π의 정확한 의미와 그에 대한 실용적 고찰

임대영
고려대학교 통계학과

April 20, 2016

Contents

1 π는 존재하지 않는다

통계를 조금 공부해보면 다음과 같은 질문이 자주 생기게 된다. '실제로 정규 분포를 따르는 것이 있을까?' 통계 질문방에 가봐도 "아이비리그에 들어가는 것은 확률과정인가요?" 혹은 "학생들의 성적은 정규분포인가요?"와 같은 질문이 수도 없이 많다. 이런 질문이 올라올 때마다 나의 답변은 "이 세상에 존재하는 그 어떤 것도 확률과정이거나 정규분포를 따르는 것은 없습니다"이다. 수학을 공부하는 사람들이 가장 많이 하는 착각이 바로 이런 것이다. 수학 책에 나오는 모든 것들이 실재할 것이라는 착각. 물론 직관적으로 이해가 잘 되는 것은 있다. 자연수가 대표적이다. 이 때문에 레오폴트 크로네커(Leopold Kronecker)라는 과거 프로이센의 수학자는 다음과 같이 말하기도 했다.

"정수는 신이 만들었지만 나머지는 인간의 창작물이다." (God made the integers. All else is the work of man.)

'수'라는 것은 인간의 위대한 발명품이지만 실제로 존재하지는 않는다. 상당히 철학적이다. 숫자가 존재하지 않는다니! 하지만 잘 생각해보면 숫자라는 건 실재하는 것이 아니라 사람이 쓰기 좋게 만들어놓은 하나의 도구라는 것이 명료해질 것이다. 자꾸 이런 직관에 의존하려고 하다보니 어차피 다 실존하지 않는 창작물인데 유독 '허수'에만 허구라는 꼬리표를 달아놓는다. 하지만 허수를 포함한 복소수도 매우 직관적으로 이해할 수 있는 것들이며 그와 동시에 존재하지 않으면서도 많은 현상과 운동, 법칙을 설명할 수 있는 좋은 도구이다. 이러한 특성은 π라고 예외일 수 없다. 매우 충격적인 사실은 물리적인 형체를 지니고 있는 이 세상의 그 어떤 원도 원주와 지름의 비율이 정확히 π가 아니다. 모든 물리적인 측정에는 오차가 따른다. 다시 한 번 강조하지만 정확히 π의 값을 지니는 것은 없다. 있다고 해도 우리는 무엇이 '정확히' π의 값을 지니는지 알 턱이 없다. π는 없다.

2 π 원주율이 아니다

π를 처음 배울 때 그 누구나 다 '원주율'이라고 배웠다. 반은 맞고 반은 틀렸다. 유클리드 기하에서는 원의 곡률이 0이고 원주와 지름의 비율이 π라는 상수가 된다. 하지만 쌍곡 기하에서는 곡률이 음수가 되며 원이 커지면 커질수록 원주와 지름의 비율은 무한대로 증가한다. 따라서 어떤 상수로 정의될 수 없다. 타원 기하에서도 마찬가지로 곡률이 양수이며 원주와 지름의 비율이 원이 커짐에 따라 감소한다. 여기서도 상수가 아니다. 그러면 π는 원주율이라고 정의할 때는 유클리드 기하를 전제하고 하는 말일 것이다. 무엇을 정의할 때 전제가 필요한 것과 전제가 필요없는 것이 있다면, 오컴의 법칙(Occam's razor)에 따라 전제가 필요없는 것을 따라야 할 터이다. 그렇다면 이러한 정의는 어떻게 할 수 있을까?

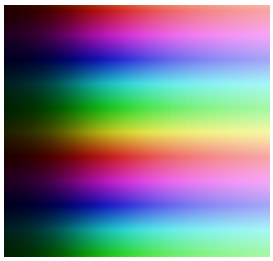

Figure 1: Mosaic image of $\exp(ix)$

오른쪽에 있는 것은 복소지수함수의 모자이크 이미지이다. 패턴이 보이는가? 그렇다면 π의 흔적을 찾은 것이다. 조금 더 명확하게 보고 싶다면 아래의 그래프를 보자.

