Introduction to Bayesian Statistics - 11

Edoardo Milotti

Università di Trieste and INFN-Sezione di Trieste

Readable explanation of MCMCs (link)

The Astrophysical Journal Supplement Series, 236:11 (18pp), 2018 May https://doi.org/10.3847/1538-4365/aab76e

Data Analysis Recipes: Using Markov Chain Monte Carlo*

David W. Hogg[1,2,3,4] and Daniel Foreman-Mackey[1,5,6]
[1] Center for Computational Astrophysics, Flatiron Institute, 162 Fifth Ave., New York, NY 10010, USA
[2] Center for Cosmology and Particle Physics, Department of Physics, New York University, 726 Broadway, New York, NY 10003, USA
[3] Center for Data Science, New York University, 60 Fifth Ave., New York, NY 10011, USA
[4] Max-Planck-Institut für Astronomie, Königstuhl 17, D-69117 Heidelberg, Germany
[5] Department of Astronomy, University of Washington, Box 351580, Seattle, WA 98195, USA
Received 2017 October 16; revised 2018 January 23; accepted 2018 February 3; published 2018 May 11

Abstract

Markov Chain Monte Carlo (MCMC) methods for sampling probability density functions (combined with abundant computational resources) have transformed the sciences, especially in performing probabilistic inferences, or fitting models to data. In this primarily pedagogical contribution, we give a brief overview of the most basic MCMC method and some practical advice for the use of MCMC in real inference problems. We give advice on method choice, tuning for performance, methods for initialization, tests of convergence, troubleshooting, and use of the chain output to produce or report parameter estimates with associated uncertainties. We argue that autocorrelation time is the most important test for convergence, as it directly connects to the uncertainty on the sampling estimate of any quantity of interest. We emphasize that sampling is a method for doing integrals; this guides our thinking about how MCMC output is best used.

1. When Do You Need MCMC?

Markov Chain Monte Carlo (MCMC) methods are methods for sampling probability distribution functions or probability density functions (pdfs). These pdfs may be either probability mass functions on a discrete space or probability densities on a continuous space, though we will concentrate on the latter in this article. MCMC methods do not require that you have a full analytic description of the properly normalized pdf for sampling to proceed; they only require that you be able to compute ratios of the pdf at pairs of locations. This makes MCMC methods ideal for sampling *posterior pdfs* in probabilistic inferences.

In a probabilistic inference, the posterior pdf $p(\theta|D)$, or pdf for the parameters θ given the data D, is constructed from the likelihood $p(D|\theta)$, or pdf for the data given the parameters, and the prior pdf $p(\theta)$ for the parameters by what is often known as the "Bayes rule,"

$$p(\theta|D) = \frac{1}{Z}\, p(D|\theta)p(\theta). \tag{1}$$

In these contexts, the constant Z, sometimes written as $p(D)$, is known by the names "evidence," "marginal likelihood," "Bayes integral," and "prior predictive probability" and is usually *extremely hard to calculate*.[7] That is, you often know the function

[7] The factor Z is often difficult to compute, because the likelihood (or the prior) can have extremely complex structure, with multiple arbitrarily compact modes, arbitrarily positioned in the (presumably high-dimensional) parameter space θ. Elsewhere, we discuss the computation of this object (Hou et al. 2014), and so have many others before us. We also have (unpublished) philosophical arguments against calculating this Z if you can possibly avoid it, but these are outside the scope of this article. The point is that MCMC methods will not require that you know Z.

$p(\theta|D)$ up to a constant factor; you can compute ratios of the pdf at pairs of points, but not the precise value at any individual point.

In addition to this normalization-insensitive property of MCMC, in its simplest forms it can be run without computing any derivatives or integrals of the function, and (as we will show below in Section 3) in its simplest forms it is *extremely easy to implement*. For all these reasons, MCMC is ideal for sampling posterior pdfs in the real situations in which scientists find themselves.

