미분적분학1을 위한 SageMath™ 기본 사용법

강원대학교 자연과학대학
수학과

버젼 beta 0.88 (16/02/09)

목 차

제 1장

SageMath 소 개

1.1 수학 계산 소프트웨어와 sage

수학 계산 소프트웨어는 컴퓨터 알고리즘을 모아놓고 사용자들이 이를 조합하여 사용할 수 있게 만든 컴퓨터 프로그램으로 프로그래밍 언어처럼 사용 할 수 있게 구성되어 있는 경우가 많다. 검증된 수학 이론을 바탕으로 이를 프로그래밍 하여 복잡한 수학 계산을 자동으로 수행한다. 많은 수학 계산 소프트웨어가 개발 되었고 몇몇의 프로그램은 산업과 많은 연구 분야에서 사용하고 있다. 아래의 주소에서 수학 계산 소프트웨어의 종류를 확인 할 수 있다.
https://en.wikipedia.org/wiki/List_of_computer_algebra_systems

sage는 오픈소스 개발 환경으로 강력하고 유명한 Python으로 프로그래밍 되어 있는 수학 계산 프로그램이다. 따라서 Sage는 Python프로그래밍으로 확장 가능하다. Sage는 Numpy, SciPy, matpoltlib, Sympy, Maxima, GAP, FLINT, R과 같은 현재까지 개발된 많은 공개 수학 계산 소프트웨어의 기능을 포함하고 있고 현재 약 100개의 공개 소프 패키지로 구성되어 있고 패키지들이 완전히 통합되어 있다. 이를 기반으로 순수 수학과 응용 수학 뿐만 아니라 공학에 이르는 많은 분야에서 활용 가능 하다.

SageMath는 사용하는 컴퓨터에 직접 설치하여 운용이 가능할 뿐만 아니라 서버에 설치하면 인터넷 웹 브라우저를 이용하여 구동할 수 있는 특징을 가지고 있는데 단순하게 이용 가능한 Cell-server와 서버 안에 개개인의 공간에서 작업 수행과 공유가 가능한 Notebook이라고 불리는 유저인터페이스를 제공한다. SageMath 소프트웨어의 개발 목표를 교육과 연구 현장에서 이용하는 많은 상용 수학 계산 소프트웨어를 대체하는 것이다.

1.2 기본 용법

1.2.1 명령어와 변수 선언

내장되어있는 Sage명령어 함수 대부분의 표현들은 영문 소문자를 기반으로 사용한다. **plot expand print**같은 내장함수와 **sin def gcd max**와 같은 명령들이 대표적인 예로 실제 수학에서 사용하는 언어를 바탕으로 구성되어 있다. Sage는 대소문자를 구분하기 때문에 변수를 선언할 때는 대소문자를 섞어 사용가능 하다. **LX1 mYPlot Solu**같은 대소문자를 혼용한 변수를 사용할 수 있다.

1.2.1 괄호의 사용법

Sage에서 사용하는 괄호는 사용법에 따라 명확히 구분 되어야 한다.

● 소괄호 ( ) : Sage문법의 기본이 된다. 다음의 명령어를 살펴보자.

```
sage : plot(sin(x),x,0,2*pi)
```

plot라는 명령어는 기본적으로 plot( )방식으로 구성되어 있고 이 안에 함수와 그림을 그릴 구간 그리고 여러 가지 부가 옵션을 지정할 수 있다. 함수와 그림을 그릴 구간 사이는 ,(콤마)로 구분 되어 있고 이를 수행할 구간을 표시하는 **,x,0,2*pi)**에도 콤마와 괄호를 이용하고 있다. 또한 sin함수를 이용하는데도 sin(x) 방식으로 사용하고 있다. 수식이나 함수를 사용 할 때도

```
sage : f(x)=(x^2*2*x*(x+1))*((x^3)*sin(x))
```

등으로 일생 상활에서는 중괄호와 대괄호도 모두 사용하여 수식을 만들지만 Sage에서는 오직 소괄호만 이용할 수 있다.