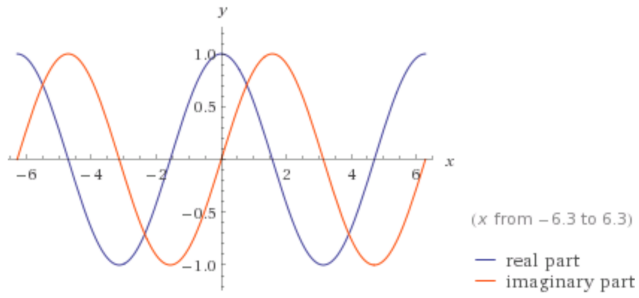

Figure 2: Graph of $\exp(ix)$

이 그래프에서 허수부가 움직이는 곡선을 잘 따라가다 보면 x축과의 교점이 3보다 약간 큰 수라는 것을 알 수 있다. 그리고 파동이 주기를 갖고 진동하고 있음을 볼 수 있다. 이 함수 하나가 π를 정의하면서도 그 중요성을 설명해 줄 수 있다. 왜 여기서 느닷없이 π가 등장하는 것일까?

이 모든 것의 근간에 있고 π와 e가 보편적으로 나타나는 그 원천에 있는 것이 다음의 간단하기 그지없는, 그렇

지만 매우 강력하고도 중요한 방정식이다.

$$f' = f$$

이 간단한 미분방정식의 해가 되는 함수 f는 여러 꼴을 가진다. 대표적으로 $f(x+c)$들도 해가 된다. 미분의 연쇄법칙(Chain rule)을 적용해보면 쉽게 증명할 수 있다. 이와 더불어 상수배를 한 함수의 집합 $\{g \mid g = kf,\ k \in \mathbb{C}\}$도 모두 해가 된다. 이렇게 생각하면 해가 여럿인 것처럼 보인다. 과연 그러할까?

아무 상수 c를 골라보자. 그리고 $f(c) f(x)$와 $f(x+c)$를 비교해보자. 미분방정식은 초기값을 고정하면 유일한 해를 갖게 된다. $f(0) = 1$이라고 해보자. 이제 위의 식은 $x = 0$일 때의 값이 같다. 그러므로 $f(x) f(c) = f(x+c)$라고 할 수 있다. 지수 함수에 익숙한 우리들은 이 등식을 보고 지수함수임을 어렵지 않게 눈치 챌 수 있지만, 이러한 유도 과정을 따라가는 것은 이 함수가 그냥 근본없이 튀어나온 게 아닌 저 미분방정식에 근원을 두고 도출된 결과이며, 위와 같은 성질을 가짐을 증명할 수 있기 때문이다. 이 수식은 매우 중요하므로 다시 한번 쓰도록 한다.

$$f(x+y) = f(x) f(y) \tag{1}$$

다시 처음의 미분방정식을 보자. $f' = f$을 만족하는 함수 f는 아무 상수 c만큼 차이나는 정의역의 값 x를 가져와도 모두 해가 되었다. 이제 $f(0) = 1$로 고정했으니 모든 x에 대해 다음의 식을 만족한다:

$$f(x+c) = f(x) f(c).$$

위와 같은 식을 만족하는 c를 우리는 '주기'라고 부른다. 그리고 이 주기는 매우 독특한 값을 갖는다. 오일러가 가장 아름답다고 했던 식에서 나오는 $c = 2\pi i$이다. 허수가 붙은 이 수가 중요한 이유는 원래의 미분방정식이 그 어떠한 단위도 갖지 않고 다른 것에 그 근거를 두지 않는 아주 간단하고 독립적인 미분방정식이기 때문이다. 그리고 이제 우리는 π를 정의할 수 있다. 이 숫자는 $2ci$가 지수함수의 주기가 되게 하는 가장 작은 양의 실수 c이다.