Say you are in this situation: You have a huge blob of data D (think of this as a vector or list or heterogeneous but ordered collection of observations). You also have a model sophisticated enough—a probabilistic, generative model, if you will[8]—that, given a setting of a huge blob of parameters (again, think of this as a vector or list or heterogeneous but ordered collection of values) θ, you can compute a pdf for data (or likelihood[9]) $p(D|\theta)$. Furthermore, say also that you can write down some kind of informative or vague prior pdf $p(\theta)$ for the parameter blob θ. If all these things are true, then—even if you cannot compute anything else—in principle a trivial-to-implement MCMC can give you a fair sampling of the posterior pdf. That is, you can run MCMC (for a very long time—see Section 5 for how long), and you will be left with a set of K parameter-blob settings θ_k such that the full set $\{\theta_k\}_{k=1}^{K}$ constitutes a fair sampling from the posterior pdf $p(\theta|D)$. We will give some sense of what "fair" means in this context below.

Miscellany of topics related to the emcee examples

1. Integrated Gaussian probability in n-dimensional space

We take a $\sim N(0,1)$ Gaussian pdf in n-dimensional space and integrate it in a spherical region of radius R surrounding the origin:

$$
\begin{aligned}
P(R) &= \frac{1}{(2\pi)^{n/2}} \int_0^R e^{-r^2/2} dV_n \\
&= \frac{1}{(2\pi)^{n/2}} \int_0^R \frac{n\pi^{n/2}}{\Gamma(n/2+1)} r^{n-1} e^{-r^2/2} dr \\
&= \frac{n}{2\Gamma(n/2+1)} \int_0^{R^2/2} x^{n/2-1} e^{-x} dx \\
&= \frac{n}{2\Gamma(n/2+1)} \gamma\left(\frac{n}{2}, \frac{R^2}{2}\right)
\end{aligned}
$$