● 대괄호 [ ] : 대괄호는 데이터 구조를 구분하는데 이용한다. 즉 컴퓨터 프로그램에서 배열 개념인 리스트를 만드는데 대괄호와 콤마를 사용한다. Sage에서 리스트는 원소들이 모여있는 공간을 얘기하는데 $[e_{1,}e_{2,}e_{3,}...e_{n}]$식으로 구성한다. 예를 들면

```
sage : v=[1,3,5,7,9]
sage : v
  [1,3,5,7,9]
```

v라는 공간에 1 3 5 7 9 5개의 자료를 넣어 리스트로 만든 모습이다. 리스트를 이용하면 행렬 형태의 데이터 구조도 만들 수 있다. 예를 들어

```
sage : A=[['a','b','c'],[1,2,3],[6,7,8]]
sage : table(A)
  a  b  c
  1  2  3
  6  7  8
```

A에 3행3렬 형태의 리스트를 만든 모습이다.

1.2.2 띄어쓰기와 =(등호) 사용법

Sage에서 띄어쓰기는 어떠한 의미도 가지지 않는다. 하지만 내장 함수나 변수 사이에 띄어쓰기를 하면 제대로 인식하지 않는다. 예를 들어

```
sage : sin(pi)
sage : sin (pi)
sage : sin( pi )
sage : sin(pi )
```

이 명령어 들은 서로 같은 명령을 수행하기 되지만

```
sage : sin(p i)
sage : s in(pi)
sage : si n(pi)
```

이 명령어를 사용하면 오류가 발생한다.

=등호는 수학에서는 같다라는 의미를 사용하지만 컴퓨터 프로그래밍에서는 주로 대입의 의미로 사용되므로 "정의" "대입" 등의 뜻으로 이해하면 편하다. 예를 들어

```
sage : a=3
```

명령은 a와 3이 같다는 의미보다 a라는 저장소에 3이라는 값을 지정한다고 생각하면 된다.

```
sage : f(x)=x^2
```

명령은 x를 변수로 갖는 f(x)라는 함수를 x^2(엑스 제곱)으로 정의한다는 의미이다. 또한 다음과 같이 쓸 수도 있는데

```
sage : p=plot(sin(x),(x,0,pi))
```

이는 $\sin(x)$함수를 0부터 2π까지 그리는 자체를 p라고 지정한다는 의미이다.

1.2.3 결과값 출력

Sage에서 명령을 입력한 다음 Shift+Enter 버튼을 누르면 명령어를 실행하게 된다.

```
sage : 3*4 3+4
```

한 줄에 2개의 명령어를 적으면 에러가 나며 Notebook이나 Cell-server환경에서

```
sage : 3*4
       3+4
```

두 줄에 명령어를 입력하게 되면 마지막으로 수행한 명령만을 값으로 반환한다. 모든 값을 도출하고 싶으면 ;(세미콜론)을 이용해 한 줄에 적으면 된다.

```
sage : 3+4; 3*4
  7
  12
```

=(등호)를 이용하여 변수나 함수를 정의하면 어떠한 결과도 출력되지 않으며 변수명을 지정하면 그 값을 반환한다.

```
sage : a=3+4; b=4+5
sage : b; a
  9
  7
```

1.2.4 주석

#을 이용하여 주석을 달 수 있고 # 이후의 명령은 수행하지 않는다.

```
sage : # This command plot the graph of $sin$ functions in red color
sage : g=plot(sin(x),x,-3,3,figsize=3,color='red')
```

1.3 대수적인 계산

1.3.1 기본 대수 부호

Sage에서 이용하는 일상 대수 계산에 대한 부호는 다음과 같다.

- **+** (덧셈)
- **-** (뺄셈)
- ***** (곱셈)
- **/** (나눗셈)
- **^** (거듭 제곱)
- **%** (나머지)

```
sage : 10%3
  1
```

- **//** (몫)

```
sage : 10//4
  2
```

- **factorial()** (팩토리얼)

```
sage : factorial(3)
  6
```

1.3.2 실수의 십진법 수치 표현

Sage는 임의의 실수에 대해 다양한 정확도의 십진수 수치 표현이 가능하다. 이를 구현하기 위해서는 명령어 **N**을 사용한다. 사용법은 다음과 같다.

```
N(실수, digits=표현할 자릿수)
```

예를 들면

```
sage : pi
 pi
```

pi는 원주율 π의 내장함수로 Sage에서는 실수로 표현하지 않고 원주율 자체로 인식한다.

```
sage : N(10*pi,digits=20)
 31.415926535897932385
```

10π를 20개의 숫자를 이용하여 십진법으로 표현하라는 명령이다.

