سودمند واقع شوند. پیش از اعمال این آزمون‌ها، برای جلوگیری هرگونه برداشت غلط از نتایج تابع‌ها، از درکِ درستِ داده‌های خود اطمینان حاصل کنید.

```
import numpy as np
from scipy import stats
# Generating a normal distribution sample
# with 100 elements
sample = np.random.randn(100)
# normaltest tests the null hypothesis.
out = stats.normaltest(sample)
print('normaltest output')
print('Z-score = ' + str(out[0]))
```

```
print('P-value = ' + str(out[1]))
# kstest is the Kolmogorov-Smirnov test for goodness of fit.
# Here its sample is being tested against the normal distribution.
# D is the KS statistic and the closer it is to 0 the better.
out = stats.kstest(sample, 'norm')
print('\nkstest output for the Normal distribution')
print('D = ' + str(out[0]))
print('P-value = ' + str(out[1]))
# Similarly, this can be easily tested against other distributions,
# like the Wald distribution.
out = stats.kstest(sample, 'wald')
print('\nkstest output for the Wald distribution')
print('D = ' + str(out[0]))
print('P-value = ' + str(out[1]))
```

پژوهشگران عمومن از توابعِ توصیفی برای آمار استفاده می‌کنند. برخی از توابعِ توصیفی موجود در بسته‌ی `stats` شاملِ میانگینِ هندسی (`gmean`)، چولگی یک نمونه (`skew`) و فراوانی داده‌ها در یک نمونه (`itemfreq`) می‌باشد. استفاده از این تابع‌ها ساده بوده و نیازی به ورودی چندانی ندارد. در ادامه به چند مثال می‌پردازیم.

```
import numpy as np
from scipy import stats
# Generating a normal distribution sample
# with 100 elements
sample = np.random.randn(100)
# The harmonic mean: Sample values have to
# be greater than 0.
out = stats.hmean(sample[sample > 0])
print('Harmonic mean = ' + str(out))
# The mean, where values below -1 and above 1 are
# removed for the mean calculation
out = stats.tmean(sample, limits=(-1, 1))
print('\nTrimmed mean = ' + str(out))
# Calculating the skewness of the sample
out = stats.skew(sample)
print('\nSkewness = ' + str(out))
# Additionally, there is a handy summary function called
# describe, which gives a quick look at the data.
out = stats.describe(sample)
print('\nSize = ' + str(out[0]))
print('Min = ' + str(out[1][0]))
print('Max = ' + str(out[1][1]))
print('Mean = ' + str(out[2]))
print('Variance = ' + str(out[3]))
print('Skewness = ' + str(out[4]))
print('Kurtosis = ' + str(out[5]))
```

توابعِ بیشترِ بسیاری در بسته‌ی `stats` در دسترس است؛ بنابراین اگر به ابزارهای بیشتری نیاز دارید، راهنمای آن را مطالعه کنید. اگر به بیشتر از آنچه که در `stats` وجود دارد احتیاج دارید، بسته‌ی *RPy* را امتحان کنید. *R* یک بسته‌ی بنیادی برای تحلیل‌های آماری است و *RPy* ابزارهای موجود در *R* را برای پایتون قابل استفاده می‌سازد. اگر از آنچه که در *SciPy* و *NumPy* وجود دارد رضایت دارید اما نیازمند تحلیلِ خودکارتری هستید، نگاهی به بسته‌ی *Pandas* بیندازید. این بسته یک ابزارِ توانمند

برای تحلیل‌های آماری بر روی داده‌های بزرگ و حجیم می‌باشد و خروجی‌های آن به هر دو صورت عددی و نمودار قابل دسترس می‌باشد.

۳-۵- تحلیل خوشه‌بندی و فضایی

از علوم زیست‌شناسی گرفته تا نجوم، تحلیل فضایی و خوشه‌بندی کلیدِ شناسایی الگوها، گروه‌ها و خوشه‌ها است. برای نمونه در زیست‌شناسی، فاصله‌ی بین گونه‌های مختلف گیاهی به چگونگی پراکندگی دانه‌ها، اندرکنش با محیط و رشد آنها اشاره دارد. در فیزیک نجوم، این فن‌های تحلیل برای جستجو و شناسایی خوشه‌های ستاره، خوشه‌های کهکشان و رشته‌های بزرگ‌مقیاس (تشکیل شده از خوشه‌های کهکشان) مورد استفاده قرار می‌گیرد. در زمینه‌ی علوم رایانه، شناسایی و نگاشت شبکه‌های پیچیده‌ی گره‌ها و اطلاعات، به نوبه‌ی خود یک مطالعه‌ی مهم به‌شمار می‌رود. در حوزه‌ی داده‌های بزرگ و داده‌کاوی، شناسایی خوشه‌های داده برای سامان‌دهی اطلاعاتِ کشف‌شده، اهمیت پیدا می‌کند.

اگر به دنبال بسته‌ای با قابلیت‌های خوب در زمینه‌ی نگره‌ی گراف می‌گردید، *NetworkX* را فراموش نکنید. این بسته یک ابزار عالی برای آفرینش، پیمانه‌بندی و مطالعه‌ی شبکه‌های پیچیده‌ی گراف به شمار می‌رود (یعنی تحلیل کوتاهترین درخت گسترش).

SciPy یک کلاسِ تحلیلِ فضایی (`scipy.spatial`) و یک کلاسِ تحلیلِ خوشه‌ای (`scipy.cluster`) را در اختیار می‌گذارد. کلاسِ فضایی شاملِ توابعی برای تحلیلِ فواصلِ بینِ نقاط داده است (برای مثال درخت *k-d*). کلاس خوشه‌ای، دو زیرکلاس را در بر می‌گیرد: کوانتشِ بردار (`vq`) و خوشه‌بندی سلسله‌مراتبی (`hierarchy`). کوانتشِ بردار، مجموعه‌ی بزرگی از نقاط داده (بردارها) را گروه‌بندی می‌کند که هر گروه با مرکز ثقل‌اش بیان می‌شود. زیرکلاس `hierarchy`، دارای توابعی است که خوشه‌ها را ساخته و زیرخوشه‌ها را تحلیل می‌کند.

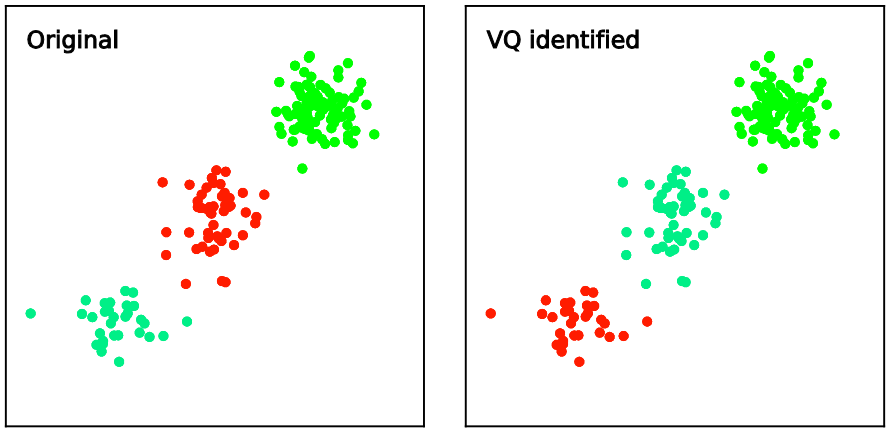
۳-۵-۱-کوانتش بردار

کوانتش بردار یک اصطلاح عمومی است که می‌تواند با پردازش سیگنال، فشرده‌سازی داده و خوشه‌بندی در ارتباط باشد. در اینجا ما بر روی خوشه‌بندی تمرکز خواهیم کرد. در ادامه، چگونگی ارسال داده به بسته‌ی `vq` برای شناسایی خوشه‌ها را بیان می‌کنیم.

```
import numpy as np
from scipy.cluster import vq
# Creating data
c1 = np.random.randn(100, 2) + 5
c2 = np.random.randn(30, 2) - 5
c3 = np.random.randn(50, 2)
# Pooling all the data into one 180 x 2 array
data = np.vstack([c1, c2, c3])
# Calculating the cluster centroids and variance
```

```
# from kmeans
centroids, variance = vq.kmeans(data, 3)
# The identified variable contains the information
# we need to separate the points in clusters
# based on the vq function.
identified, distance = vq.vq(data, centroids)
# Retrieving coordinates for points in each vq
# identified core
vqc1 = data[identified == 0]
vqc2 = data[identified == 1]
vqc3 = data[identified == 2]
```

مطابق شکل ۳-۱۳، نتایج خوشه‌های شناسایی شده به خوبی با داده‌های اصلی هم‌خوانی دارد (خوشه‌های تولید شده سمت چپ و خوشه‌های شناسایی شده در سمت راست قرار دارند).



شکل ۳-۱۳: خوشه‌های اصلی (سمت چپ) و خوشه‌های شناسایی شده vq.kmeans (سمت راست).