where $\gamma(s,x) = \int_0^x t^{s-1} e^{-t} dt$

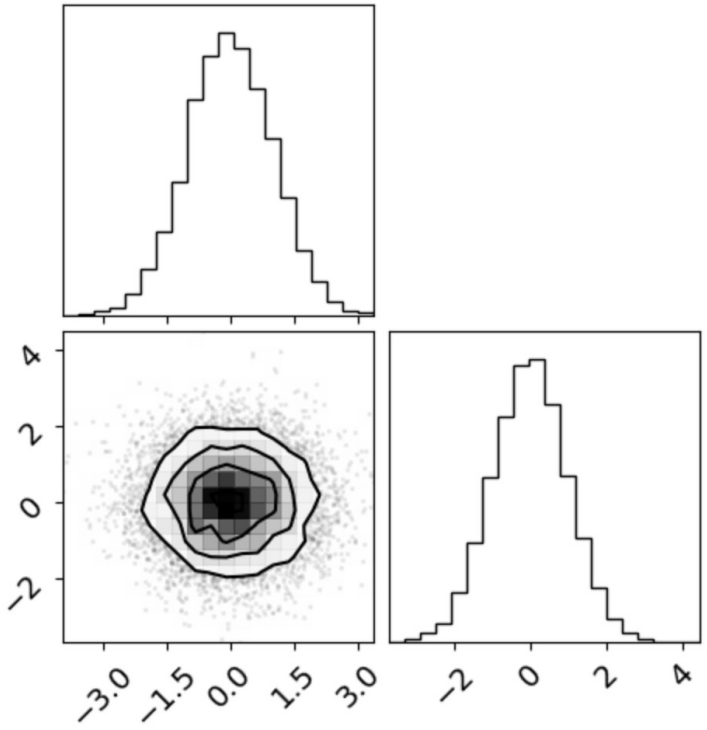
is the lower incomplete gamma function

dim	R in units of st. devs.					
	1	2	3	4	5	6
1	0.682689	0.9545	0.9973	0.999937	0.999999	1.
2	0.393469	0.864665	0.988891	0.999665	0.999996	1.
3	0.198748	0.738536	0.970709	0.998866	0.999985	1.
4	0.090204	0.593994	0.938901	0.996981	0.99995	1.
5	0.0374342	0.450584	0.890936	0.993156	0.999861	0.999999
6	0.0143877	0.323324	0.826422	0.986246	0.999659	0.999997
7	0.00517146	0.220223	0.747344	0.974884	0.999241	0.999993
8	0.00175162	0.142877	0.657704	0.95762	0.998445	0.999982
9	0.000562497	0.0885875	0.562726	0.933118	0.997029	0.99996
10	0.000172116	0.052653	0.467896	0.900368	0.994654	0.999916
11	0.0000503899	0.030083	0.378108	0.858869	0.990883	0.999831
12	0.0000141649	0.0165636	0.29707	0.808764	0.985177	0.999676
13	3.83473e-6	0.00880861	0.227056	0.75087	0.976916	0.999407
14	1.00238e-6	0.00453381	0.168949	0.686626	0.965433	0.998957
15	2.53564e-7	0.00226266	0.122483	0.617948	0.950057	0.998232
16	6.21969e-8	0.00109672	0.0865865	0.547039	0.930175	0.997107
17	1.48197e-8	0.000517067	0.0597382	0.476165	0.90529	0.995413
18	3.43549e-9	0.000237447	0.0402573	0.407453	0.875084	0.992944
19	7.75939e-10	0.00010634	0.0265206	0.342722	0.839458	0.989444
20	1.70967e-10	0.0000464981	0.0170927	0.283376	0.798569	0.984619