1.3.3 ==(이중 등호)의 사용법과 대수적으로 방정식의 해 구하기

이중 등호(==)는 등호를 중심으로 양쪽 표현의 같음의 사실을 나타내준다. 예를 들어

```
sage : x=2
sage : x==3
 False
sage : x==2
 True
```

이중 등호를 이용하여 방정식의 해를 구하는 명령어를 작성할 수 있다. 먼저 명령어 solve를 소개한다. 이 명령어는 해를 구하라는 명령으로 사용법은 다음과 같다.

```
solve(equation, variable)
```

예를 들어 $x^2+x+2=0$의 해를 구하는 명령은 다음과 같다.

```
sage : solve(x^2+x+2==0,x)
 [x == -1/2*I*sqrt(7) - 1/2, x == 1/2*I*sqrt(7) - 1/2]
```

결과를 보면 Sage는 기본적으로 복소수 집합에서 해를 찾음을 알 수 있다. 연립 방정식의 해를 찾는 명령은 다음과 같이 사용한다.

```
solve([eqs1,…,eqsn], var1,…,varn)
```

```
sage : x,y=var('x,y')
sage : solve ([2*x-y==3, x+4*y==-2] ,x,y)
 [[x == (10/9), y == (-7/9)]]
```

1.3.2 대수 계산을 위한 유용한 명령어

- 함수를 간단히 하는 명령으로 .simplify_full()을 사용 할 수 있다.

```
function.simplify_full()
```

```
sage : (sin(2*x)).simplify_full()
 2*cos(x)*sin(x)
```

● 변수가 x인 함수에 어떤 값 c를 대입하는 방법은 명령어 **.substitute(x=c)**을 통해 사용 가능하다.

```
function.substitute(x=c)
```

다변수 함수도 다음과 같이 사용 가능 하다.

```
f(x,y).substitute(x=c,y=d)
```

● 함수 $f(x)$를 구간 (a,b)에서는 $f(x)=f1$와 같이 정의하고 구간 (c,d)에서는 $f(x)=f2$로 정의하고 싶을 때 **piecewise**라는 명령어를 사용한다. 소문자로 되어 있는 이 명령어 외에 맨 앞의 글자가 대문자인 **Piecewise**라는 명령어도 있는데 두 명령은 비슷한 역할을 하지만 대상이 다르고 나중에 더 확장된 명령을 하기 위해서는 **piecewise**명령어가 더 쓰임새가 많으므로 이것을 이용하도록 하겠다.

```
piecewise([((a,b),f1), ((c,d),f2)])
```

● 방정식 $f(x)=0$의 x에 관한 해를 구할 때 위에서 언급한 것과 같이 **solve**명령을 사용한다.

```
solve(f(x)==0,x)
```

● y를 x에 관한 함수로 정의하고 싶으면 다음과 같이 사용한다.

```
y(x)=function('y',x)
```

● 자연수를 소인수분해 하거나 수식을 인수분해 할 때 **factor**명령을 사용한다.

```
factor(number)                    factor(function)
```

```
sage : factor(910)
 2*5*7*13
sage : factor(x^5+11*x^4+46*x^3+90*x^2+81*x+27)
 (x + 3)^3*(x + 1)^2
```

● 수식을 전개할 때는 **expand**명령을 사용한다.

```
expand(expression)
```

```
sage : expand((x+1)^5)
 x^5 + 5*x^4 + 10*x^3 + 10*x^2 + 5*x + 1
```

● 등식에서 좌변이나 우변에 있는 항목만 지정하고자 할때는 **rhs()**(우변) **lhs()**(좌변)을 이용 한다.

equation.rhs()	equation.lhs()

```
sage : var('y')
sage : a=(x^3==y^2)
sage : a.rhs()
 y^2
sage : a.lhs()
 x^3
```

1.4 함수

Sage에서 함수를 지정하는 방법은 2가지가 있다. 유리 함수 $f(x)=\frac{x^2-x+4}{x-1}$을 정의하는 방법을 예로 들어 살펴보자.