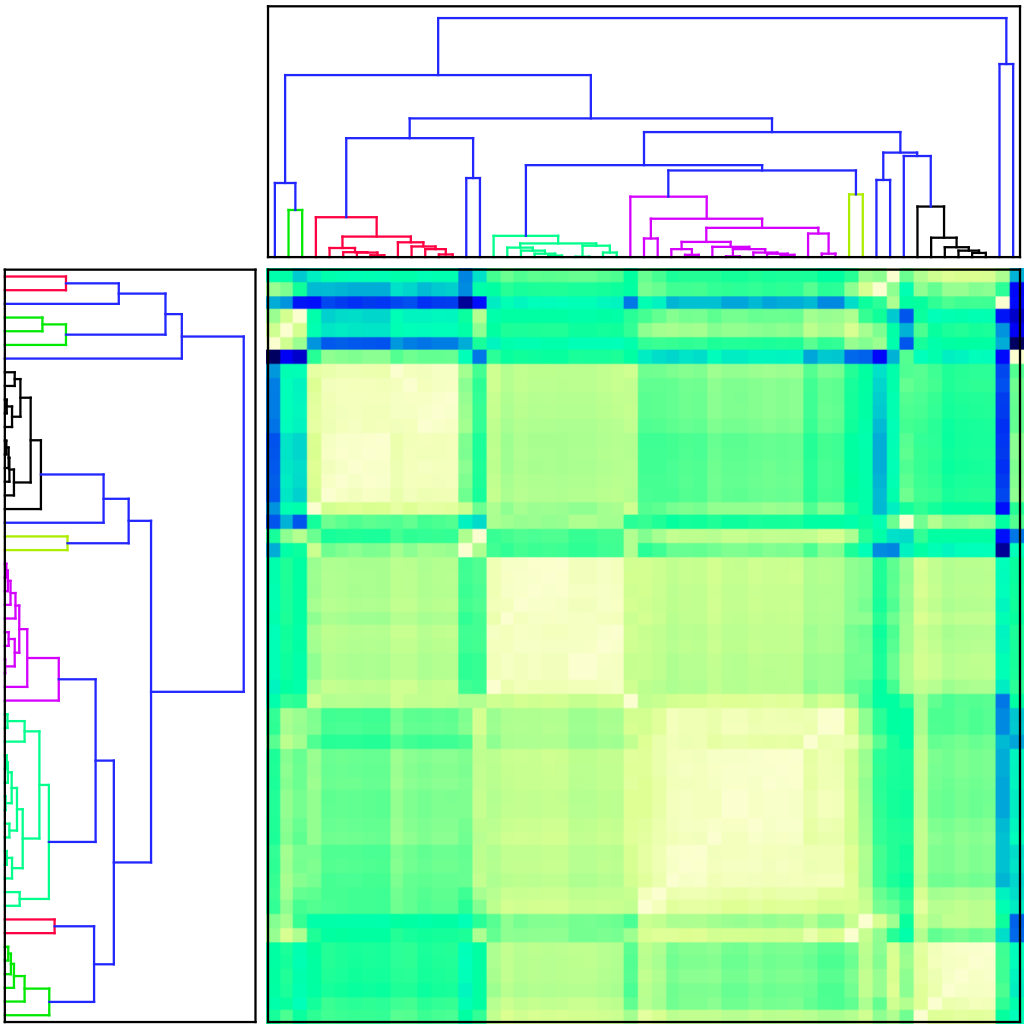
۳-۵-۲-خوشه‌بندی سلسله‌مراتبی

خوشه‌بندی سلسله‌مراتبی یک ابزارِ توان‌مند برای شناسایی ساختارهایی است که در ساختارهای بزرگ‌تر گنجانده شده‌اند. از سوی دیگر، کار کردن با خروجی نیازمند دقت زیاد است؛ زیرا در این رویکرد خوشه‌ها به دقت فن kmeans شناسایی نمی‌شوند. در زیر مثالی است که در آن یک دستگاهِ چندخوشه‌ای خلق می‌گردد. برای استفاده از تابع سلسله‌مراتبی، یک ماتریس فاصله می‌سازیم و خروجی، یک نمودار درختی خواهد بود. برای اینکه ببینید خوشه‌بندی سلسله‌مراتبی چگونه کار می‌کند به شکل ۳-۱٤ رجوع کنید.

```
import numpy as np
import matplotlib.pyplot as mpl
from mpl_toolkits.mplot3d import Axes3D
from scipy.spatial.distance import pdist, squareform
import scipy.cluster.hierarchy as hy
# Creating a cluster of clusters function
def clusters(number = 20, cnumber = 5, csize = 10):
# Note that the way the clusters are positioned is Gaussian randomness.
    rnum = np.random.rand(cnumber, 2)
```

```
    rn = rnum[:,0] * number
    rn = rn.astype(int)
    rn[np.where(rn < 5 )] = 5
    rn[np.where(rn > number/2. )] = round(number / 2., 0)
    ra = rnum[:,1] * 2.9
    ra[np.where(ra < 1.5)] = 1.5
    cls = np.random.randn(number, 3) * csize
    # Random multipliers for central point of cluster
    rxyz = np.random.randn(cnumber-1, 3)
    for i in range(cnumber-1):
        tmp = np.random.randn(rn[i+1], 3)
        x = tmp[:,0] + ( rxyz[i,0] * csize )
        y = tmp[:,1] + ( rxyz[i,1] * csize )
        z = tmp[:,2] + ( rxyz[i,2] * csize )
        tmp = np.column_stack([x,y,z])
        cls = np.vstack([cls,tmp])
    return cls

# Generate a cluster of clusters and distance matrix.
cls = clusters()
D = pdist(cls[:,0:2])
D = squareform(D)
# Compute and plot first dendrogram.
fig = mpl.figure(figsize=(8,8))
ax1 = fig.add_axes([0.09,0.1,0.2,0.6])
Y1 = hy.linkage(D, method='complete')
cutoff = 0.3 * np.max(Y1[:, 2])
Z1 = hy.dendrogram(Y1, orientation='right', color_threshold=cutoff)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
# Compute and plot second dendrogram.
ax2 = fig.add_axes([0.3,0.71,0.6,0.2])
Y2 = hy.linkage(D, method='average')
cutoff = 0.3 * np.max(Y2[:, 2])
Z2 = hy.dendrogram(Y2, color_threshold=cutoff)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
# Plot distance matrix.
ax3 = fig.add_axes([0.3,0.1,0.6,0.6])
idx1 = Z1['leaves']
idx2 = Z2['leaves']
D = D[idx1,:]
D = D[:,idx2]
ax3.matshow(D, aspect='auto', origin='lower', cmap=mpl.cm.YlGnBu)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)
# Plot colorbar.
fig.savefig('cluster_hy_f01.pdf', bbox = 'tight')
```



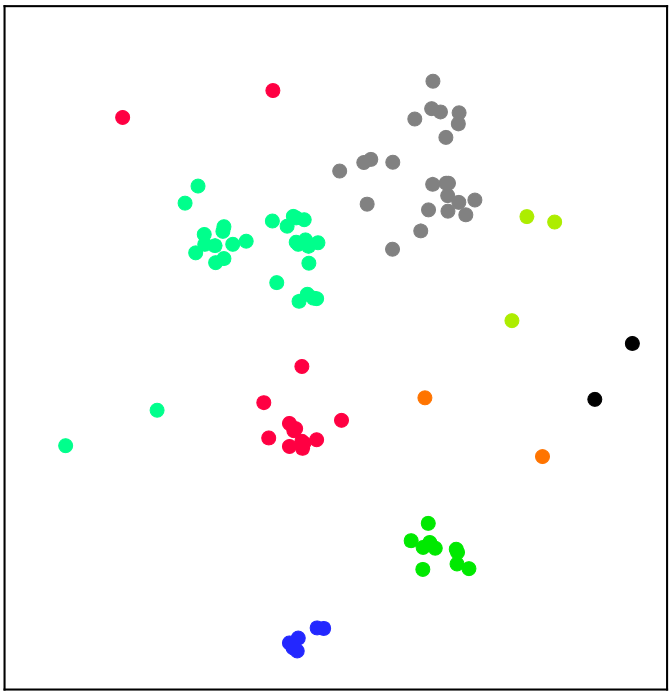
شکل ۳-۱٤: نمودار پیکسلی ماتریس فاصله بوده و دو نمودار درختی بیان‌گر شیوه‌های مختلف سلسله‌مراتبی است

با مشاهده‌ی ماتریسِ فاصله در شکل به همراه نمودارهای درختی، چگونگی شناسایی ساختارهای کوچک آشکار می‌گردد. پرسش این است که چگونه می‌توانیم ساختارها را از یکدیگر تمیز دهیم؟ در اینجا ما از یک تابع به نام `fcluster` استفاده می‌کنیم. این تابع اندیس‌های مربوط به هر خوشه را در یک آستانه‌ی مشخص، برمی‌گرداند. خروجی تابع `fcluster` به روش انتخابی برای محاسبه‌ی تابع پیوند (تک یا کامل) بستگی دارد. مقدار برش که به خوشه تخصیص داده می‌شود بعنوان ورودی دوم تابع `fcluster` قابل تعریف است. در تابع `dendrogram`، مقدار پیش‌فرضِ برش برابر `0.7*np.max(Y[:,2])` می‌باشد که ما در اینجا از همان مقدار با ضریب ۰/۳ استفاده خواهیم کرد.

```
# Same imports and cluster function from the previous example
# follow through here.
# Here we define a function to collect the coordinates of
# each point of the different clusters.
def group(data, index):
    number = np.unique(index)
    groups = []
```

```
    for i in number:
        groups.append(data[index == i])
    return groups
# Creating a cluster of clusters
cls = clusters()
# Calculating the linkage matrix
Y = hy.linkage(cls[:,0:2], method='complete')
# Here we use the fcluster function to pull out a
# collection of flat clusters from the hierarchical
# data structure. Note that we are using the same
# cutoff value as in the previous example for the dendrogram
# using the 'complete' method.
cutoff = 0.3 * np.max(Y[:, 2])
index = hy.fcluster(Y, cutoff, 'distance')
# Using the group function, we group points into their
# respective clusters.
groups = group(cls, index)
# Plotting clusters
fig = mpl.figure(figsize=(6, 6))
ax = fig.add_subplot(111)
colors = ['r', 'c', 'b', 'g', 'orange', 'k', 'y', 'gray']
for i, g in enumerate(groups):
    i = np.mod(i, len(colors))
    ax.scatter(g[:,0], g[:,1], c=colors[i], edgecolor='none', s=50)
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
fig.savefig('cluster_hy_f02.pdf', bbox = 'tight')
```

خوشه‌هایی که به صورت سلسله‌مراتبی شناسایی شده‌اند در شکل ۳-۱۵ نشان داده شده است.



شکل ۳-۱۵: خوشه‌های شناسایی شده

۳-۶-پردازش سیگنال و تصویر

SciPy این امکان را به ما می‌دهد تا فایل‌های تصویری مانند *JPEG* و *PNG* را بدون آنکه نگران ساختار فایل در تصاویر رنگی باشیم، بخوانیم و ویرایش کنیم. در زیر مثالی ساده از کار کردن با فایل‌های تصویری برای ساخت یک تصویر زیبا که از ایستگاه فضایی بین‌المللی (*ISS*) گرفته شده است را بررسی خواهیم کرد (شکل ۳-۱۶).

شکل ۳-۱۶: یک تصویر انباشته که از صدها عکس گرفته شده از ایستگاه فضایی بین‌المللی تشکیل شده است

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob
# Getting the list of files in the directory
files = glob('space/*.JPG')
# Opening up the first image for loop
im1 = imread(files[0]).astype(np.float32)
# Starting loop and continue co-adding new images
for i in range(1, len(files)):
    print(i)
    im1 += imread(files[i]).astype(np.float32)
# Saving img
imsave('stacked_image.jpg', im1)
```

تصاویر *JPG* در محیط پایتون آرایه‌های *NumPy* با مقدارهای `(426, 640, 3)` هستند که سه لایه به ترتیب شامل قرمز، سبز و آبی می‌باشد.

در تصویر انباشته‌ی اولیه، مشاهده‌ی دنباله‌ی ستاره‌ها در بالای زمین تقریبن ناممکن است. برای بهبود این اثر، مثال قبل را به گونه‌ای ویرایش می‌کنیم تا شکل ۳-۱۷ حاصل گردد.

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob
# This function allows us to place in the
# brightest pixels per x and y position between
# two images. It is similar to PIL's
# ImageChop.Lighter function.
def chop_lighter(image1, image2):
    s1 = np.sum(image1, axis=2)
    s2 = np.sum(image2, axis=2)
    index = s1 < s2
    image1[index, 0] = image2[index, 0]
    image1[index, 1] = image2[index, 1]
    image1[index, 2] = image2[index, 2]
    return image1
# Getting the list of files in the directory
files = glob('space/*.JPG')
# Opening up the first image for looping
im1 = imread(files[0]).astype(np.float32)
im2 = np.copy(im1)
# Starting loop
for i in range(1, len(files)):
    print(i)
    im = imread(files[i]).astype(np.float32)
    # Same before
    im1 += im
    # im2 shows star trails better
    im2 = chop_lighter(im2, im)
# Saving image with slight tweaking on the combination
# of the two images to show star trails with the
# co-added image.
imsave('stacked_image.jpg', im1/im1.max() + im2/im2.max()*0.2)
```

شکل ۳-۱۷: یک تصویر انباشته که از صدها عکس گرفته شده از ایستگاه فضایی بین‌المللی تشکیل شده است

اگر بخواهید بدون استفاده از *SciPy* با تصاویر کار کنید، هنگام دخیره سازی آنها بصورت تصویر، باید توجه بیشتر و دقیق‌تری نسبت به آرایش درست مقادیر در آرایه‌ها داشته باشید. اما *SciPy* به آسانی این کار را انجام داده و در عوض به ما اجازه می‌دهد تا توجه بیشتری روی پردازش تصویر و بدست آوردن نتایج دلخواه داشته باشیم.

۳-۷- ماتریس‌های تُنُک

با استفاده از *NumPy* می‌توانیم با سرعتِ قابلِ قبولی بر روی آرایه‌هایی با حدود 10^6 درایه، عملیات انجام دهیم. بسته به میزان حافظه‌ی رَمِ قابلِ درسترس، هنگامی که شمار درایه‌ها به 10^7 افزایش یابد، سرعت محاسبات کمتر شده و حافظه‌ی پایتون محدود می‌گردد. اما اگر لازم باشد تا با آرایه‌های بزرگتر، مثلن آرایه‌ای با 10^{10} عضو کار کنید، چاره چیست؟ اگر بیشتر اعضای این آرایه‌های بزرگ شامل درایه‌های صفر باشند، آنگاه در دسته‌ی ماتریس‌های تُنُک قرار خواهند گرفت. اگر با ماتریس‌های تُنُک به درستی رفتار شود، زمان محاسبات و حافظه‌ی مصرفی می‌تواند به طور چشم‌گیری کاهش یابد. مثال ساده‌ی زیر این موضوع را نشان می‌دهد.

شما می‌توانید بایت‌های اشغال شده توسط یک آرایه‌ی *NumPy* را با استفاده از تابع `nbytes` تعیین کنید. این دستور برای تعیین این که چه چیزی حافظه‌ی برنامه‌ی شما را بیشتر اشغال کرده است، می‌تواند مفید واقع شود. همین کار را می‌توان بر روی ماتریس‌های `sparse` نیز انجام داد. بدین منظور، از دستور `data.nbytes` استفاده می‌شود.

```
import numpy as np
from scipy.sparse.linalg import eigsh
```

```
from scipy.linalg import eigh
import scipy.sparse
import time
N = 3000
# Creating a random sparse matrix
m = scipy.sparse.rand(N, N)
# Creating an array clone of it
a = m.toarray()
print('The numpy array data size: ' + str(a.nbytes) + ' bytes')
print('The sparse matrix data size: ' + str(m.data.nbytes) + ' bytes')
# Non-sparse
t0 = time.time()
res1 = eigh(a)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Non-sparse operation takes ' + dt)
# Sparse
t0 = time.time()
res2 = eigsh(m)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Sparse operation takes ' + dt)
```

در اینجا، حافظه‌ی تخصیص داده شده به آرایه‌های *NumPy* و ماتریس تُنُک به ترتیب برابر ٦٨ مگابایت و ٠/٦٨ مگابایت بود. به همان نسبت، زمانِ مصرف شده برای اجرای دستورِ محاسبه‌ی مقدارها و بردارهای ویژه بر روی رایانه‌ی ما، برابـر ٣٦/٦ ثانیـه و ٠/٢ ثانیه بود. این یعنی ماتریس تُنک از دیدگاه حافظه، ١٠٠بار بهینه‌تر بوده و محاسبه‌ی مقادیر و بردارهای ویژه در حالت ماتریس تُنُک ١٥٠برابر سریع‌تر انجام می‌شود.

اگر با ماتریس‌های تُنُک آشنایی ندارید، پیشنهاد می‌کنیم این راهنما را مطالعـه کنیـد؛ کـه در آن مـوارد پایـه‌ای و اساسی در مورد ماتریس‌های تُنُک و عملیات بر روی آنها مورد بررسی قرار گرفته است.