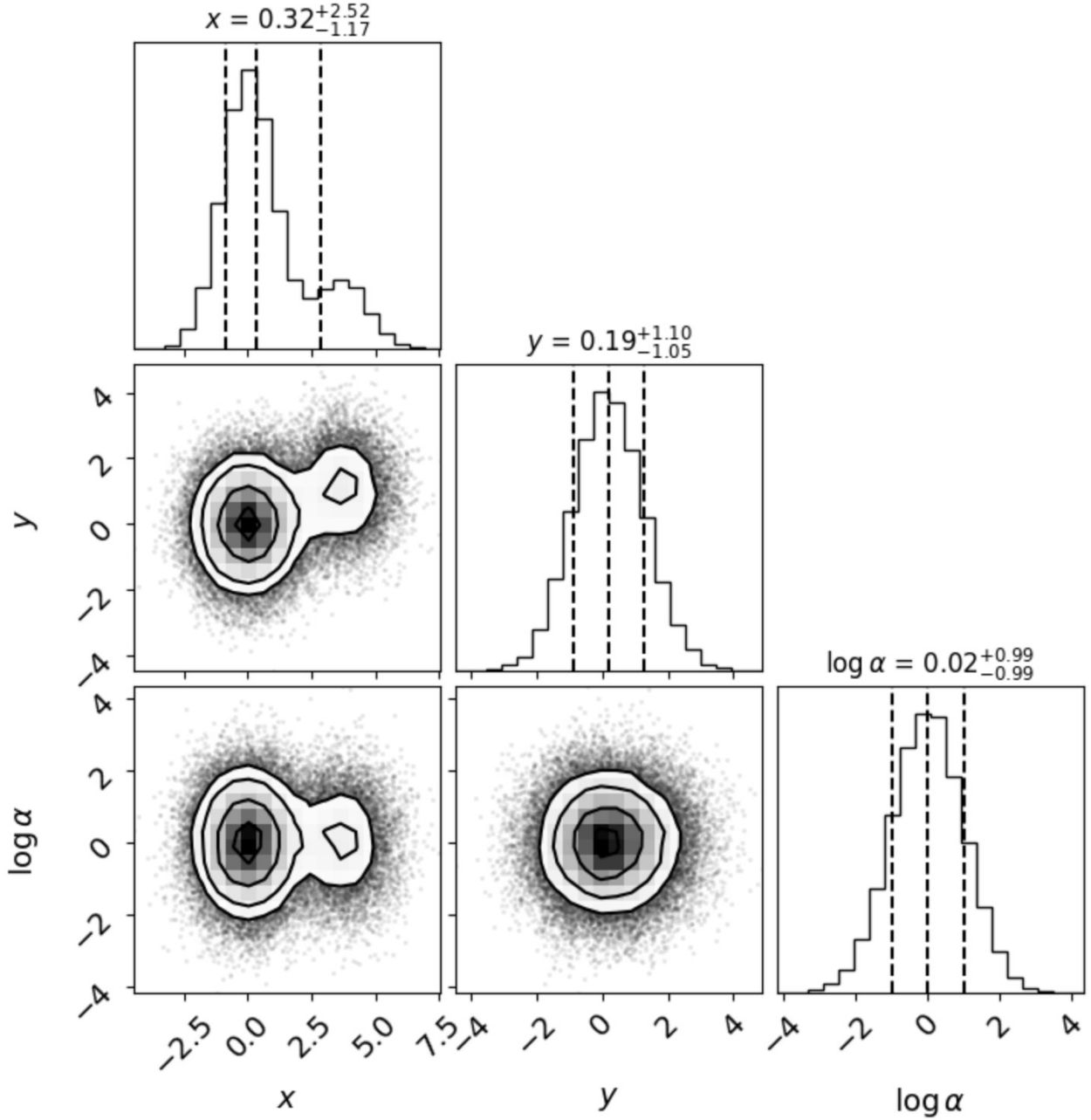
2. **corner plots**

Useful python package `corner.py` (https://corner.readthedocs.io/en/latest/)

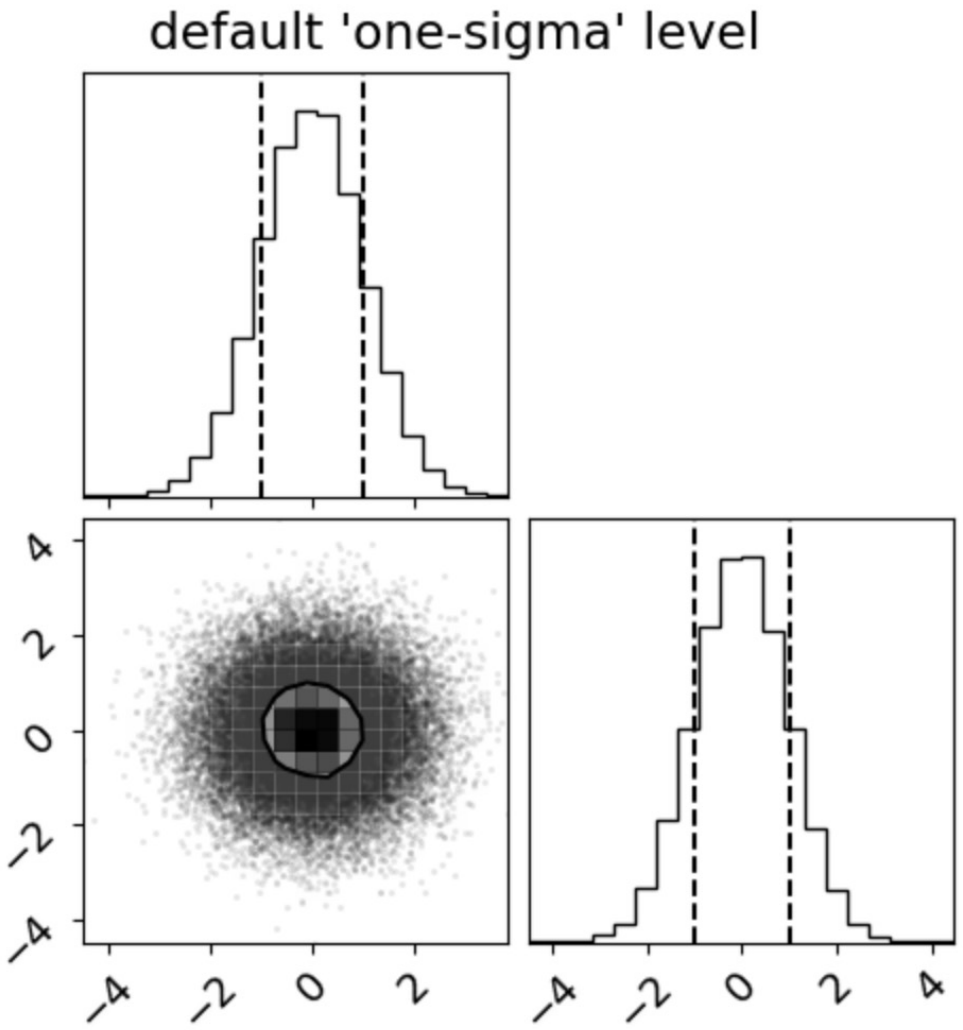
```
import corner
import numpy as np

ndim, nsamples = 2, 10000
np.random.seed(42)
samples = np.random.randn(ndim * nsamples).reshape([nsamples, ndim])
figure = corner.corner(samples)
```