최근 들어 π보다 2π가 더 중요하고 근본적인 의미를 지니므로 이것을 τ라는 상수로 지정해서 사용해야 한다는 움직임이 있다. 틀린 말은 아니다. 유클리드 기하에서 원주와 지름의 비를 계산하는 것은 조금 어색하다. 우리는 지름보다 반지름을 더 많이 사용하기 때문에 원주와 반지름의 비가 더 자연스럽다. 그리고 이는 2π가 된다. 지수함수의 주기도 2π이다. 이 성질은 함수 f의 푸리에 변환(Fourier transform)을 아래와 같이 정의하면 정규화 상수가 $(2\pi)^{-1/2}$가 되게 한다:

$$\hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, e^{-i\omega x}\, dx.$$

일반적인 주파수를 사용하더라도

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i\omega x}\, dx$$

가 되어 여전히 2π가 남는다. 하지만 관례적으로 π를 많이 써서 이미 굳어진 전통이고 무엇이 정론인가 하는 사소한 문제로 싸우는 것은 수학의 발전에 그다지 도움이 되지 못한다. 더 중요한 것은 이 숫자가 e와 어떤 연관성을 가지며 어떻게 도출되는지를 알고 있는가 하는 것이다.

3 계산가능수(computable numbers)와 π

이번 장에서는 초월수(transcendental numbers)와 대수적 수(algebraic numbers)와 조금 다른 수의 집합을 소개한다. 초월수와 대수적 수도 수 체계에서 꽤 중요한 역할을 하고 있지만, 실용성의 측면에서 조금 다른 숫자의 분류가 필요하다. 이제 더이상 인간의 사고 내에서 수학을 하는 시대가 지나가고 본격적으로 컴퓨터를 활용하여 수학을 응용하는

시대가 되어가고 있기 때문이다. 어떠한 수는 컴퓨터로 연산해낼 수 있고 어떠한 수는 연산해낼 수 없다면, 실수 집합 전부를 대상으로 생각하는 것은 의미가 없다. 그 속에서 π는 어떤 위치에 있는지 알아보도록 하자.

수학은 매우 강한 추상성을 지니고 있다. 개념상으로 생각해볼 수 있는 것도 실제로 연산하려고 하면 불가능한 것도 있고 어느 정도의 오차를 감안하더라도 꽤 정확하게 근사할 수 있는 것도 있다. 이미 실수 체계를 구축하고 칸토어(Cantor)가 실수 집합은 불가산이라는 것을 증명한 순간 실수는 인간이 운용할 수 있는 능력의 한계를 벗어난 것이다. 우주에 존재하는 모든 원자를 동원하더라도 실수 집합을 저장하고 만들어낼 수 없다. 그렇기 때문에 어떻게 하면 오차를 최소화하면서 빠르게 연산해낼 수 있을까 하는 것이 중요해졌다. 수학자들 중에서도 실용성을 추구하는 사람들 혹은 실제로 수학을 응용하고 있는 분야의 연구자들, 예를 들어 물리학자나 공학자들은 실수 체계 중에 어디까지 인간이 사용할 수 있는 것인지 생각해보기 시작했다. 컴퓨터의 등장과 함께 이러한 고민은 수치해석이라는 학문으로 집대성되었다.

요즘 장안의 화제가 되고 있는 인공지능이라는 개념을 최초로 제시한 사람은 '이미테이션 게임'이라는 영화로도 만들어진 '앨런 튜링(Alan Turing)'이다. 지금은 일상이 되어버린 컴퓨터의 초창기 모델을 생각해낸 사람이기도 하다. 그는 컴퓨터 과학자이자 수학자이기도 했는데 '계산가능수에 대하여(On computable numbers)'라는 논문[1]에서 '계산가능수'를 제시한다. 계산가능수는 유한하고 종국에 끝이 나는 알고리즘으로 원하는 오차한계 이내로 연산해낼 수 있는 실수를 말한다. 이제 어떤 새로운 수를 쓸 때 그 수를 작은 오차로 근사시킬 수 있는지가 중요해졌다. π는 당연히 계산가능수이다. 하지만 놀라운 것은 계산가능수라는 집합은 아무리 커봤자 가산집합이며 대부분의 실수는 계산가능하지 않다. 그러니까 우리가 쓰는 숫자는 실수 중 극히 일부분이라는 뜻이며 아무리 빠른 컴퓨터로 연산을 해도 대부분의 실수는 영겁의 시간이 지나도 계산해낼 수 없다는 뜻이 되겠다.