در هندسه‌ی دوبُعدی و سه‌بُعدی، داده‌ها با ساختارِ تُنُک در زمینه‌های گوناگونی مانند مهندسی، دینامیـک سـیالات محاسـباتی، الکترومغناطیس، ترمودینامیک و صوت‌شناسی کاربرد پیدا می‌کنند. از نمونه‌هایِ غیرهندسیِ کاربردِ ماتریس‌هایِ تُنُک نیز می‌توان بـه بهینه‌سازی، مدل‌سازی اقتصادی، ریاضیات، آمار و نگره‌های گراف/شبکه اشاره کرد.

با استفاده از `scipy.io` می‌توانید فرمت‌های متداول ماتریس تُنُک مانند *Matrix Market*، *Harwell-Boeing* یا فایـل‌هـای *MATLAB* را فراخوانی و ویرایش کنید. این ویژگی می‌تواند برای به اشتراک گذاری، همکاری و تبادل داده‌ها با دیگـران سـودمند باشد. در بخش بعد، این قابلیت‌های `scipy.io` را بررسی می‌کنیم.

٣-٨- خواندن و نوشتن فایل‌ها فراتر از *SciPy*

NumPy مجموعه‌ی خوبی از قابلیت‌های خواندن و نوشتن فایل‌های *ASCII* فراهم آورده است. اگر می‌خواهید اطلاعات تنهـا در محیط پایتون قابل خواندن باشد، پشتیانی عالی از فرمت دودویی به شما این امکان را خواهد داد. اما در مورد دیگـر فایـل‌هـای

دودویی که فراگیرتر هستند چطور؟ اگر از متلب استفاده می‌کنید یا با افرادی که با آن کار می‌کنند همکاری دارید، همانطور که در بخش گذشته بطور خلاصه اشاره شد، می‌توانید به سادگی با استفاده از *NumPy* این فایل‌ها را خوانده و ویرایش کنید (با استفاده از دستورهای `scipy.io.loadmat` و `scipy.savemat`).

در رشته‌هایی مانند ستاره‌شناسی، جغرافیا و پزشکی یک زبان برنامه‌نویسی به نام *IDL* وجود دارد. این زبان فایل‌ها را به فرمت دودویی ذخیره می‌کند و با استفاده از بسته‌ی داخلی *NumPy* که با نام `scipy.io.readsav` قابل فراخوانی است، می‌توان فایل‌های مزبور را خواند. این ماژول، انعطاف‌پذیر و سریع است اما فاقد قابلیت‌های نوشتن می‌باشد.

در آخر، شما می‌توانید فایل‌های *Matrix Market* را خوانده و ویرایش کنید. این فایل‌ها، برای به اشتراک‌گذاری داده‌ها با ساختار ماتریسی که به فرمت *ASCII* تهیه شده‌اند، بسیار متداول هستند. این فرمت همچنین در دیگر زبان‌ها مانند *C*، فرترن ومتلب نیز به‌خوبی پشتیبانی می‌شود؛ بنابراین بدلیل فراگیر بودن و خوانایی مناسب، می‌تواند فرمت خوبی برای ذخیره‌سازی به شمار رود. گفتنی است که این فرمت برای ماتریس‌های تُنُک نیز مناسب می‌باشد.

فصل چهارم

SciKit: یک گام فراتر از *SciPy*

SciPy و *NumPy* ابزارهایی عالی هستند و بیشتر قابلیت‌های موردنیازمان را فراهم می‌کنند. با این وجود، برخی اوقات نیازمند ابزارهای پیشرفته‌تری هستیم و اینجا است که `scikits` وارد عمل می‌شود. `scikits` مجموعه‌ای است بسته‌ها است که کامل‌کننده‌ی *SciPy* به شمار می‌رود. در حال حاضر بیش از ۵۰ بسته‌ی `scikits` در دسترس است که لیستی از این‌ها را می‌توان در اینجا مشاهده کرد. در این بخش ما دو بسته‌ی متداول و فراگیر را بررسی خواهیم کرد: `Scikit-image`، یک ماژول تصویر کامل‌تر از `scipy.ndimage`؛ این ماژول یک ابزار پردازش تصویر برای *SciPy* می‌باشد. و `Scikit-learn` که یک بسته‌ی یادگیری ماشین است و می‌تواند برای مجموعه‌ای از اهداف علمی و مهندسی مورد بهره قرار گیرد.

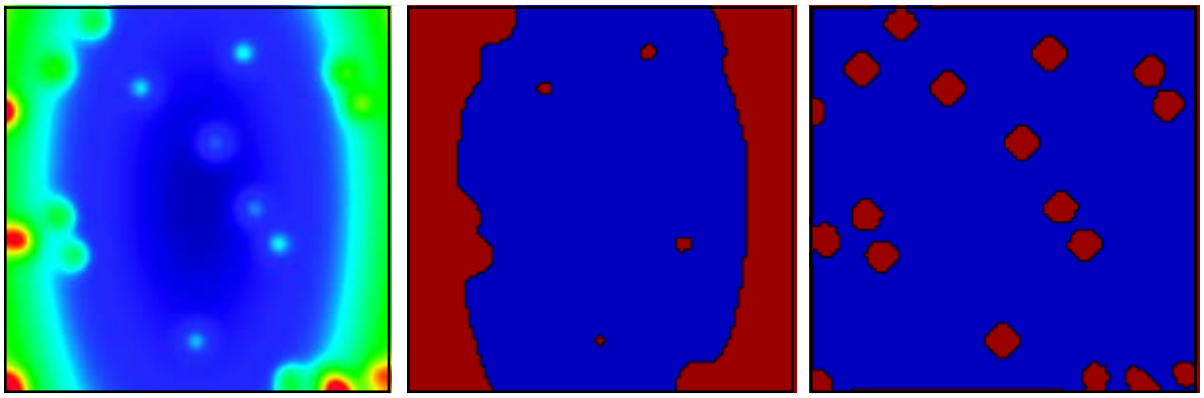
Scikit-Image **-۱-۴**

کلاس `ndimage` در *SciPy* ابزارهای سودمند بسیاری را برای پردازش چندبُعدی داده‌ها، مثل فیلترهای ساده (مانند هموارسازی گوسی)، تبدیل فوریه، ریخت‌شناسی (مانند فرسایش دودویی)، درون‌یابی و اندازه‌گیری فراهم می‌آورد. با استفاده از این تابع‌ها می‌توانیم برنامه‌هایی بنویسیم که عملیات پیچیده‌تری انجام دهند. خوشبختانه `scikit-image` این کار را برای ما انجام داده و توابع پیشرفته‌تر بیشتری را برای اهداف پژوهشی گرد هم آورده است. این ماژول‌های پیشرفته و سطحِ بالا مواردی مانند تبدیل فضای رنگ، الگوریتم‌هایِ تنظیمِ شدتِ تصویر، شناساییِ چهره، فیلترهایی برای تشدید و کاهش نویز، قابلیت‌های خواندن/نوشتن و... را در بر می‌گیرند.

۴-۱-۱- آستانه‌گذار پویا

یکی از کاربردهایِ متداولِ علمِ تصویر، بخش‌بندی کردن اجزای تصویر می‌باشد که به آن آستانه‌گذاری هم گفته می‌شود. فنِ سنتیِ آستانه‌گذاری هنگامی که پس‌زمینه‌ی تصویر صاف باشد، خوب کار می‌کند. متاسفانه، این حالت همواره رخ نمی‌دهد و پس‌زمینه در سرتاسر تصویر تغییر خواهد کرد. از این رو، فن‌های انطباق‌پذیر آستانه‌گذاری گسترش یافته‌اند و ما می‌توانیم در بسته‌ی `scikit-image` از آنها استفاده کنیم. در ادامه، یک تصویر با پس‌زمینه‌ی غیریکنواخت که دارای نقطه‌های تار با موقعیت تصادفی در سرتاسر آن است، تولید می‌کنیم. سپس برای جداسازی نقطه‌های تار از پس‌زمینه، یک تابعِ ساده‌یِ آستانه‌گذارِ انطباق‌پذیر بر روی تصویر اعمال می‌کنیم.

```
import numpy as np
import matplotlib.pyplot as mpl
import scipy.ndimage as ndimage
import skimage.filters as skif
# Generating data points with a non-uniform background
x = np.random.uniform(low=0, high=100, size=20).astype(int)
y = np.random.uniform(low=0, high=100, size=20).astype(int)
# Creating image with non-uniform background
func = lambda x, y: x**2 + y**2
grid_x, grid_y = np.mgrid[-1:1:100j, -2:2:100j]
bkg = func(grid_x, grid_y)
bkg = bkg / np.max(bkg)
# Creating points
clean = np.zeros((100,100))
clean[(x,y)] += 5
clean = ndimage.gaussian_filter(clean, 3)
clean = clean / np.max(clean)
# Combining both the non-uniform background
# and points
fimg = bkg + clean
fimg = fimg / np.max(fimg)
# Defining minimum neighboring size of objects
block_size = 3
# Adaptive threshold function which returns image
# map of structures that are different relative to
# background
adaptive_cut = skif.threshold_adaptive(fimg, block_size, offset=0)
# Global threshold
global_thresh = skif.threshold_otsu(fimg)
global_cut = fimg > global_thresh
# Creating figure to highlight difference between
# adaptive and global threshold methods
fig = mpl.figure(figsize=(8, 4))
fig.subplots_adjust(hspace=0.05, wspace=0.05)
ax1 = fig.add_subplot(131)
ax1.imshow(fimg)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax2 = fig.add_subplot(132)
ax2.imshow(global_cut)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax3 = fig.add_subplot(133)
ax3.imshow(adaptive_cut)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)
fig.savefig('scikit_image_f01.pdf', bbox_inches='tight')
```



شکل ٤-١: آستانه‌گذاری. تصویر اصلی تولید شده (چپ)، الگوریتم آستانه‌گذار سنتی (وسط)، الگوریتم آستانه‌گذار پویا (راست)

در این حالت، همان گونه که در شکل ٤-١ مشاهده می‌شود، آشکار است که فنِ آستانه‌گـذاریِ انطبـاق‌پـذیر (راسـت) بهتـر از روش سنتی (وسط) عمل می‌کند. باید دانست که بیشترین کُد نوشته شده در بالا مربوط به تولیـد تصـویر و رسـم خروجـی نتـایج است. کُدِ اصلی که برای آستانه‌گذاری انطباق‌پذیرِ تصویر به کار رفته تنها از دو خط تشکیل شده است.

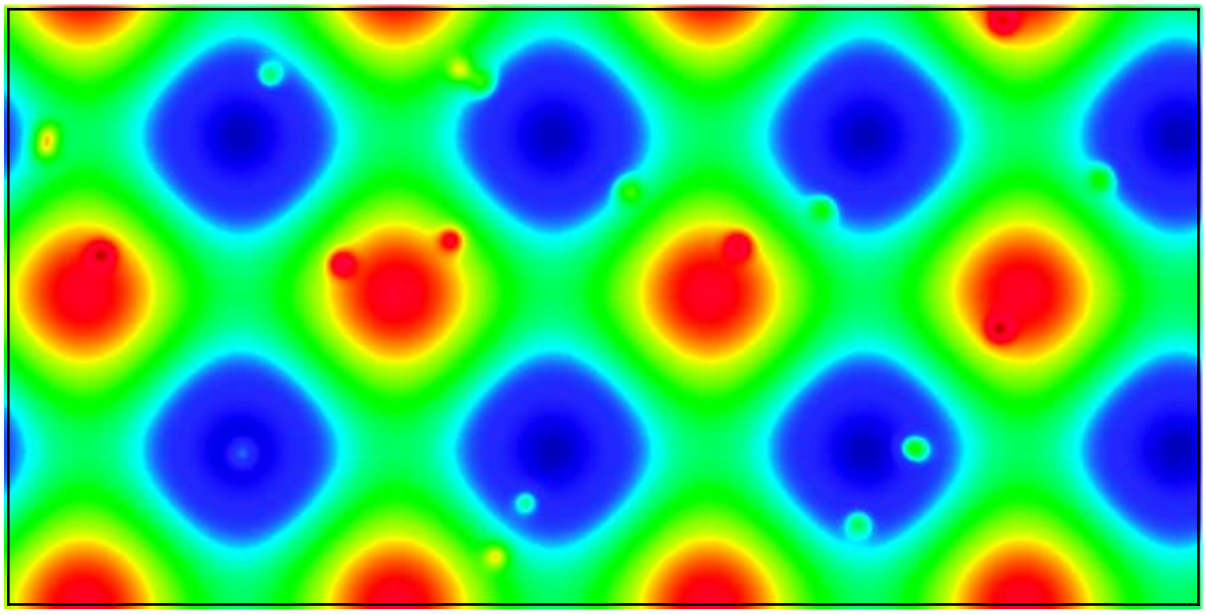
۴-۱-۲- بیشینه‌های محلی

اینجا همان مساله‌ی پیشین با اندکی تفاوت مورد بررسی قرار می‌گیرد: چگونه می‌توانیم نقاطی را که بر روی یک پـس‌زمینـه‌ی غیریکنواخـت واقـع شـده‌انـد را شناسـایی کـرده و مختصـات پیکسـلی آنهـا را بدسـت آوریـم؟ مـی‌تـوانیم از دسـتور `skimage.morphology.is_local_maximum`[٤] که تنها تصویر را بعنوان متغیر ورودی می‌گیرد، بهره بگیریم. ایـن تـابع بطـور شگفت‌انگیزی خوب کار می‌کند. شکل ٤-٢ را ببینید.

```
import numpy as np
import matplotlib.pyplot as mpl
import scipy.ndimage as ndimage
import skimage.morphology as morph
# Generating data points with a non-uniform background
x = np.random.uniform(low=0, high=200, size=20).astype(int)
y = np.random.uniform(low=0, high=400, size=20).astype(int)
# Creating image with non-uniform background
func = lambda x, y: np.cos(x)+ np.sin(y)
grid_x, grid_y = np.mgrid[0:12:200j, 0:24:400j]
bkg = func(grid_x, grid_y)
bkg = bkg / np.max(bkg)
# Creating points
clean = np.zeros((200,400))
clean[(x,y)] += 5
clean = ndimage.gaussian_filter(clean, 3)
```