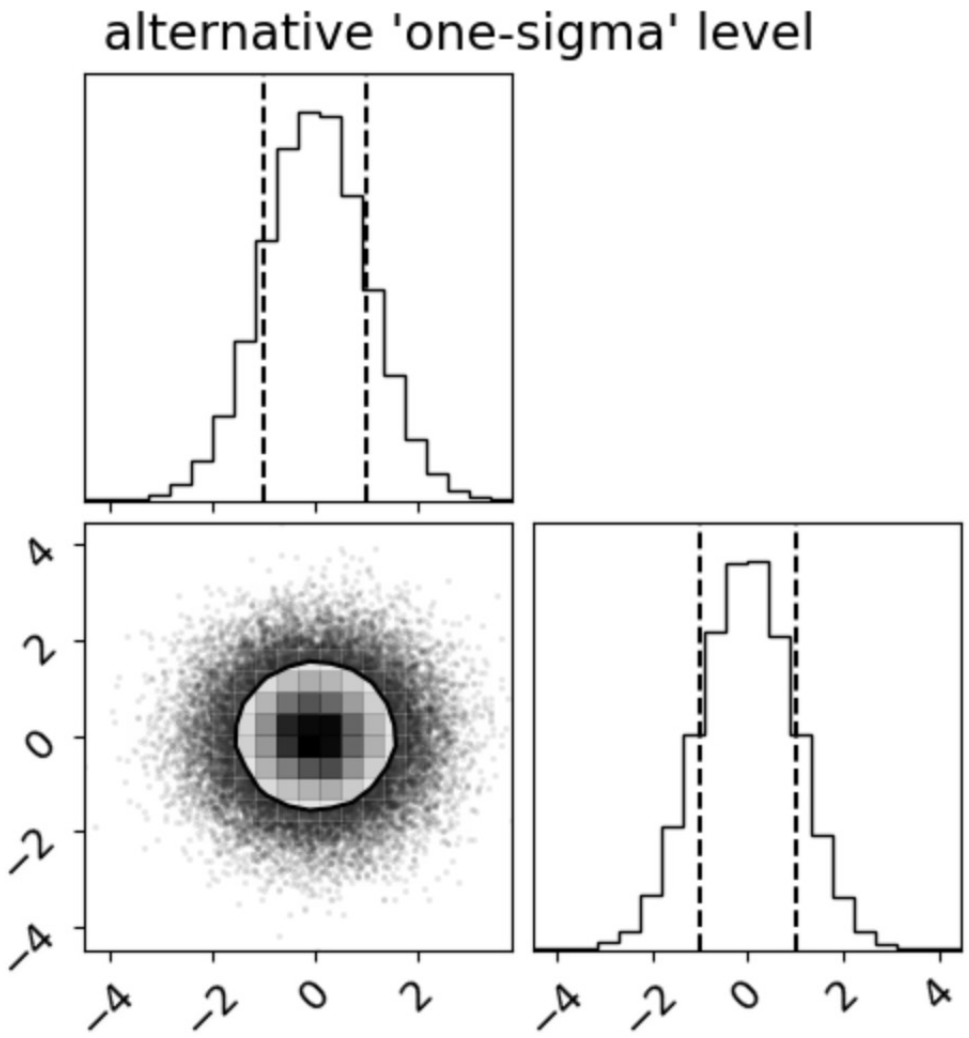
```python
# Set up the parameters of the problem.
ndim, nsamples = 3, 50000

# Generate some fake data.
np.random.seed(42)
data1 = np.random.randn(ndim * 4 * nsamples // 5).reshape(
    [4 * nsamples // 5, ndim]
)
data2 = 4 * np.random.rand(ndim)[None, :] + np.random.randn(
    ndim * nsamples // 5
).reshape([nsamples // 5, ndim])
data = np.vstack([data1, data2])

# Plot it.
figure = corner.corner(
    data,
    labels=[
        r"$x$",
        r"$y$",
        r"$\log \alpha$"
    ],
    quantiles=[0.16, 0.5, 0.84],
    show_titles=True,
    title_kwargs={"fontsize": 12},
)
```



```
fig = corner.corner(x, quantiles=(0.16, 0.84), levels=(1 - np.exp(-0.5),))
_ = fig.suptitle("default 'one-sigma' level")
```



```
fig = corner.corner(x, quantiles=(0.16, 0.84), levels=(0.68,))
_ = fig.suptitle("alternative 'one-sigma' level")
```