3.1 π는 어떻게 계산할까

중등교육을 거쳐 대학에 들어와서 수학을 다시 배우면 그때까지 배운 수학과는 초점이 조금 다르다는 것을 느낀다. '무엇이냐' 이전에 '존재하냐'부터 따지고 존재한다고 해서 곧바로 '무엇이냐'로 넘어가지 않고 '그럼 유일하냐'를 생각한다. 무턱대고 찾지 않는다는 의미이다. π 역시 그러한 숫자가 있음을 알았다면 우리가 사용할 수 있는 숫자인지 아는 것이 필요하다. 바로 전에 π는 계산가능한 수라는 것을 알았으므로 이제는 그러면 어떻게 계산해낼 것인지가 중요해진다.

역사적으로 π는 다양한 방법으로 근사되었다. 가장 원시적인 방법으로는 이집트인들이 사용했던 22/7이라는 숫자가 있다. 이 숫자는 약 3.142857로 소숫점 아래 2자리까지 정확하므로 그다지 좋은 근사치라고 볼 수 없다. 2세기 무렵의 천체물리학자 프톨레마이오스는 π의 근사값으로 377/120을 썼다. 이것도 약 3.141667로 소숫점 아래 3자리까지 정확하다. 근대적인 수학이 있기 전 사람들은 이와 같이 유리수로 π를 근사시키려고 노력했다. 근대적인 수학이 어느 정도 자리 잡은 후에는 무한합을 이용하여 계산하려는 노력이 많았다. 이런 무한합 꼴은 '근사'한다고 표현하기에는 부적절하다. 왜냐하면 무한개의 항을 모두 더하면 그 값이 정확히 π가 되기 때문이다.

컴퓨터가 생기고 나서 π를 더 정확하게 알려는 노력이 있었고, 여러 알고리즘이 개발되었다. 이러한 알고리즘의 기저에는 수렴 속도에 대한 고민이 깔려있다. 이 말인즉슨 컴퓨터로 무한합을 더해 나갈 때 수용할 수 있는 오차 (tolerance)를 정해놓고 그 오차 이하로 내려가면 중단하게 되는데, 이 중단까지 걸리는 시간이 얼마나 길지 생각해 보아야 한다는 것이다. 아래에는 몇 가지 근대적인 알고리즘을 소개한다.

3.1.1 몬테칼로 방법

몬테칼로 방법 혹은 몬테칼로 시뮬레이션은 어떤 수치를 계산하는 데에 확률변수의 랜덤성을 이용하는 방법 일체를 이른다. π 역시 몬테칼로 방법으로 근사시킬 수 있는데, 이는 π가 어떤 확률변수의 기댓값으로 표현될 수 있기 때문이다. 확률 변수 X, Y가 각각 독립인 Uniform(0,1)을 따른다고 할 때 다음과 같이 정의하자:

$$Z = \begin{cases} 1, & X^2 + Y^2 \leq 1 \\ 0, & \text{otherwise} \end{cases}.$$

그러면 Z의 기댓값 $\mathbb{E}Z$는 $\pi/4$가 됨을 보일 수 있다. 강대수의 법칙에 따라 Z를 무한히 많이 뽑은 후 그것의 산술평균을 내게 되면 $\pi/4$가 된다. 이 과정을 다음과 같은 알고리즘으로 표현할 수 있다.