1.4.1 함수를 정의하는 대표적인 2가지 방법

● 방법 1 : 우리가 일상생활에서 함수를 정의하는 것과 같이 직관적으로 하는 방법이다.

```
sage : f(x)=(x^2-x+4)/(x-1)
sage : f(x)
 (x^2 - x + 4)/(x - 1)
```

방금 정의한 함수 $f(x)$의 $x=5$에서의 함숫값을 알고 싶다면 우리가 일생 수학에서 적는 방법처럼 **f(5)**라고 표현하면 된다.

```
sage : f(5)
 6
```

● 방법 2 : Sage는 Python을 기반으로 작성된 프로그램이므로 Python에서 사용하는 함수의 정의를 그대로 사용할 수 있다. 여기에 사용되는 명령은 함수의 이름을 지정하는데 사용하는 **def**명령어와 이 정의의 반환 값을 알려주는 **return**명령으로 구성되어 있다. **def**명령을 사용할 때는 뒤에 반드시 :(콜론)을 붙여줘야 한다.

```
sage : def g(x): return (x^2-x +4) /(x -1)
sage : g(x)
 (x^2 - x + 4)/(x - 1)
sage : g(5)
 6
```

1.4.3 미분적분학에 쓰이는 함수 정리

● 다항함수 : 곱셈 표시*에 유의하여 함수를 정의한다.

예) $f(x)=x^3+3x^2+2x+1$

sage : **f(x)=x^3+3*x^2+2*x+1**

● 무리함수 : 제곱근을 의미하는 명령어 **sqrt()**를 사용하거나 지수에 분수 형태를 사용하여 만들 수 있다.

예) $f(x)=\sqrt{2x+1}$

sage : **f(x)=sqrt(2*x+1)**

예) $f(x)=(x+1)^{\frac{1}{3}}=\sqrt[3]{x+1}$

sage : **f(x)=(x+1)^(1/3)**

● 유리함수 : 분모와 분자에 괄호를 주의하여 만들면 된다.

예) $f(x)=\dfrac{x-1}{x^2+x+1}$

sage : **f(x)=(x-1)/(x^2+x+1)**

● 삼각함수 : Sage에 내장되어 있는 명령어를 이용한다.

예) $f(x)=\sin x+\cos x+\tan x$

sage : **f(x)=sin(x) + cos(x) + tan(x)**

예) $f(x)=\sec x+\csc x+\cot x$

sage : **f(x)=sec(x) + csc(x) + cot(x)**

● 지수함수 : 거듭 제곱 표현인 ^을 사용하여 만든다. 지수함수의 밑이 e(자연상수)인 경우 명령어 **exp(x)** 함수를 이용할 수 있다.

예) $f(x)=3^{x^2+3x}$

sage : **f(x)=3^(x^2+3*x)**

예) $f(x)=e^{x+1}$

sage : **f(x)=exp(x+1)**

● 로그함수 : Sage에 저장되어 있는 명령어 **log(x)**는 밑이 e(자연상수)인 로그함수를 뜻하며 명령어 **log(x,a)**는 밑이 a인 로그 함수를 의미한다.

예) $f(x)=\log_2 x=\dfrac{\ln x}{\ln 2}$

sage : **f(x)=log(x,2)**

예) $f(x)=\ln(x^2+x+1)$

sage : **f(x)=log(x^2+x+1)**

● 역삼각함수 : Sage에 내장되어 있는 명령어를 이용한다.

예) $f(x)=\sin^{-1}x+\cos^{-1}x+\tan^{-1}x$

```
sage : f(x)=asin(x) + acos(x) + atan(x)
```

1.4.2 구간별로 다르게 정의된 함수를 정의하는 방법

구간별로 다르게 정의된 다음의 함수를 정의하는 방법에 대해 알아보자

$$f(x)=\begin{cases}-x, & \text{if } |x|<1\\ x, & \text{if } |x|\geq 1\end{cases}$$

먼저 1.4.1의 방법1을 이용해 함수를 정의하기 위해서는 위에서 소개된 **piecewise** 명령을 사용할 수 있다. 이 함수는 구간 $(-\infty,-1]$와 $[1,\infty)$에서는 함수 x로 구간 $(-1,1)$에서는 함수 $-x$로 정의된다. 이를 이용해서 다음과 같은 명령어를 생각해 보자 참고로 무한대 기호 ∞의 표현은 영문 소문자 o를 2개 붙여 **oo**로 사용 한다.

```
sage : f=piecewise([((-oo,-1), x), ((-1,1), -x), ((1,oo), x)])
sage : f(x)
 piecewise(x|-->x on (-oo, -1), x|-->-x on (-1, 1), x|-->x on
 (1, +oo); x)
sage : f(-1.5); f(-0.5); f(0.5); f(1.5)
 -1.50000000000000
 0.500000000000000
 -0.500000000000000
 1.50000000000000
```