[٤] این تابع در نسخه‌ی جدید `scikit` حذف شده است. به جای آن، تابع `skimage.feature.peak_local_max` قابل استفاده است – مترجم

```
clean = clean / np.max(clean)
# Combining both the non-uniform background
# and points
fimg = bkg + clean
fimg = fimg / np.max(fimg)
# Calculating local maxima
lm1 = morph.is_local_maximum(fimg)
x1, y1 = np.where(lm1.T == True)
# Creating figure to show local maximum detection
# rate success
fig = mpl.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
ax.imshow(fimg)
ax.scatter(x1, y1, s=100, facecolor='none', edgecolor='#009999')
ax.set_xlim(0,400)
ax.set_ylim(0,200)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
fig.savefig('scikit_image_f02.pdf', bbox_inches='tight')
```

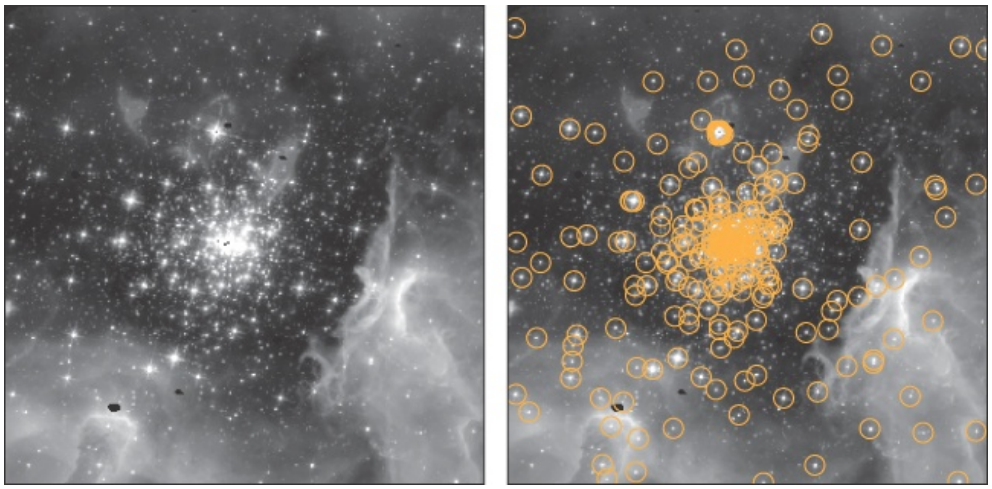


شکل ٤-٢: بیشینه‌های محلی شناسایی شده (دایره‌ها)

اگر به دقت به شکل نگاه کنید، متوجه خواهید شد بیشینه‌هایی شناسایی شده‌اند که به چشمه‌های تار اشاره نمی‌کند و به جای آن، در قله‌های پس‌زمینه واقع شده‌اند. این قله‌ها دردسرساز هستند؛ اما بنا به تعریف، تابع `skimage.morphology.is_local_maximum` آنها را پیدا خواهد کرد. چگونه می‌توانیم این موارد اشتباه را حذف کنیم؟ از آنجا که مختصات بیشینه‌های محلی در دست می‌باشد، می‌توانیم به دنبال ویژگی‌هایی بگردیم که چشمه‌ها را از بقیه متمایز می‌کنند. پس‌زمینه در مقایسه با چشمه‌ها نسبتن نَرم و هموار است؛ بنابراین می‌توانیم با استفاده از انحراف معیار استاندارد قله‌ها نسبت پیکسل‌های مجاور محلی‌شان، آن‌ها را متمایز کنیم.

اما بسته‌ی `scikit-image` چگونه با مسائلِ پژوهشیِ دنیایِ واقعی روبرو می‌شود؟ در ستاره‌شناسی، شارِ واحدِ سطحِ دریافتی از ستارگان را می‌توان بوسیله‌ی تصاویر و با سنجش سطوحِ شدتِ در محلِ آنها اندازه‌گیری کرد؛ پروسه‌ای که به آن شیدسنجی (فوتومتری) گفته می‌شود. شیدسنجی در چندین زبان برنامه‌نویسی انجام شده است اما هنوز بسته‌ای عملی در این زمینه برای پایتون وجود ندارد. اولین گام در شیدسنجی، شناسایی ستارگان است. در مثال بعدی، از تابع `is_local_maximum` برای شناسایی چشمه‌ها (ستارگان) در یک خوشه‌ی ستاره‌ای به نام *NGC 3603* بهره خواهیم گرفت. این خوشه توسط تلسکوپ فضایی هابل مشاهده شد. خاطر نشان می‌کند که از یک بسته‌ی دیگر به نام *PyFITS* در اینجا استفاده شده است. *PyFITS* یک بسته‌ی استاندارد ستاره‌شناسی برای فراخوانی داده‌های دودویی که با فرمت *FITS* ذخیره شده‌اند، می‌باشد.

```
import numpy as np
import pyfits
import matplotlib.pyplot as mpl
import skimage.morphology as morph
import skimage.exposure as skie
# Loading astronomy image from an infrared space telescope
img = pyfits.getdata('stellar_cluster.fits')[500:1500, 500:1500]
# Prep file scikit-image environment and plotting
limg = np.arcsinh(img)
limg = limg / limg.max()
low = np.percentile(limg, 0.25)
high = np.percentile(limg, 99.5)
opt_img = skie.exposure.rescale_intensity(limg, in_range=(low, high))
# Calculating local maxima and filtering out noise
lm = morph.is_local_maximum(limg)
x1, y1 = np.where(lm.T == True)
v = limg[(y1, x1)]
lim = 0.5
x2, y2 = x1[v > lim], y1[v > lim]
# Creating figure to show local maximum detection
# rate success
fig = mpl.figure(figsize=(8,4))
fig.subplots_adjust(hspace=0.05, wspace=0.05)
ax1 = fig.add_subplot(121)
ax1.imshow(opt_img)
ax1.set_xlim(0, img.shape[1])
ax1.set_ylim(0, img.shape[0])
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax2 = fig.add_subplot(122)
ax2.imshow(opt_img)
ax2.scatter(x2, y2, s=80, facecolor='none', edgecolor='#FF7400')
ax2.set_xlim(0, img.shape[1])
ax2.set_ylim(0, img.shape[0])
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
fig.savefig('scikit_image_f03.pdf', bbox_inches='tight')
```



شکل ٤-۳: ستاره‌ها (دایره‌های نارنجی) در تصویر تلسکوپ فضایی هابل که توسط تابع `is_local_maxima` شناسایی شده‌اند

تابع `skimage.morphology.is_local_maximum` بالغ بر ۳۰۰۰۰ بیشینه‌ی محلی در این تصویر یافته که شماری از آنها نادرست انتخاب شده‌اند. یک مقدار آستانه‌گذار برای حذف هر گونه قله‌ای که مقدار پیکسلی‌اش زیر ۰/۵ است (در تصویر هم‌پایه شده) اعمال می‌کنیم تا مقدار بیشینه‌های محلی تا حدود ۲۰۰ عدد کاهش یابد. راه‌های بهتری برای فیلتر کردن بیشینه‌های غیرستاره‌ای (نویزها) وجود دارد اما ما برای سادگی با روش کنونی کار خواهیم کرد. بر پایه‌ی شکل ٤-۳، می‌توان مشاهده کرد که یافته‌ها روی‌هم‌رفته مناسب هستند. پس از اینکه محل ستاره‌ها را شناسایی کردیم، می‌توانیم الگوریتم‌هایِ سنجشِ شار را اعمال کنیم که البته این موضوع فراتر از بحث این فصل می‌باشد.

۴-۲- *Scikit-Learn*

احتمالن گسترده‌ترین بسته‌ی `scikit`، `scikit-learn` است. این بسته یک مجموعه‌یِ ساده‌یِ یادگیری ماشین بوده و شامل ابزارهایی مرتبط با یادگیری با نظارت و یادگیری بدونِ نظارت می‌باشد. ممکن است بپرسید: «یادگیری ماشین چه کاری را می‌تواند برایم انجام دهد که قبلن نمی‌توانستم؟!». پاسخ یک واژه است: پیش‌بینی. اجازه بدهید فرض کنیم با مساله‌ای روبرو شده‌ایم که در آن نمونه‌ای از داده‌های آزمایشگاهی را در اختیارمان قرار داده‌اند. آیا می‌توان در مورد آنها پیش‌بینی کرد؟ برای یافتن پاسخ، تلاش خواهیم کرد که یک مدل تحلیلی برای توصیف داده‌ها خلق کنیم؛ اگر چه به دلیل وابستگی‌های پیچیده، این کار همواره عملی نیست. اما چه می‌شد اگر می‌توانستید داده‌ها را وارد ماشین کرده، به ماشین آموزش دهید که چه چیزی در مورد داده‌ها خوب است و چه چیزی بد، و در نهایت به آن اجازه می‌دادید تا پیش‌بینی‌ها را خودش انجام دهد؟ این همان چیزی است که به عنوان یادگیری ماشین شناخته می‌شود و اگر از آن به درستی استفاده شود، می‌تواند بسیار قدرتمند باشد.

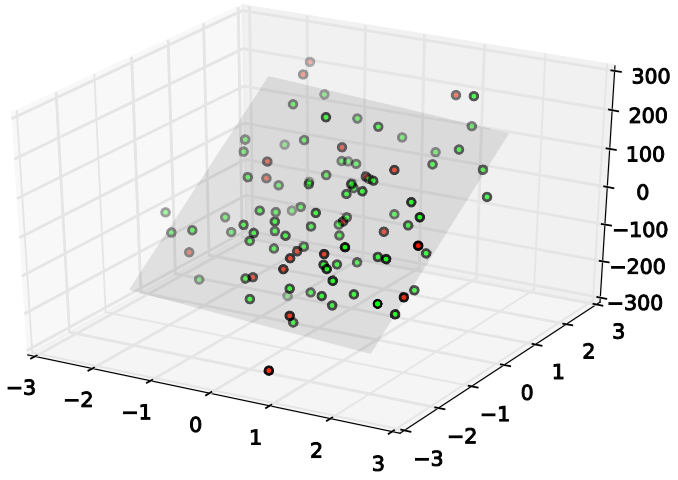
`scikit-learn` یک بسته‌ی فوق‌العاده بوده و راهنمای آن پربار و به خوبی سامان‌دهی شده است. در ادامه می‌خواهیم مثال‌هایی که در بخش‌های پیشین بررسی کرده‌ایم را بازخوانی کنیم و ببینیم آیا `scikit-learn` می‌تواند پاسخ‌هایی بهتر ارائه دهد یا خیر.

۴-۲-۱- رگرسیون خطی

ما در فصل ۳ یک خط را بر مجموعه‌ای از داده‌ها برازش دادیم که در واقع بیان‌گرِ یک مساله‌ی رگرسیونِ خطی بود. اگر با داده‌هایی سروکار داشته باشیم که شمار بُعدهای آن بالاتر باشد، چگونه می‌توان به پاسخِ رگرسیونِ خطی دست یافت؟ scikit-learn ابزارهای زیادی مانند رگرسیونِ لاسو و ریج برای انجام این کار دارد. در اینجا ما تابعِ رگرسیونِ کمینه‌ی مربعاتِ معمولی که مسایل ریاضی به فرمِ زیر را حل می‌کند، بررسی خواهیم کرد:

$$\min_{w} \|X\beta - y\| \qquad (۱-۳)$$

که w مجموعه ضرایب است. تعداد ضرایب به شمار ابعاد مساله $N=MD-1$ بستگی دارد که $M>1$ بوده و عددی صحیح است. در مثال زیر، رگرسیون خطی یک صفحه در فضای سه‌بُعدی را محاسبه می‌کنیم؛ از این رو شمار ضرایب ۲ خواهد بود. در ادامه نشان می‌دهیم که چگونه با استفاده از دستور LinearRegression، مدل را با داده‌ها آموزش داده، بهترین پاسخ را تقریب زده، یک پیش‌بینی از داده‌ها داشته و هم‌خوانی دیگر داده‌ها با مدل را تست کنید. یک خروجی تصویری از رگرسیون خطی در شکل ٤-٤ نشان داده شده است.