Algorithm 1 Monte-Carlo method for π

procedure MCPI(τ) ▷ τ is the tolerance
  $n \leftarrow 0$ ▷ set total number to 0
  $c \leftarrow 0$ ▷ set count to 0
  $p_{\text{old}} \leftarrow 0$
  $p_{\text{new}} \leftarrow 0$
  $\epsilon \leftarrow \tau + 1$
  while $\epsilon > \tau$ **do** ▷ while increment between iterations is greater than the tolerance
    $n \leftarrow n + 1$ ▷ increment on each iteration
    $X \leftarrow \text{Unif}\,(0, 1)$
    $Y \leftarrow \text{Unif}\,(0, 1)$
    if $X^2 + Y^2 \leq 1$ **then**
      $c \leftarrow c + 1$ ▷ count the iteration satisfying the condition
    end if
    $p_{\text{new}} \leftarrow c/n$
    $\epsilon \leftarrow p_{\text{new}} - p_{\text{old}}$
    $p_{\text{old}} \leftarrow p_{\text{new}}$
  end while
  $p_{\text{new}} \leftarrow 4 \times p_{\text{new}}$
end procedure

이 방법은 의사코드(pseudocode)이며 어떤 언어로도 옮길 수 있지만 효율적인 코드는 아니다. 각 언어마다 더 효율적인 기능을 제공하니 그것을 이용하여 다시 짤 수는 있다. 그렇다면 속도의 측면에서 이 방법은 좋은 알고리즘인가? 그렇지 않다. 수렴속도가 굉장히 느리다. 매우 흥미로운 근사 방법이긴 하지만 π의 소숫점 전개를 구할 때 몬테칼로 방법을 쓰지는 않는다. 컴퓨터가 없던 시절, 몬테칼로와 비슷한 방법으로 π를 구하려고 했던 사람이 있었다. 이를 '뷔퐁의 바늘(Buffon's needle)' 문제라고 부르는데 1901년 이탈리아의 수학자 마리오 라차리니는 똑같은 실험으로 355/113이라는 매우 근사한 추정치를 구하였다. 몬테칼로 방법이 느린 수렴 속도에도 의미가 있는 이유는 무리수인 π를 유리수로 근사시켜 나가는 이른바 '코쉬수열(Cauchy sequence)'의 예로 볼 수 있기 때문이다. 이로써 유리수 집합은 실수 집합에 대해 조밀(dense)하다는 것이 경험적으로 증명된다.

3.1.2 라마누잔의 방법

이번에 소개할 알고리즘은 인도의 위대한 수학자 라마누잔이 쓴 방법이다. 라마누잔이 제시한 식은 다음과 같다.

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!}{(n!)^4} \frac{26390n + 1103}{396^{4n}}$$

라마누잔은 그의 논문[2]에서 위의 알고리즘 도출 과정에 대해 딱히 설명을 하지 않기 때문에 이런 알고리즘을 어떻게 발견했고 저러한 상수 $k_1, \ldots, k_3$까지 어떤 방법으로 생각해냈는지 알 턱이 없다. 알고리즘의 형태로 옮기게 되면 아래와 같다.

Algorithm 2 Ramanujan's algorithm for π

procedure RJPI(τ) ▷ τ is the tolerance
  $k_1 \leftarrow 26390,\ k_2 \leftarrow 1103,\ k_3 \leftarrow 396$
  $n \leftarrow 0$
  RJpi $\leftarrow 0$
  $\epsilon \leftarrow \tau + 1$
  while $\epsilon > \tau$ **do**
    $\epsilon \leftarrow \frac{(4n)!}{(n!)^4} \frac{k_1 n + k_2}{(k_3)^{4n}}$
    RJpi $\leftarrow$ RJpi $+\epsilon$
    $n \leftarrow n + 1$
  end while
  RJpi $\leftarrow \frac{9801}{\sqrt{8}\,\text{RJpi}}$
end procedure

3.1.3 Chudnovsky 형제의 방법

이번에 소개할 알고리즘은 지금까지 π를 계산해내는데 가장 빠르다고 판명되었다. 앞서 소개한 라마누잔의 알고리즘은 사실 하나가 아니라 꽤 여러가지인데 모두 π^{-1}를 무한합꼴로 표현하고 있으며, 이는 훗날 라마누잔-사토 급수(Ramanujan-Sato series)로 일반화된다. 아무튼 이번 알고리즘은 Chudnovsky 형제가 1989년에 낸 논문[3]에서 처음 소개되었다. 이것 역시 라마누잔-사토 급수의 일종으로 π^{-1}를 무한합으로 표현하고 있다.