3. **multiprocessing**

Useful python package `multiprocessing` (https://docs.python.org/3/library/multiprocessing.html)

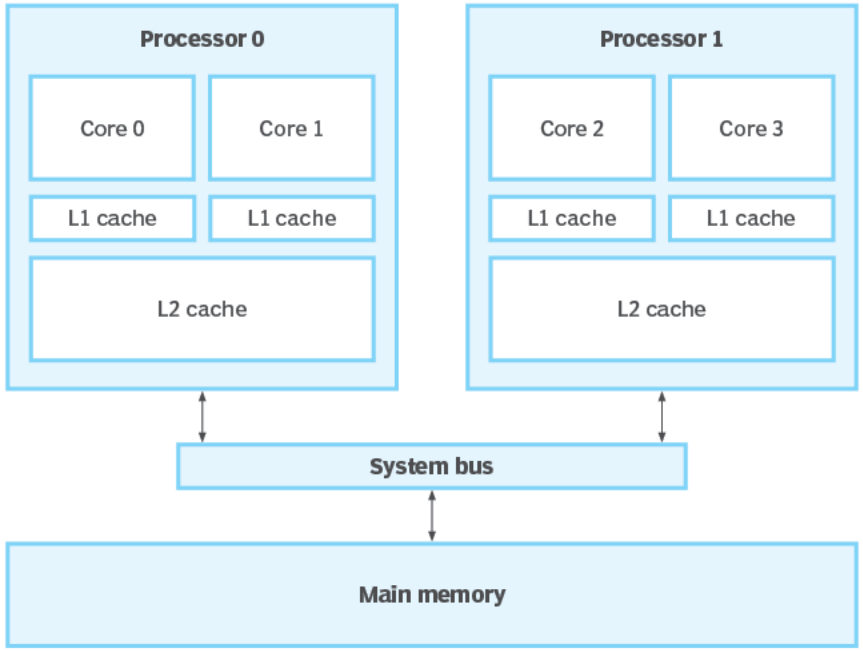
Here we use the package ONLY in the context of **`emcee`**.

```python
import multiprocessing
import time

def cube(x):
    return x**3

if __name__ == "__main__":
    pool = multiprocessing.Pool()
    start_time = time.perf_counter()
    processes = [pool.apply_async(cube, args=(x,)) for x in range(1,1000)]
    result = [p.get() for p in processes]
    finish_time = time.perf_counter()
    print(f"Program finished in {finish_time-start_time} seconds")
    print(result)
```

Multiprocessing means using two or more central processing units (CPUs) in a single computer system.

Its definition can vary, but generally it refers to the ability to support multiple CPUs and to the capacity to distribute work among them.

Present–day multicore processors can easily have 12, 24 or more cores on the same motherboard, enabling concurrent processing of numerous tasks.



What are processes?

A process, or a running process, is a collection of instructions carried out by the computer processor.
A process's execution proceeds in a sequential manner, just as a computer program which is written in a text file, but when executed, it becomes a process and performs sequentially all the tasks crafted in the program.

A normal computer runs several processes continuously to manage operating system, and all hardware and applications installed on the machine. This can range from simple background tasks like a spell-checker or system event handlers to a complex application like Microsoft Word.