شکل ٤-٤: نتایج رگرسیون خطی در فضای سه‌بُعدی با scikit-learn

```
import numpy as np
import matplotlib.pyplot as mpl
from mpl_toolkits.mplot3d import Axes3D
from sklearn import linear_model
from sklearn.datasets.samples_generator import make_regression
# Generating synthetic data for training and testing
X, y = make_regression(n_samples=100, n_features=2, n_informative=1,\
random_state=0, noise=50)
# X and y are values for 3D space. We first need to train
# the machine, so we split X and y into X_train, X_test,
# y_train, and y_test. The *_train data will be given to the
# model to train it.
X_train, X_test = X[:80], X[-20:]
y_train, y_test = y[:80], y[-20:]
```

```
# Creating instance of model
regr = linear_model.LinearRegression()
# Training the model
regr.fit(X_train, y_train)
# Printing the coefficients
print(regr.coef_)
# [-10.25691752 90.5463984 ]
# Predicting y-value based on training
X1 = np.array([1.2, 4])
print(regr.predict(X1))
# 350.860363861
# With the *_test data we can see how the result matches
# the data the model was trained with.
# It should be a good match as the *_train and *_test
# data come from the same sample. Output: 1 is perfect
# prediction and anything lower is worse.
print(regr.score(X_test, y_test))
# 0.949827492261
fig = mpl.figure(figsize=(8, 5))
ax = fig.add_subplot(111, projection='3d')
# ax = Axes3D(fig)
# Data
ax.scatter(X_train[:,0], X_train[:,1], y_train, facecolor='#00CC00')
ax.scatter(X_test[:,0], X_test[:,1], y_test, facecolor='#FF7800')
# Function with coefficient variables
coef = regr.coef_
line = lambda x1, x2: coef[0] * x1 + coef[1] * x2
grid_x1, grid_x2 = np.mgrid[-2:2:10j, -2:2:10j]
ax.plot_surface(grid_x1, grid_x2, line(grid_x1, grid_x2),
alpha=0.1, color='k')
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
ax.zaxis.set_visible(False)
fig.savefig('scikit_learn_regression.pdf', bbox='tight')
```

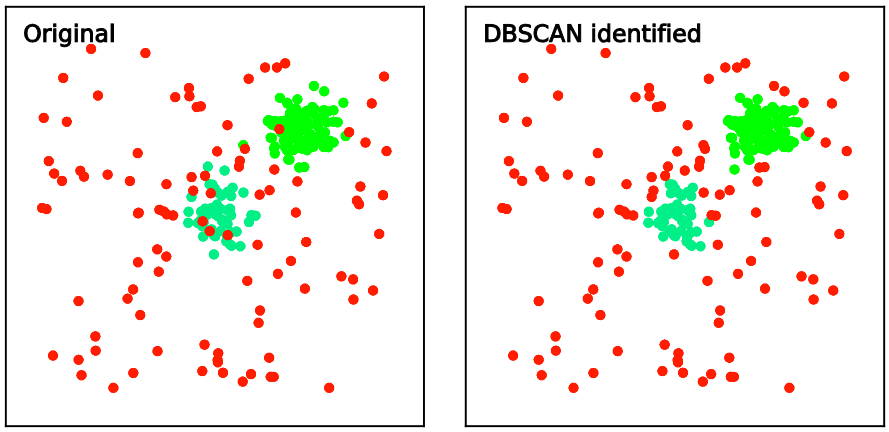
تابع `LinearRegression` در ابعاد بالاتر هم عملکرد مناسبی دارد. از این رو، کار کردن با شمار زیادی از ورودی‌ها در یک مدل، ساده خواهد بود. توصیه می‌شود که نیم‌نگاهی به مدل‌های دیگر رگرسیون خطی داشته باشید؛ چرا که ممکن است برای داده‌های شما مناسب‌تر باشد.

۴-۲-۲-خوشه‌بندی

SciPy دو بسته برای تحلیل خوشه‌بندی با کوانتش برداری (`kmeans`) و خوشه‌بندی سلسله‌مراتبی دارد. در این میان، روش `kmeans` برای بخش‌بندی داده‌ها به اجزای گوناگون بر اساس ویژگی‌های فضایی، ساده‌تر و اجرایی‌تر است. `Scikit-learn` شامل مجموعه‌ای از ابزارها برای تحلیل خوشه‌بندی است که فراتر از *SciPy* هستند. برای مقایسه‌ی مناسب‌تر با تابع `kmeans` در *SciPy*؛ از الگوریتم *DBSCAN* در مثال بعدی استفاده می‌شود. این الگوریتم، نقاط مرکزی را به گونه‌ای پیدا می‌کند که در یک شعاع معلوم نقاط داده‌ی زیادی داشته باشند. پس از اینکه نقطه‌ی مرکزی تعریف شد، فرآیند بصورت تکراری تا زمانی ادامه پیدا

می‌کند که دیگر نقطه‌ی مرکزی قابل تعریفی در محدوده‌ی شعاعِ بیشینه وجود نداشته باشد. الگوریتم مزبور، هنگامی کـه در داده‌هـا نویز وجود داشته باشد، در مقایسه با kmeans فوق‌العاده عمل می‌کند.

```
import numpy as np
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
# Creating data
c1 = np.random.randn(100, 2) + 5
c2 = np.random.randn(50, 2)
# Creating a uniformly distributed background
u1 = np.random.uniform(low=-10, high=10, size=100)
u2 = np.random.uniform(low=-10, high=10, size=100)
c3 = np.column_stack([u1, u2])
# Pooling all the data into one 150 x 2 array
data = np.vstack([c1, c2, c3])
# Calculating the cluster with DBSCAN function.
# db.labels_ is an array with identifiers to the
# different clusters in the data.
db = DBSCAN(eps=0.95, min_samples=10).fit(data)
labels = db.labels_
# Retrieving coordinates for points in each
# identified core. There are two clusters
# denoted as 0 and 1 and the noise is denoted
# as -1. Here we split the data based on which
# component they belong to.
dbc1 = data[labels == 0]
dbc2 = data[labels == 1]
noise = data[labels == -1]
# Setting up plot details
x1, x2 = -12, 12
y1, y2 = -12, 12
fig = plt.figure()
fig.subplots_adjust(hspace=0.1, wspace=0.1)
ax1 = fig.add_subplot(121, aspect='equal')
ax1.scatter(c1[:, 0], c1[:, 1], lw=0.5, color='#00CC00')
ax1.scatter(c2[:, 0], c2[:, 1], lw=0.5, color='#028E9B')
ax1.scatter(c3[:, 0], c3[:, 1], lw=0.5, color='#FF7800')
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_xlim(x1, x2)
ax1.set_ylim(y1, y2)
ax1.text(-11, 10, 'Original')
ax2 = fig.add_subplot(122, aspect='equal')
ax2.scatter(dbc1[:, 0], dbc1[:, 1], lw=0.5, color='#00CC00')
ax2.scatter(dbc2[:, 0], dbc2[:, 1], lw=0.5, color='#028E9B')
ax2.scatter(noise[:, 0], noise[:, 1], lw=0.5, color='#FF7800')
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_xlim(x1, x2)
ax2.set_ylim(y1, y2)
ax2.text(-11, 10, 'DBSCAN identified')
fig.savefig('scikits_422_ex1.pdf', bbox_inches='tight')
```



شکل ٤-۵: مثالی از اینکه چگونه الگوریتم *DBSCAN* از بسته‌ی کوانتش برداری بهتر عمل می‌کند. نقاط با توزیع یکنواخت بعنوان عضوهای خوشه در نظر گرفته نشده‌اند

تقریبن همه‌ی نقاط داده که در ابتدا بعنوان بخشی از خوشه تعریف شده بودند، حفظ شده و نقاط داده‌ی پس‌زمینه‌ی نویزی نیز حذف شدند (شکل ٤-۵). این رویداد بر برتری *DBSCAN* نسبت به روش `kmeans` تاکید می‌کند؛ بویژه در مواردی که داده‌هایی در نمونه حضور دارند که نباید بخشی از خوشه باشند. آشکار است که این حالت به ویژگی‌هایِ فضاییِ توزیعِ داده شده بستگی خواهد داشت.

فصل پنجم

سخن پایانی

۵-۱- چکیده

هدف این کتاب، آشنا کردن شما با بسته‌های *NumPy* و *SciPy* بود؛ به گونه‌ای که بتوانید از ابزارهای آن برای کارهای پژوهشی خودتان استفاده کنید. راهنمای آنلاین *NumPy* و *SciPy* بسیار جامع بوده پیدا کردن آنچه که از این بسته‌ها می‌خواهید ممکن است زمان‌بر باشد. البته که همه‌ی ما می‌خواهیم در کمترین زمان ممکن با این ابزارها آشنا شده و از آنها بهره بگیریم. امیدواریم که این کتاب توانسته باشد این کار را برای شما انجام دهد.

ما در این کتاب چگونگی استفاده از آرایه‌های *NumPy* برای اندیس‌گذاری، عملیات ریاضی و خواندن و نوشتن داده‌ها را پوشش دادیم. در *SciPy*، ابزارهایی که برای پژوهش علمی مناسب بودند را بازخوانی کردیم؛ مانند بهینه‌سازی، درونیابی، تابع‌اولیه‌گیری، خوشه‌بندی، آمار و مانند اینها.

در آخر، با دو تا از توان‌مندترین بسته‌های *scikit* آشنا شدیم. *Scikit-image* بسته‌ی قدرتمندی است که قابلیت‌های تصویری فراتر از *SciPy* را فراهم می‌آورد. با استفاده از *scikit-learn* نشان دادیم که چگونه می‌توان با یادگیری ماشین مسائل دشوار و پیچیده را حل کرد.

۵-۲-گام بعدی

اکنون شما با *SciPy* و *NumPy* و دو بسته‌ی *scikit* آشنا هستید. توابع و ابزارهایی که در این کتاب پوشش داده شد به شما این اجازه را می‌دهد تا به راحتی و با اطمینان بیشتر، مطالعات پژوهشی خود را دنبال کنید. افزون بر این، با بهره‌گیری از این ابزارها احتمالن راه‌های نوینی را برای حل مساله خواهید یافت که پیش از این از آنها مطلع نبودید. اگر به دنبال قابلیت‌های بیشتری هستید (مثلن انتگرال‌های نامعین) باید بسته‌های دیگر را بررسی کنید. یک منبع آنلاین خوب شامل هزاران بسته‌ی ثبت شده، تارنمای *PyPI* است که می‌توانید با یک جستجوی ساده در آن، چیزی را که می‌خواهید پیدا کنید.

همچنین، عضویت در لیستِ ایمیل‌هایی که مرتبط با زمینه‌ی پژوهشی خودتان است می‌تواند ایده‌ی خوبی باشد. در آنجا، بررسی گفتگوها و تبادل نظر بین کاربران پایتون و یا مطرح کردن پرسش‌هایتان، می‌تواند مسیر را برای یافتن آنچه که نیاز دارید

هموار کند. یکی دیگر از منابع خوب و مناسب دیگر برای این کار، تارنمای *stackoverflow.com* است. این تارنما، قطب اصلی پرسش و پاسخ و ارائه‌ی راهکار برای مسائل مرتبط با برنامه‌نویسی به شمار می‌رود.

۶ پیوست

کدهای پایتون

این بخش در برگیرنده‌ی کدهایی است که در این کتاب مورد استفاده قرار گرفته است.