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} (-1)^k \frac{(6k)!\,(545140134k + 13591409)}{(3k)!\,(k!)^3\,(640320)^{3k+3/2}}$$

그리고 이것을 다시 의사코드(pseudocode)로 쓰게 되면 아와 같이 나타낼 수 있다. 이것이 매우 놀라운 것은 매 반복 시행—while 루프를 말함—마다 소숫점 아래 약 14자리씩 맞아떨어져 나간다고 하니 정말 빛의 속도로 수렴함을 알 수 있다.

Algorithm 3 Chudnovsky brothers' algorithm for π

procedure CBPI(τ) ▷ τ is the tolerance
  $k_1 \leftarrow 545140134,\ k_2 \leftarrow 13591409,\ k_3 \leftarrow 640320,\ k_4 \leftarrow 100100025,\ k_5 \leftarrow 327843840,\ k_6 \leftarrow 53360$
  CBpi $\leftarrow 0$
  $n \leftarrow 0$
  $\epsilon \leftarrow \tau + 1$
  while $\epsilon > \tau$ **do**
    $\epsilon \leftarrow (-1)^n \dfrac{(6n)!\,(k_2 + nk_1)}{(n!)^3\,(3n)!\,(8k_4k_5)^n}$
    CBpi $\leftarrow$ CBpi $+\epsilon$
    $n \leftarrow n + 1$
  end while
  CBpi $\leftarrow \dfrac{k_6\sqrt{k_3}}{\text{CBpi}}$
end procedure

3.2 기타

위에서 소개한 알고리즘 이외에도 다양한 알고리즘들이 있다. 예를 들어 arctan를 전개한 그레고리 급수(Gregory's series)와 $x = 1$일 때의 그레고리 급수인 라이프니치 공식(Leibniz formula)도 있다. 하지만 매우 느려서 잘 사용하지는 않는다. 다양한 알고리즘 이외에 그러면 컴퓨터로 관련된 계산을 할 때 π를 매번 계산할까? 그렇지 않다. 대부분의 현대적인 프로그래밍 언어는 C로 만들어졌는데 C에서는 π를 `math.h`라는 헤더파일에 `#define M_PI 3.14159265358979323846`와 같이 상수로 선언/정의되어 있다. 그 이유는 컴퓨터의 메모리는 무한하지 않아서 π를 무한한 소숫점까지 저장할 수 없으며 원하는 정확도까지 끊어서 근사값으로 쓸 수밖에 없기 때문이다. 위에서 C가 정의해놓은 상수는 `double` 자료형에서 최대로 저장할 수 있는 만큼이다. 더 많은 자리 수를 쓰고 싶다면 스스로 비트 수를 더 할당하여 새로운 상수로 저장하는 수밖에 없지만 대부분의 과학적 컴퓨팅에서 그 정도 정확도는 요하지 않는다.

4 π는 정말 신비로울까

예로부터 알 수 없는 것들에는 신비로움이 따라다니기 마련이다. 숫자 역시 예외일 수 없다. 피타고라스는 만물의 근원이 수라고 하였으며 그중에서도 자연수만을 섬겼다. 그래서 무리수가 등장하면 악마의 수라고 하면서 배척하기 일쑤였다. 도처에서 보이는 π라는 숫자가 무리수라는 것이 증명되면서 사람들은 새로이 등장한 이 '신비로운' 숫자에 온갖 미신을 투영하기 시작했다. 그리하여 π의 소숫점 전개에는 모든 존재하는 수열이 들어가 있다느니, 그 숫자들이 랜덤이라느니, 삼라만상의 근원이 담겨있다느니, π의 소숫점 전개에 마그나 카르타(Magna Carta)가 있다느니 하는 주장을 하기 시작한다. 이번 장에서는 그 미신을 파헤쳐 보도록 하자.