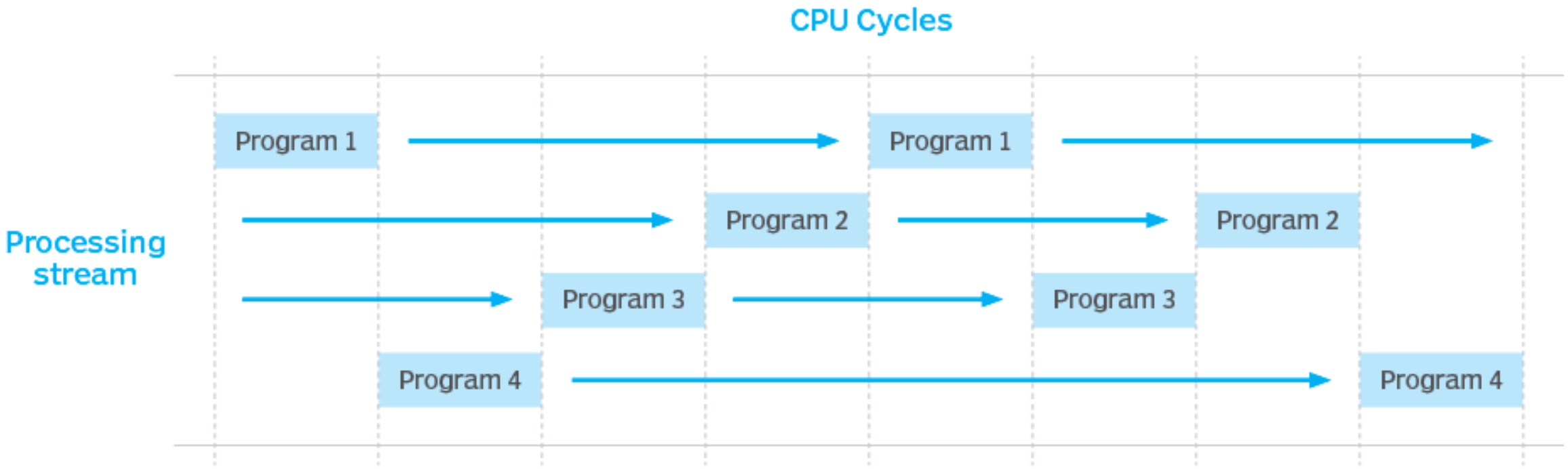
Processes vs. threads

According to a common misconception processes and threads are the same while, in fact, they are different execution sequences. Here is a list of differences between a processes and a threads:

- A process is part of a running program, while a thread belongs to a given process.
- Threads are small when compared to processes.
- A process takes much longer to terminate.
- The creation of a process takes longer than the creation of a thread.
- Processes require more time for context switching.
- Unlike threads, which share memory, processes have their own memory space.
- Data is not shared across processes, but it is between threads.

Multiprocessing can also be confused with multitasking or time sharing, the management of programs and the system services they request as tasks that can be interleaved, and with multithreading, the management of multiple execution paths through the computer or of multiple users sharing the same copy of a program.

Illustration of multithreading with 4 threads



PRO's

Multiprocessing environments are widely adopted and offer a many advantages such as increased speed, throughput and reliability. Common benefits of multiprocessing include the following:

- **Reliability**. If one processor fails in a multiprocessor system, the other processors can pick up the slack and continue to function. While the shutting down of one processor might cause a gradual slowdown, the system can still function smoothly. This makes multiprocessing systems highly reliable.
- **Increased throughput**. Throughput is the number of processes executed at a given time. Given that multiprocessor systems use many CPUs to handle data, increased performance is expected when the system uses parallel processing. This means more tasks can be accomplished in a shorter amount of time, as they're divided among different processors.
- **Cost saving**. Multiprocessing systems are more economical compared to multiple single processor systems. This is because multiple processors within a single system share the same memory, disk space, buses and peripherals.

CON's

While multiprocessing improves system performance and reliability, it does come with the following challenges:

- **Expensive**. Systems with multiple processors may be expensive. Having just one processor is less expensive than having two or more.
- **Deadlocks**. In systems with multiple processors, a deadlock can occur if one processor attempts to access an I/O device while another processor is trying to use it.
- **Extra memory requirements**. Due to their improved computing capability, multiprocessor computers are widely used. However, they require more memory. In multiprocessing architectures, memory is shared across all processes and each processor needs memory space. All processors work together and have direct access to the main memory, which causes an increase in memory usage.
- **Complex operating system**. In multiprocessing OSs, each CPU has its own operating system, which assigns each processor with several minor tasks and the load is distributed among the processors. However, the use of multiple processors makes it more complex for the OSs to function.