- **مثال ۱**

```
import timeit
import numpy as np

# Create an array with 10^7 elements.
arr = np.arange(1e7)

# Converting ndarray to list
larr = arr.tolist()

# Lists cannot by default broadcast,
# so a function is coded to emulate
# what an ndarray can do.


def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist

# Number of tries
N = 10


# We are not using IPython's magic timeit command here. This enables you to
# run the script in as a script.
# NumPy array broadcasting
time1 = timeit.timeit('arr * 1.1', 'from __main__ import arr', number=N) / N
print(time1)

# List and custom function for broadcasting
time2 = timeit.timeit('list_times(larr, 1.1)',
       'from __main__ import list_times, larr', number=N) / N
print(time2)
```

- **مثال ۲**

```
import timeit
import numpy as np

# Create an array with 10^7 elements.
arr = np.arange(1e7)

# Converting ndarray to list
larr = arr.tolist()

# Lists cannot by default broadcast,
# so a function is coded to emulate
# what an ndarray can do.


def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist

# Number of tries
N = 10


# We are not using IPython's magic timeit command here. This enables you to
# run the script in as a script.
# NumPy array broadcasting
time1 = timeit.timeit('arr * 1.1', 'from __main__ import arr', number=N) / N
print(time1)

# List and custom function for broadcasting
time2 = timeit.timeit('list_times(larr, 1.1)',
       'from __main__ import list_times, larr', number=N) / N
print(time2)
```

- **مثال ۳**

```
# Creating an array of zeros and defining column types
import numpy as np

recarr = np.zeros((2,), dtype=('i4,f4,a10'))
toadd = [(1, 2., "Hello"), (2, 3., "World")]
recarr[:] = toadd

# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))

# Now creating the columns we want to put
# in the recarray
col1 = np.arange(2) + 1
col2 = np.arange(2, dtype=np.float32)
col3 = ['Hello', 'World']

# Here we create a list of tuples that is
# identical to the previous toadd list.
```

```
toadd = zip(col1, col2, col3)

# Assigning values to recarr
recarr[:] =list(toadd)

# Assigning names to each column, which
# are now by default called 'f0', 'f1', and 'f2'.

recarr.dtype.names = ('Integers', 'Floats', 'Strings')

# If we want to access one of the columns by its name, we
# can do the following.

print(recarr['Integers'])
#[1, 2]
```

- **مثال ۴**

```
import numpy as np

alist = [[1, 2], [3, 4]]

# To return the (0,1) element we must index as shown below.
alist[0][1]

# Converting the list defined above into an array
arr = np.array(alist)

# To return the (0,1) element we use ...
arr[0, 1]

# Now to access the last column, we simply use ...
arr[:, 1]

# Accessing the columns is achieved in the same way,
# which is the bottom row.
arr[1, :]

# Creating an array
arr = np.arange(5)

# Creating the index array
index = np.where(arr > 2)
print(index)

# Creating the desired array
new_arr = arr[index]

# We use the previous array
new_arr = np.delete(arr, index)

index = arr > 2
print(index)
new_arr = arr[index]
```

- **مثال ۵**

```
import numpy as np
import matplotlib.pyplot as plt

# First let us create an image
img1 = np.zeros((20, 20)) + 3
img1[4:-4, 4:-4] = 6
img1[7:-7, 7:-7] = 9
# See subplot A

# Let's filter out all values larger than 2 and less than 6
index1 = img1 > 2
index2 = img1 < 6
compound_index = index1 & index2

# The compound statement can alternatively be written as
compound_index = (img1 > 3) & (img1 < 7)
img2 = np.copy(img1)
img2[compound_index] = 0
# See plot B

# Making the boolean arrays even more complex we can do
index3 = img1 == 9
index4 = (index1 & index2) | index3
img3 = np.copy(img1)
img3[index4] = 0
# See subplot C

fig = plt.figure(313, figsize=(6, 6))
fig.subplots_adjust(hspace=None, wspace=None)
ax1 = fig.add_subplot(131)
ax1.imshow(img1, origin='lower left', interpolation='nearest',
           vmin=0, vmax=10, cmap=plt.cm.Greens)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_title('Plot A')

ax2 = fig.add_subplot(132)
ax2.imshow(img2, origin='lower left', interpolation='nearest',
           vmin=0, vmax=10, cmap=plt.cm.Greens)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_title('Plot B')

ax3 = fig.add_subplot(133)
ax3.imshow(img3, origin='lower left', interpolation='nearest',
           vmin=0, vmax=10, cmap=plt.cm.Greens)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)
ax3.set_title('Plot C')
fig.savefig('ch212_f01.pdf', bbox_inches='tight')
plt.close(fig)
```

- **مثال ۶**

```
import numpy.random as rand

# Creating a 100-element array with random values
# from a standard normal distribution or, in other
# words, a Gaussian distribution.
# The sigma is 1 and the mean is 0.
a = rand.randn(100)

# Here we generate an index for filtering
# out undesired elements.
index = a > 0.2
b = a[index]

# We execute some operation on the desired elements.
b = b ** 2 - 2

# Then we put the modified elements back into the
# original array.
a[index] = b
```

- **مثال ۷**

```
# Note, example is modified to allow execution without errors.
# Make sure there is a file called 'somefile.txt' in the same
# directory as this Python script.


# Opening the text file with the 'r' option,
# which only allows reading capability
f = open('somefile.txt', 'r')

# Parsing the file and splitting each line,
# which creates a list where each element of
# it is one line
alist = f.readlines()

# Closing file
f.close()

# Noted changed from example in book
alist.append(str(1))

# After a few operations, we open a new text file
# to write the data with the 'w' option. If there
# was data already existing in the file, it will be overwritten.
f = open('newtextfile.txt', 'w')

# Writing data to file
f.writelines(alist)
```

```
# Closing file
f.close()
```

- **مثال ۸**

```
import numpy as np

# Loading and existing file
arr = np.loadtxt('somefile.txt')

# Saving a new file
np.savetxt('somenewfile.txt', arr)

# Opening an existing file with the append option
f = open('existingfile.txt', 'a')

# Creating some random data to append to the existing file
data2append = np.random.rand(100)

# With np.savetxt we replace the file name with the file handle.
np.savetxt(f, data2append)

f.close()
```

- **مثال ۹**

```
import numpy as np
# example.txt file looks like the following
#
# XR21 32.789 1
# XR22 33.091 2

table = np.loadtxt('example.txt',
             dtype={'names': ('ID', 'Result', 'Type'),
             'formats': ('S4', 'f4', 'i2')})

# array([('XR21', 32.78900146484375, 1),
#        ('XR22', 33.090999603271484, 2)],
#  dtype=[('ID', '|S4'), ('Result', '<f4'), ('Type', '<i2')])
```

- **مثال ۱۰**

```
import numpy as np

# Creating a large array
data = np.empty((1000, 1000))
```

```
# Saving the array with numpy.save
np.save('test.npy', data)

# If space is an issue for large files, then
# use numpy.savez instead. It is slower than
# numpy.save because it compresses the binary
# file.
np.savez('test.npz', data)

# Loading the data array
newdata = np.load('test.npz')
```

- **مثال ۱۱**

```
import numpy as np

# Defining the matrices
A = np.matrix([[3, 6, -5],
               [1, -3, 2],
               [5, -1, 4]])

B = np.matrix([[12],
               [-2],
               [10]])

# Solving for the variables, where we invert A
X = A ** (-1) * B
print(X)

# matrix([[ 1.75],
#   [ 1.75],
#   [ 0.75]])


a = np.array([[3, 6, -5],
              [1, -3, 2],
              [5, -1, 4]])

# Defining the array
b = np.array([12, -2, 10])

# Solving for the variables, where we invert A
x = np.linalg.inv(a).dot(b)
print(x)
# array([ 1.75,  1.75,  0.75])
```

- **مثال ۱۲**

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
```

```
# Let's create a function to model and create data
def func(x, a, b):
    return a * x + b

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 2)

# Adding noise to the data
yn = y + 0.9 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)

#popt returns the best fit values for parameters of the given model (func)
print(popt)

# Plot out the current state of the data and model
ym = func(x, popt[0], popt[1])
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y, c='k', label='Function')
ax.scatter(x, yn)
ax.plot(x, ym, c='r', label='Best fit')
ax.legend(loc='upper left')
fig.savefig('scipy_311_ex1.pdf', bbox_inches='tight')
```

- **مثال ۱۳**

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Let's create a function to model and create data
def func(x, a, b, c):
    return a * np.exp(-(x - b) ** 2 / (2 * c ** 2))

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 5, 2)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)

#popt returns the best fit values for parameters of the given model (func)
print(popt)

# Plot out the current state of the data and model
ym = func(x, popt[0], popt[1], popt[2])
fig = plt.figure()
```

```
ax = fig.add_subplot(111)
ax.plot(x, y, c='k', label='Function')
ax.scatter(x, yn)
ax.plot(x, ym, c='r', label='Best fit')
ax.legend(loc='upper left')
fig.savefig('scipy_311_ex2.pdf', bbox_inches='tight')
```

- **مثال ۱۴**

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Let's create a function to model and create data
def func(x, a0, b0, c0, a1, b1, c1):
    return a0 * np.exp(-(x - b0) ** 2 / (2 * c0 ** 2))\
    + a1 * np.exp(-(x - b1) ** 2 / (2 * c1 ** 2))

# Generating clean data
x = np.linspace(0, 20, 200)
y = func(x, 1, 3, 1, -2, 15, 0.5)
# y2 = func(x[np.where(x > 10)], 0, 1, 1, -2, 15,0.5)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Plot out the current state of the data and model
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y, c='k', label='Function')
ax.scatter(x, yn)

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn, p0=[1, 3, 1, 1, 15, 1])

#popt returns the best fit values for parameters of the given model (func)
print(popt)

ym = func(x, popt[0], popt[1], popt[2], popt[3], popt[4], popt[5])
ax.plot(x, ym, c='r', label='Best fit')
ax.legend()
fig.savefig('scipy_311_ex3.pdf', bbox_inches='tight')
```

- **مثال ۱۵**

```
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

# Defining line function
line = lambda x: x + 3
```

```
# Solving for solution
solution = fsolve(line, -2)
print(solution)

# Plotting output
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
x = np.linspace(-5, 5, 1000)
y = line(x)
ax.hlines(0, -5, 0, color='black', alpha=0.15, linestyle='--')
ax.plot(x, y, label='Function')
ax.scatter(solution[0], 0, marker='o',\
           edgecolor='red', facecolor='none', s=100, label='Root')
ax.legend(loc='upper left')
ax.set_xlim(-4.5, -1.5)
ax.set_ylim(-1.5, 1.5)
fig.savefig('scipy_312_ex1.pdf', bbox_inches='tight')
```

- **مثال ۱۶**

```
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

# Defining function to simplify intersection solution
def findIntersection(func1, func2, x0):
    return fsolve(lambda x: func1(x) - func2(x), x0)

# Defining functions that will intersect
funky = lambda x: np.cos(x / 5) * np.sin(x / 2)
line = lambda x: 0.01 * x - 0.5

# Defining range and getting solutions on intersection points
x = np.linspace(0, 45, 10000)
result = findIntersection(funky, line, [15, 20, 30, 35, 40, 45])

# Plotting ouptut
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
ax.plot(x, funky(x), label='Funky func')
ax.plot(x, line(x), label='Line func')
ax.scatter(result, line(result), marker='o',
           edgecolor='red', facecolor='none', s=100)
ax.legend(loc='lower left')
ax.set_xlim(0, 45)
ax.set_ylim(-1, 1)
fig.savefig('scipy_312_ex2.pdf', bbox_inches='tight')
```

- **مثال ۱۷**

```
import numpy as np
from scipy.interpolate import interp1d
```

```
import matplotlib.pyplot as plt

# Setting up fake data.
x = np.linspace(0, 10 * np.pi, 20)
y = np.cos(x)

# Interpolating data
fl = interp1d(x, y, kind='linear')
fq = interp1d(x, y, kind='quadratic')

# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yintl = fl(xint)
yintq = fq(xint)

# Plotting output
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
ax.plot(xint, yintl, label='Linear')
ax.plot(xint, yintq, label='Quadratic')
ax.scatter(x, y, marker='o',
           edgecolor='red', facecolor='none', s=50)
ax.legend(loc='upper left')

ax.set_xlim(0, 10 * np.pi)
ax.set_ylim(-2, 2)
fig.savefig('scipy_32_ex1.pdf', bbox_inches='tight')
```

- **مثال ۱۸**

```
import numpy as np
from scipy.interpolate import UnivariateSpline
import matplotlib.pyplot as plt


# Setting up fake data with artifical noise.
sample = 30
x = np.linspace(1, 10 * np.pi, sample)
y = np.cos(x) + np.log10(x) + np.random.randn(sample) / 10

# Interpolating the data.
f = UnivariateSpline(x, y, s=1)

# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yint = f(xint)
yclean = np.cos(xint) + np.log10(xint)

# Making figure.
fig = plt.figure(figsize=(8, 4))
```

```
ax = fig.add_subplot(111)
ax.plot(xint, yint, label='Interpolation')
ax.plot(xint, yclean, label='Original', c='orange')
ax.scatter(x, y, marker='o',
                    edgecolor='none', facecolor='black', s=20)
ax.legend(loc='upper left')

ax.set_xlim(1, 10 * np.pi)
fig.savefig('scipy_32_ex2.pdf', bbox_inches='tight')
```

- **مثال ۱۹**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

# Defining a function
ripple = lambda x, y: np.sqrt(x ** 2 + y ** 2) + np.sin(x ** 2 + y ** 2)

# Generarting gridded data
grid_x, grid_y = np.mgrid[0:5:1000j, 0:5:1000j]

# Generating saplte that interpolation function will see
xy = np.random.rand(1000, 2)
saplte = ripple(xy[:, 0] * 5, xy[:, 1] * 5)

# Interpolating data
grid_z0 = griddata(xy * 5, saplte, (grid_x, grid_y), method='cubic')

# Making figure.
fig = plt.figure(figsize=(8, 4))

x0, x1 = 0, 1000
y0, y1 = 0, 1000
ax1 = fig.add_subplot(121)
ax1.imshow(ripple(grid_x, grid_y).T, cmap=plt.cm.Blues,
                    interpolation='nearest')
ax1.scatter(xy[:, 0] * 1e3, xy[:, 1] * 1e3, facecolor='black',
               edgecolor='none', s=1)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_xlim(x0, x1)
ax1.set_ylim(y0, y1)

ax2 = fig.add_subplot(122)
ax2.imshow(grid_z0.T, cmap=plt.cm.Blues, interpolation='nearest',
              vmin=0.05, vmax=7.87)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_xlim(x0, x1)
ax2.set_ylim(y0, y1)

fig.savefig('scipy_32_ex3.pdf', bbox_inches='tight')
```