π의 소숫점 전개가 정말 임의의 수열일까? 이에 대해 쓰여진 논문[4]에 따르면 어떠한 진법으로 나타내더라도 그리고 자릿수를 어떻게 골라내더라도 그 빈도가 모두 같다면 그 숫자를 '정규수(normal numbers)'라고 부른다. 그리고 거의 대부분의 실수는 정규수이다. 숫자의 정규성을 따지는 이유는 우리가 흔히 '랜덤'이라고 말하면 생각하는 것이 특정 숫자가 현격히 더 많이 발생하거나 적게 발생하는 것이 아니라 예측할 수 없도록 모두가 일정한 확률을 가지고

발생하는 현상이기 때문이다. 그렇다면 어떠한 자릿수 뭉텅이를 가져오더라도 그것을 예측할 수 없게끔 나머지 숫자 뭉텅이와 같은 빈도로 들어있어야 그 숫자가 '랜덤'이라고 말할 수 있다. 이것을 가장 잘 대변해주는 숫자의 특성이 바로 정규성이다.

그런데 거의 모든 실수가 정규수라니... 이에 따르면 예를 들어서 2130394는 π에만 들어있는 것이 아니라 모든 실수의 소숫점 전개에 들어있다. π가 유달리 특별할 것이 없는 것이다. 그러니까 이것을 다른 말로 표현한다면 대부분의 유난히 π에만 이런 랜덤성을 부여하여 우대해주는 현상은 잘못된 믿음에 기인한 것이라 할 수 있겠다.

5 π와 철학

지금까지 숫자 π와 연관된 주제들을 간단하게 두루두루 다루어보았다. 하지만 이러한 고민을 잘 더듬어 따라가다 보면 꽤나 심오한 철학적 '테마'들과 연결되어 있다는 것을 알 수 있다. 왜냐하면 이 모든 수학적 모델링은 우주가 일정한 규칙성을 따르고 있다는 전제 아래 이루어지고 있기 때문이다. 자연 현상에 물리학적 설명을 덧붙이는 행태도 바로 그런 현상을 설명하고 있는 일정한 규칙이 존재하고 있다고 믿기 때문이다. 이 전제가 참인지에 대한 명확한 답은 없다. 그저 일상적인 관찰을 통해 그 설명이 맞다고 기존의 믿음을 강화해나갈 뿐이다. 하지만 때때로 그러한 규칙성을 벗어난 현상을 보게 되고 때로 자신이 예상했던 결과가 빗나가는 것도 보게 되면서 자연현상이 규칙성뿐만 아니라 임의성(randomness, 랜덤성)도 따름을 알게 되었다. 이에 자연스러운 질문을 할 수밖에 없게 된다. 어떻게 만물을 구성하는 자연이 보기에 상반되는 규칙성과 랜덤성을 동시에 따를 수 있는가.

이에 대한 통계학은 다음과 같이 대답한다. 자연 현상들이 우연히 발생하는 듯이 보이지만 사실상 불규칙적인 일련의 현상 속에는 일정한 통계적 규칙이 숨어있다고. 그리고 또 다른 설명은 무한과 유한을 대비함으로써 얻을 수 있다. 그러니까 무한한 흐름 속에 우리는 유한한 일부분만을 관찰하게 되고, 무한한 전체는 정말 랜덤일 수 있지만 우리가 관측한 일부분은 규칙성을 띨 수 있다는 것이다. 두 번째 설명이 시사하는 바는 생각보다 크다. 왜냐하면 우주가 진화해나가는 것을 하나의 시간적으로 무한한 프로세스라 생각한다면, 그 전체적인 과정이 아무리 랜덤하다 하더라도, 즉 그 어떤 확률 법칙도 따르지 않는다 하더라도, 국지적으로 보면 어떤 규칙성을 따르고 있기 때문에 예측을 아예 할 수 없지는 않다는 것을 의미하기 때문이다. 또한 경험적으로도 천체과학자들이 인류를 다른 행성으로 보낼 때 행성의 움직임을 예측하여 유인 우주선을 띄워 안착시키기 위해 필요한 계산을 아주 잘 할 수 있기 때문에, 우리는 '예측'에 한해서 불가능하지 않다는 것을 매우 잘 알고 있다. 그렇지만 이 모든 예측은 특정한 오차 이내에서만 가능하며 그 국지성이 깨질수록, 즉 위의 예로 들자면 시간이 길어지면 길어질수록 그 오차가 커져서 결국엔 예측이 불가능한 정도에 이르게 된다.