- **مثال ۲۰**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import SmoothBivariateSpline as SBS

# Defining a function
ripple = lambda x, y: np.sqrt(x ** 2 + y ** 2) + np.sin(x ** 2 + y ** 2)

# Generarting gridded data
grid_x, grid_y = np.mgrid[0:5:1000j, 0:5:1000j]

# Generating saplte that interpolation function will see
xy = np.random.rand(1000, 2)
x, y = xy[:, 0], xy[:, 1]
saplte = ripple(xy[:, 0] * 5, xy[:, 1] * 5)

# Interpolating data
fit = SBS(x * 5, y * 5, saplte, s=0, kx=4, ky=4)
interp = fit(np.linspace(0, 5, 1000), np.linspace(0, 5, 1000))

# Making figure.
fig = plt.figure(figsize=(8, 4))

x0, x1 = 0, 1000
y0, y1 = 0, 1000
ax1 = fig.add_subplot(121)
ax1.imshow(ripple(grid_x, grid_y).T, cmap=plt.cm.Blues,
                    interpolation='nearest')
ax1.scatter(xy[:, 0] * 1e3, xy[:, 1] * 1e3, facecolor='black',
               edgecolor='none', s=1)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_xlim(x0, x1)
ax1.set_ylim(y0, y1)

ax2 = fig.add_subplot(122)
ax2.imshow(interp.T, cmap=plt.cm.Blues, interpolation='nearest',
              vmin=0.05, vmax=7.87)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_xlim(x0, x1)
ax2.set_ylim(y0, y1)

fig.savefig('scipy_32_ex4.pdf', bbox_inches='tight')
```

- **مثال ۲۱**

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

```
# Defining function to integrate
func = lambda x: np.cos(np.exp(x)) ** 2

# Integrating function with upper and lower
# limits of 0 and 3, respectively
solution = quad(func, 0, 1000)
print(solution)

# The first element is the desired value
# and the second is the error.

# Plotting output
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
x1 = np.linspace(-1, 4, 10000)
x2 = np.linspace(0, 3, 10000)
ax.plot(x1, func(x1), label='Function')
ax.fill_between(x2, 0, func(x2), alpha=0.2)
ax.set_xlim(-1, 4)
ax.set_ylim(0, 1.05)
fig.savefig('scipy_331_ex1.pdf', bbox_inches='tight')
```

- **مثال ۲۲**

```
import numpy as np
from scipy.integrate import quad, trapz
import matplotlib.pyplot as plt

# Setting up fake data.
x = np.sort(np.random.randn(150) * 4 + 4).clip(0, 5)
# x = np.clip()
func = lambda x: np.sin(x) * np.cos(x ** 2) + 1
y = func(x)

# Integrating function with upper and lower
# limits of 0 and 5, respectively.
fsolution = quad(func, 0, 5)
dsolution = trapz(y, x=x)
print('fsolution = ' + str(fsolution[0]))
print('dsolution = ' + str(dsolution))
print('The difference is ' + str(np.abs(fsolution[0] - dsolution)))

fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(111)
x1 = np.linspace(-1, 6, 10000)
x2 = np.linspace(0, 5, 10000)

ax.plot(x1, func(x1), label='Function')
ax.fill_between(x2, 0, func(x2), alpha=0.2)
ax.scatter(x, y, marker='o', edgecolor='none', facecolor='red',
                    s=7, zorder=3, label='Data points')
ax.legend(loc='upper left')
ax.set_xlim(-1, 6)
```

```
ax.set_ylim(0, 2.5)
fig.savefig('scipy_332_ex1.pdf', bbox_inches='tight')
```

- **مثال ۲۳**

```
import string
import subprocess
import numpy as np
import scipy.stats as s
import matplotlib.pyplot as plt

# Setting up fiture size and spacing
fig = plt.figure(figsize=(8, 9))
fig.subplots_adjust(hspace=0, wspace=0)

# The distributions that will be plotted
dists = ['norm', 'lognorm', 'gamma', 'invgauss', 'cauchy',
         'logistic', 'maxwell', 'powerlaw', 'rdist',
         'wald', 'alpha', 'rayleigh', 'triang', 'lomax',
         'laplace', 'gilbrat', 'fisk', 'erlang', 't', 'nakagami']

dists = np.sort(dists)

for i, D in enumerate(dists):
    print(D)
    x = np.linspace(-5, 5, 1000)
    func = s.__dict__[D]

    try:
        dist = func(loc=0)
        pdf = dist.pdf(x)
    except:
        dist = func(0.7)
        pdf = dist.pdf(x)

    ax = fig.add_subplot(5, 4, i + 1)
    D = str.upper(D[0]) + D[1::]
    ax.plot(x, pdf, label=D)
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.set_xlim(-4, 4)
    ax.set_ylim(0, pdf.max() * 1.5)
    ax.legend(loc='upper right', markerscale=0.0001, prop={'size': 10})

    leg = plt.gca().get_legend()
    ltext = leg.get_texts()
    llines = leg.get_lines()
    frame = leg.get_frame()
    plt.setp(llines, linewidth=0)
    leg.draw_frame(False)

savename = 'scipy_341_ex1.pdf'
fig.savefig(savename, bbox_inches='tight')
```

```
# Calling OS for pdfcrop (Unix and Linux based systems only)
try:
    subprocess.call(['pdfcrop', savename, savename])
except:
    print('This feature will only work if pdfcrop is available')
```

- **مثال ۲۴**

```
import numpy as np
from scipy.stats import norm

# Setup the sample range
x = np.linspace(-5, 5, 1000)

# Here set up the parameters for the normal distribution.
# where loc is the mean and scale is the standard deviation.
dist = norm(loc=0, scale=1)

# Calling norm's PDF and CDF
pdf = dist.pdf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values from
sample = dist.rvs(500)
```

- **مثال ۲۵**

```
import numpy as np
from scipy.stats import geom

# Here set up the parameters for the normal distribution.
# where loc is the mean and scale is the standard deviation.
p = 0.5
dist = geom(p)

# Setup the sample range
x = np.linspace(0, 5, 1000)

# Calling norm's PMF and CDF
pmf = dist.pmf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values from
sample = dist.rvs(500)
```

- **مثال ۲۶**

```
import numpy as np
from scipy import stats
```

```
# Generating a normal distribution sample
# with 100 elements
sample = np.random.randn(100)

# The harmonic mean: Sample values have to
# be greater than 0.
out = stats.hmean(sample[sample > 0])
print('Harmonic mean = ' + str(out))

# The mean, where values below -1 and above 1 are
# removed for the mean calculation
out = stats.tmean(sample, limits=(-1, 1))
print('\nTrimmed mean = ' + str(out))

# Calculating the skewness of the sample
out = stats.skew(sample)
print('\nSkewness = ' + str(out))

# Additionally, there is a handy summary function called
# describe, which gives a quick look at the data.
out = stats.describe(sample)
print('\nSize = ' + str(out[0]))
print('Min = ' + str(out[1][0]))
print('Max = ' + str(out[1][1]))
print('Mean = ' + str(out[2]))
print('Variance = ' + str(out[3]))
print('Skewness = ' + str(out[4]))
print('Kurtosis = ' + str(out[5]))
```

- **مثال ۲۷**

```
import numpy as np
from scipy.cluster import vq
import matplotlib.pyplot as plt

# Creating data
c1 = np.random.randn(100, 2) + 5
c2 = np.random.randn(30, 2) - 5
c3 = np.random.randn(50, 2)

# Pooling all the data into one 150 x 2 array
data = np.vstack([c1, c2, c3])

# Calculating the cluster centriods and variance
# from kmeans
centroids, variance = vq.kmeans(data, 3)

# The identified variable contains the information
# we need to separate the points in clusters
# based on the vq function.
identified, distance = vq.vq(data, centroids)

# Retrieving coordinates for points in each vq
```

```
# identified core
vqc1 = data[identified == 0]
vqc2 = data[identified == 1]
vqc3 = data[identified == 2]

#Setting up plot details
x1, x2 = -10, 10
y1, y2 = -10, 10

fig = plt.figure()
fig.subplots_adjust(hspace=0.1, wspace=0.1)

ax1 = fig.add_subplot(121, aspect='equal')
ax1.scatter(c1[:, 0], c1[:, 1], lw=0.5, color='#00CC00')
ax1.scatter(c2[:, 0], c2[:, 1], lw=0.5, color='#028E9B')
ax1.scatter(c3[:, 0], c3[:, 1], lw=0.5, color='#FF7800')
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_xlim(x1, x2)
ax1.set_ylim(y1, y2)
ax1.text(-9, 8, 'Original')

ax2 = fig.add_subplot(122, aspect='equal')
ax2.scatter(vqc1[:, 0], vqc1[:, 1], lw=0.5, color='#00CC00')
ax2.scatter(vqc2[:, 0], vqc2[:, 1], lw=0.5, color='#028E9B')
ax2.scatter(vqc3[:, 0], vqc3[:, 1], lw=0.5, color='#FF7800')
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_xlim(x1, x2)
ax2.set_ylim(y1, y2)
ax2.text(-9, 8, 'VQ identified')

fig.savefig('scipy_351_ex1.pdf', bbox_inches='tight')
```

- **مثال ۲۸**

```
import numpy as np
from scipy.spatial.distance import pdist, squareform
import scipy.cluster.hierarchy as hy
import matplotlib.pyplot as plt


# Creating a cluster of clusters function
def clusters(number=20, cnumber=5, csize=10):
    # Note that the way the clusters are positioned is Gaussian randomness.
    rnum = np.random.rand(cnumber, 2)
    rn = rnum[:, 0] * number
    rn = rn.astype(int)
    rn[np.where(rn < 5)] = 5
    rn[np.where(rn > number / 2.)] = round(number / 2., 0)
    ra = rnum[:, 1] * 2.9
    ra[np.where(ra < 1.5)] = 1.5

    cls = np.random.randn(number, 3) * csize
```

```
    # Random multipliers for central point of cluster
    rxyz = np.random.randn(cnumber - 1, 3)
    for i in range(cnumber - 1):
        tmp = np.random.randn(rn[i + 1], 3)
        x = tmp[:, 0] + (rxyz[i, 0] * csize)
        y = tmp[:, 1] + (rxyz[i, 1] * csize)
        z = tmp[:, 2] + (rxyz[i, 2] * csize)
        tmp = np.column_stack([x, y, z])
        cls = np.vstack([cls, tmp])
    return cls

# Generate a cluster of clusters and distance matrix.
cls = clusters()
D = pdist(cls[:, 0:2])
D = squareform(D)

# Compute and plot first dendrogram.
fig = plt.figure(figsize=(8, 8))
ax1 = fig.add_axes([0.09, 0.1, 0.2, 0.6])
Y1 = hy.linkage(D, method='complete')
cutoff = 0.3 * np.max(Y1[:, 2])
Z1 = hy.dendrogram(Y1, orientation='right', color_threshold=cutoff)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)

# Compute and plot second dendrogram.
ax2 = fig.add_axes([0.3, 0.71, 0.6, 0.2])
Y2 = hy.linkage(D, method='average')
cutoff = 0.3 * np.max(Y2[:, 2])
Z2 = hy.dendrogram(Y2, color_threshold=cutoff)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)

# Plot distance matrix.
ax3 = fig.add_axes([0.3, 0.1, 0.6, 0.6])
idx1 = Z1['leaves']
idx2 = Z2['leaves']
D = D[idx1, :]
D = D[:, idx2]
ax3.matshow(D, aspect='auto', origin='lower', cmap=plt.cm.YlGnBu)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)

# Plot colorbar.
fig.savefig('scipy_352_ex1.pdf', bbox='tight')
```

- **مثال ۲۹**

```
import numpy as np
import scipy.cluster.hierarchy as hy
import matplotlib.pyplot as plt

# Generate random features and distance matrix.
```

```
def clusters(number=20, cnumber=10, csize=10):
    # Note, the way the clusters are positioned is gaussian randomness
    rnum = np.random.rand(cnumber, 2)
    rn = rnum[:, 0] * number
    rn = rn.astype(int)
    rn[np.where(rn < 5)] = 5
    rn[np.where(rn > number / 2.)] = round(number / 2., 0)

    ra = rnum[:, 1] * 2.9
    ra[np.where(ra < 1.5)] = 1.5

    cls = np.random.randn(number, 3) * csize

    # Random multipliers for central point of cluster
    rxyz = np.random.randn(cnumber - 1, 3)
    for i in range(cnumber - 1):
        tmp = np.random.randn(rn[i + 1], 3)
        x = tmp[:, 0] + (rxyz[i, 0] * csize)
        y = tmp[:, 1] + (rxyz[i, 1] * csize)
        z = tmp[:, 2] + (rxyz[i, 2] * csize)
        tmp = np.column_stack([x, y, z])
        cls = np.vstack([cls, tmp])
    return cls

# Here we define a function to collect the coordinates of
# each point of the different clusters.
def group(data, index):
    number = np.unique(index)
    groups = []
    for i in number:
        groups.append(data[index == i])

    return groups


# Creating a cluster of clusters
cls = clusters()

# Calculating the linkage matrix
Y = hy.linkage(cls[:, 0:2], method='complete')

# Here we use the fcluster function to pull out a
# collection of flat clusters from the hierarchical
# data structure. Note that we are using the same
# cutoff value as in the previous example for the dendrogram
# using the 'complete' method.
cutoff = 0.3 * np.max(Y[:, 2])
index = hy.fcluster(Y, cutoff, 'distance')

# Using the group function, we group points into their
# respective clusters.
groups = group(cls, index)

# Plotting clusters
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111)
colors = ['r', 'c', 'b', 'g', 'orange', 'k', 'y', 'gray']
```

```
for i, g in enumerate(groups):
    i = np.mod(i, len(colors))
    ax.scatter(g[:, 0], g[:, 1], c=colors[i], edgecolor='none', s=50)
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

fig.savefig('scipy_352_ex2.pdf', bbox='tight')
```

- **مثال ۳۰**

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob

# Getting the list of files in the directory
files = glob('space/*.JPG')

# Opening up the first image for loop
im1 = imread(files[0]).astype(np.float32)

# Starting loop and continue co-adding new images
for i in range(1, len(files)):
    print(i)
    im1 += imread(files[i]).astype(np.float32)

# Saving img
imsave('scipy_36_ex1.jpg', im1)
```

- **مثال ۳۱**

```
import numpy as np
from scipy.misc import imread, imsave
from glob import glob

# This function allows us to place in the
# brightest pixels per x and y position between
# two images. It is similar to PIL's
# ImageChop.Lighter function.
def chop_lighter(image1, image2):
    s1 = np.sum(image1, axis=2)
    s2 = np.sum(image2, axis=2)

    index = s1 < s2
    image1[index, 0] = image2[index, 0]
    image1[index, 1] = image2[index, 1]
    image1[index, 2] = image2[index, 2]
    return image1

# Getting the list of files in the directory
files = glob('space/*.JPG')
```

```
# Opening up the first image for looping
im1 = imread(files[0]).astype(np.float32)
im2 = np.copy(im1)

# Starting loop
for i in range(1, len(files)):
    print(i)
    im = imread(files[i]).astype(np.float32)
    # Same before
    im1 += im
    # im2 image shows star trails better
    im2 = chop_lighter(im2, im)

# Saving image with slight tweaking on the combination
# of the two images to show star trails with the
# co-added image.
imsave('scipy_36_ex2.jpg', im1 / im1.max() + im2 / im2.max() * 0.2)
```

- **مثال ۳۲**

```
import numpy as np
from scipy.sparse.linalg import eigsh
from scipy.linalg import eigh
import scipy.sparse
import time

N = 3000
# Creating a random sparse matrix
m = scipy.sparse.rand(N, N)

# Creating an array clone of it
a = m.toarray()

print('The numpy array data size: ' + str(a.nbytes) + ' bytes')
print('The sparse matrix data size: ' + str(m.data.nbytes) + ' bytes')

# Non-sparse
t0 = time.time()

res1 = eigh(a)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Non-sparse operation takes ' + dt)

# Sparse
t0 = time.time()
res2 = eigsh(m)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Sparse operation takes ' + dt)
```

- **مثال ۳۳**

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.ndimage as ndimage
import skimage.filter as skif

# Generating data points with a non-uniform background
x = np.random.uniform(low=0, high=100, size=20).astype(int)
y = np.random.uniform(low=0, high=100, size=20).astype(int)

# Creating image with non-uniform background
func = lambda x, y: x ** 2 + y ** 2
grid_x, grid_y = np.mgrid[-1:1:100j, -2:2:100j]
bkg = func(grid_x, grid_y)
bkg = bkg / np.max(bkg)

# Creating points
clean = np.zeros((100, 100))
clean[(x, y)] += 5
clean = ndimage.gaussian_filter(clean, 3)
clean = clean / np.max(clean)

# Combining both the non-uniform background
# and points
fimg = bkg + clean
fimg = fimg / np.max(fimg)

# Defining minimum neighboring size of objects
block_size = 3

# Adaptive threshold function which returns image
# map of structures that are different relative to
# background
adaptive_cut = skif.threshold_adaptive(fimg, block_size, offset=0)

# Global threshold
global_thresh = skif.threshold_otsu(fimg)
global_cut = fimg > global_thresh

# Creating figure to highlight difference between
# adaptive and global threshold methods
fig = plt.figure(figsize=(8, 4))
fig.subplots_adjust(hspace=0.05, wspace=0.05)

ax1 = fig.add_subplot(131)
ax1.imshow(fimg)
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)

ax2 = fig.add_subplot(132)
ax2.imshow(global_cut)
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)

ax3 = fig.add_subplot(133)
ax3.imshow(adaptive_cut)
ax3.xaxis.set_visible(False)
ax3.yaxis.set_visible(False)
```

```
fig.savefig('scikits_411_ex1.pdf', bbox_inches='tight')
```

- **مثال ۳۴**

```
import numpy as np
import scipy.ndimage as ndimage
import skimage.morphology as morph
import matplotlib.pyplot as plt

# Generating data points with a non-uniform background
x = np.random.uniform(low=0, high=200, size=20).astype(int)
y = np.random.uniform(low=0, high=400, size=20).astype(int)

# Creating image with non-uniform background
func = lambda x, y: np.cos(x) + np.sin(y)
grid_x, grid_y = np.mgrid[0:12:200j, 0:24:400j]
bkg = func(grid_x, grid_y)
bkg = bkg / np.max(bkg)

# Creating points
clean = np.zeros((200, 400))
clean[(x, y)] += 5
clean = ndimage.gaussian_filter(clean, 3)
clean = clean / np.max(clean)

# Combining both the non-uniform background
# and points
fimg = bkg + clean
fimg = fimg / np.max(fimg)

# Calculating local maxima
lm1 = morph.is_local_maximum(fimg)
x1, y1 = np.where(lm1.T == True)

# Creating figure to show local maximum detection
# rate success
fig = plt.figure(figsize=(8, 4))

ax = fig.add_subplot(111)
ax.imshow(fimg)
ax.scatter(x1, y1, s=100, facecolor='none', edgecolor='#009999')
ax.set_xlim(0, 400)
ax.set_ylim(0, 200)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

fig.savefig('scikits_412_ex1.pdf', bbox_inches='tight')
```

- **مثال ۳۵**

```
import numpy as np
import pyfits
import skimage.morphology as morph
import skimage.exposure as skie
import matplotlib.pyplot as plt

# Loading astronomy image from an infrared space telescope
img = pyfits.getdata('stellar_cluster.fits')[500:1500, 500:1500]

# Prep file scikit-image environment and plotting
limg = np.arcsinh(img)
limg = limg / limg.max()
low = np.percentile(limg, 0.25)
high = np.percentile(limg, 99.5)
opt_img = skie.exposure.rescale_intensity(limg, in_range=(low, high))

# Calculating local maxima and filtering out noise
lm = morph.is_local_maximum(limg)
x1, y1 = np.where(lm.T == True)
v = limg[(y1, x1)]
lim = 0.5
x2, y2 = x1[v > lim], y1[v > lim]

# Creating figure to show local maximum detection
# rate success
fig = plt.figure(figsize=(8, 4))
fig.subplots_adjust(hspace=0.05, wspace=0.05)

ax1 = fig.add_subplot(121)
ax1.imshow(opt_img)
ax1.set_xlim(0, img.shape[1])
ax1.set_ylim(0, img.shape[0])
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)

ax2 = fig.add_subplot(122)
ax2.imshow(opt_img)
ax2.scatter(x2, y2, s=80, facecolor='none', edgecolor='#FF7400')
ax2.set_xlim(0, img.shape[1])
ax2.set_ylim(0, img.shape[0])
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)

fig.savefig('scikits_412_ex2.pdf', bbox_inches='tight')
```

- **مثال ۳۶**

```
import numpy as np
from sklearn import linear_model
from sklearn.datasets.samples_generator import make_regression
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generating synthetic data for training and testing
```

```
X, y = make_regression(n_samples=100, n_features=2, n_informative=1,
                       random_state=0, noise=50)

# X and y are values for 3D space. We first need to train
# the machine, so we split X and y into X_train, X_test,
# y_train, and y_test. The *_train data will be given to the
# model to train it.
X_train, X_test = X[:80], X[-20:]
y_train, y_test = y[:80], y[-20:]

# Creating instance of model
regr = linear_model.LinearRegression()

# Training the model
regr.fit(X_train, y_train)

# Printing the coefficients
print(regr.coef_)
# [-10.25691752  90.5463984 ]

# Predicting y-value based on training
X1 = np.array([1.2, 4])
print(regr.predict(X1))
# 350.860363861

# With the *_test data we can see how the result matches
# the data the model was trained with.
# It should be a good match as the *_train and *_test
# data come from the same sample. Output: 1 is perfect
# prediction and anything lower is worse.
print(regr.score(X_test, y_test))

fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(111, projection='3d')

# Data
ax.scatter(X_train[:, 0], X_train[:, 1], y_train, facecolor='#00CC00')
ax.scatter(X_test[:, 0], X_test[:, 1], y_test, facecolor='#FF7800')

# Function with coefficient variables
coef = regr.coef_
line = lambda x1, x2: coef[0] * x1 + coef[1] * x2

grid_x1, grid_x2 = np.mgrid[-2:2:10j, -2:2:10j]
ax.plot_surface(grid_x1, grid_x2, line(grid_x1, grid_x2),
                alpha=0.1, color='k')
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
ax.zaxis.set_visible(False)
fig.savefig('scikits_421_ex1.pdf', bbox='tight')
```

- **مثال ۳۷**

```
import numpy as np
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt


# Creating data
c1 = np.random.randn(100, 2) + 5
c2 = np.random.randn(50, 2)

# Creating a uniformly distributed background
u1 = np.random.uniform(low=-10, high=10, size=100)
u2 = np.random.uniform(low=-10, high=10, size=100)
c3 = np.column_stack([u1, u2])

# Pooling all the data into one 150 x 2 array
data = np.vstack([c1, c2, c3])

# Calculating the cluster with DBSCAN function.
# db.labels_ is an array with identifiers to the
# different clusters in the data.
db = DBSCAN(eps=0.95, min_samples=10).fit(data)
labels = db.labels_

# Retrieving coordinates for points in each
# identified core. There are two clusters
# denoted as 0 and 1 and the noise is denoted
# as -1. Here we split the data based on which
# component they belong to.
dbc1 = data[labels == 0]
dbc2 = data[labels == 1]
noise = data[labels == -1]

# Setting up plot details
x1, x2 = -12, 12
y1, y2 = -12, 12

fig = plt.figure()
fig.subplots_adjust(hspace=0.1, wspace=0.1)

ax1 = fig.add_subplot(121, aspect='equal')
ax1.scatter(c1[:, 0], c1[:, 1], lw=0.5, color='#00CC00')
ax1.scatter(c2[:, 0], c2[:, 1], lw=0.5, color='#028E9B')
ax1.scatter(c3[:, 0], c3[:, 1], lw=0.5, color='#FF7800')
ax1.xaxis.set_visible(False)
ax1.yaxis.set_visible(False)
ax1.set_xlim(x1, x2)
ax1.set_ylim(y1, y2)
ax1.text(-11, 10, 'Original')

ax2 = fig.add_subplot(122, aspect='equal')
ax2.scatter(dbc1[:, 0], dbc1[:, 1], lw=0.5, color='#00CC00')
ax2.scatter(dbc2[:, 0], dbc2[:, 1], lw=0.5, color='#028E9B')
ax2.scatter(noise[:, 0], noise[:, 1], lw=0.5, color='#FF7800')
ax2.xaxis.set_visible(False)
ax2.yaxis.set_visible(False)
ax2.set_xlim(x1, x2)
ax2.set_ylim(y1, y2)
```

```
ax2.text(-11, 10, 'DBSCAN identified')

fig.savefig('scikits_422_ex1.pdf', bbox_inches='tight')
```

واژه‌نامه

اعشاری	*Float*	تورفتگی	*Indentation*
انبارش	*Storage*	چندتایی	*Tuple*
انباشت‌گاه	*Repository*	خوشه‌بندی	*Clustering*
انباشته	*Stacked*	دستور جادویی	*Magic Command*
اندیس	*Index*	رشته	*String*
انطباق‌پذیر	*Adaptive*	زیرساخت	*Platfrom*
آستانه‌گذاری	*Thresholding*	سلسله‌مراتبی	*Hierarchical*
برازش	*Fit*	شیء	*Object*
برش	*Slicing*	صحیح	*Integer*
بسته	*Package*	فرسایش	*Erosion*
پایانه	*Terminal*	کلاس	*Class*
پوشانه	*Mask*	کوانتش	*Quantization*
پویا	*Dynamic*	ماژول	*Module*
تُنُک	*Sparse*	نحو	*Syntax*
تابع‌اولیه‌گیری	*Integration*	نما	*View*
تجزیه کردن	*Parse*	همگردان	*Compiler*
تعاملی	*Interactive*	هموارسازی	*Smoothing*