모든 현상을 규칙성을 부여해 설명하려고 했던 생각의 방식을 결정론(determinism)이라 부르며 그 반대를 랜덤성이라 부른다. 우리가 세계를 바라보는 시선은 결정론에서 랜덤성으로 흘러왔고 그 시선은 고스란히 수학에 담겼다. 그 이유는 단순하다. 인류는 수학을 사용해서 세상을 재구성하려 했기 때문이다. 당연히 세상을 바라보는 시각이 바뀌면 수학을 사용하는 방식도 바뀌게 마련이다. π와 관련된 계산가능성, 알고리즘, 랜덤성 등은 바로 이러한 맥락에서 중요하다. 인간의 진리추구 능력이 어디까지 미칠 수 있는가. 유한한 자원으로 계산해낼 수 있는가. 그 방법은 무엇인가. 마지막으로 그 수는 얼마나 랜덤인가. 따지고 보면 π는 온 우주를 담고 있지는 않지만 인류의 지성사를 담고 있는 것이 아닐까 싶다.

References

[1] Turing, Alan (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem", _Proceedings of the London Mathematical Society_ 42 (1):230-265. DOI=10.2307/2268810

[2] J. M. Borwein, P. B. Borwein, and D. H. Bailey. 1989. Ramanujan, modular equations, and approximations to Pi or how to compute one billion digits of Pi. Am. Math. Monthly 96, 3 (March 1989), 201-219. DOI=http://dx.doi.org/10.2307/2325206

[3] Chudnovsky, David V.; Chudnovsky, Gregory V. (1989), "The Computation of Classical Constants", Proceedings of the National Academy of Sciences of the United States of America 86 (21): 8178–8182, doi:10.1073/pnas.86.21.8178, ISSN 0027-8424, JSTOR 34831, PMC: 298242, PMID 16594075

[4] Ferdinánd Filip, Jan Šustek (2010). "An elementary proof that almost all real numbers are normal", Acta universitatis sapientiae. Mathematica 2(1), 99-110, `http://www.emis.de/journals/AUSM/C2-1/math21-8.pdf`

A Python code for algorithms

A.1 Monte-Carlo algorithm for π

Listing 1: Monte-Carlo algorithm

```
from scipy.stats import uniform
import math
from __future__ import division
def MCpi(n):
  return 4.0 * sum(uniform.rvs(size = n)**2.0 + uniform.rvs(size = n)**2.0 <= 1.0) / n
```

A.2 Ramanujan algorithm for π

Listing 2: Ramanujan's algorithm

```
import math
from __future__ import division
def RJpi(tau):
  n = 0.0
  k1 = 26390.0
  k2 = 1103.0
  k3 = 396.0
  eps = tau + 1.0
  rjpi = 0
  epsList = []
  while(eps > tau):
    eps = math.factorial(4.0*n)/((math.factorial(n))**4.0) * (k1 * n + k2)/(k3**(4.0*n))
    epsList.append(eps)
    rjpi += eps
    n += 1
  rjpi = 9801.0/(math.sqrt(8)*rjpi)
  return rjpi, epsList
```

A.3 Chudnovsky brothers' algorithm for π

Listing 3: Chudnovsky brothers' algorithm

```
import math
from __future__ import division
def CBpi(tau):
  k1 = 545140134.0
  k2 = 13591409.0
  k3 = 640320.0
  k4 = 100100025.0
  k5 = 327843840.0
  k6 = 53360.0
```

```
cbpi = 0.0
n = 0.0
eps = tau + 1
while eps > tau:
  eps = ((-1.0)**n) * math.factorial(6.0*n) * ( k2 + n * k1) / ((math.factorial(n)**3.0) * math.
      factorial(3.0 * n) * ((8.0 * k4 * k5)**n))
  cbpi = cbpi + eps
  n += 1
return k6 * math.sqrt(k3) / cbpi
```