몇몇 지점에 있는 함숫값을 반환했더니 정의가 잘 이루어진 것처럼 보인다. 과연 그럴까?

```
sage : f(1)
 Traceback (click to the left of this block for traceback)
 ...
 ValueError: point 1 is not in the domain
```

두 구간 사이에 있는 점 $x=-1$과 $x=1$에서는 함수가 정의되지 않았기 때문에 점 $x=-1$과 $x=1$에 대해서도 함수를 정의해 줘야 한다. (Sage에서는 반폐구간 $[a,b)$을 지원하지 않는다.)

```
sage : f=piecewise([((-oo,-1),x),((-1,1),-x),((1,oo),x),([-1,-1],x),([1,1], x)])
sage : f(x)
 piecewise(x|-->x on (-oo, -1), x|-->-x on (-1, 1), x|-->x on
 (1, +oo), x|-->x on {-1}, x|-->x on {1}; x)
sage : f(-1); f(1)
```

```
-1
1
```

위와 같은 함수는 구간 $(-1,1)$에서는 함수 $-x$이고 이를 제외한 나머지 실수 영역에서는 함수x이다. 이를 이용하여 다음과 같이 정의할 수도 있다.

```
sage : f1=piecewise([((-1,1),-x)])
sage : f=f1.extension(x)
sage : f(x)
 piecewise(x|-->-x on (-1, 1), x|-->x on (-oo, -1] + [1, +oo); x)

sage : f(-3/2); f(-1); f(-1/2); f(1/2); f(1); f(3/2)
 -3/2
 -1
 1/2
 -1/2
 1
 3/2
```

위 명령은 먼저 구간 $(-1,1)$에서 함수 $-x$를 지정한 함수명을 $f1(x)$를 정의하고 **extension**명령을 이용하여 이를 제외한 나머지 영역을 함수 $f(x)$로 지정하는 방법이다. 용법은 다음과 같다.

```
function.extension(extension function, extension domain)
```

Extension domain항목은 입력하지 않으면 현재 정의되어 있는 영역이외의 모든 실수가 된다.

1.4.1의 방법2를 이용해 위 함수를 정의하기 위해 조건명령어인 **if**를 사용하겠다. 또한 절댓값을 표현하기 위해 절댓값을 출력하는 내장함수인 **abs**를 사용한다.

```
sage : def f(x):
     :       if abs(x)<1:
     :               return -x
     :       else:
     :               return x

sage : f(-3/2); f(-1); f(-1/2); f(1/2); f(1); f(3/2)
 -3/2
 -1
 1/2
 -1/2
 1
 3/2
```

제 2장

함수의 그래프와 극한 및 연속

2.1 함수의 그래프 그리기

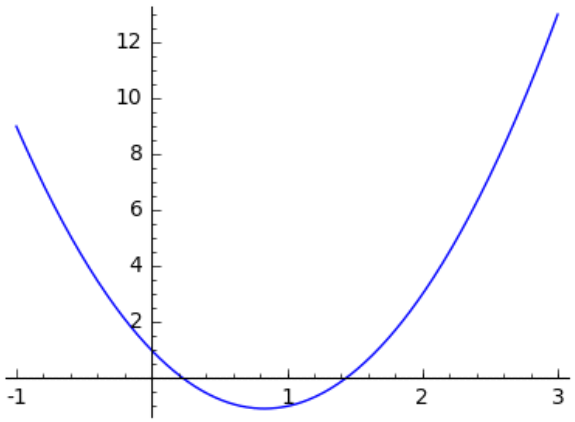
2.1.1 Plot 명령의 기초

이번 내용에서는 Sage를 이용하여 함수의 그래프를 그리는 방법과 다양한 옵션을 활용하는 방법에 대해 알아볼 것이다. 함수 $y=f(x)$의 그래프를 $[a,b]$구간에서 그리는 가장 기본적인 명령어 구문은 다음과 같다.

```
plot(f(x),x,a,b)
```

예를 들어 함수 $f(x)=3x^2-5x+1$을 그래프를 구간 $[-1,3]$에서 그리는 명령과 결과는 다음과 같다.

sage : **plot(3*x^2-5*x+1,x,-1,3)**



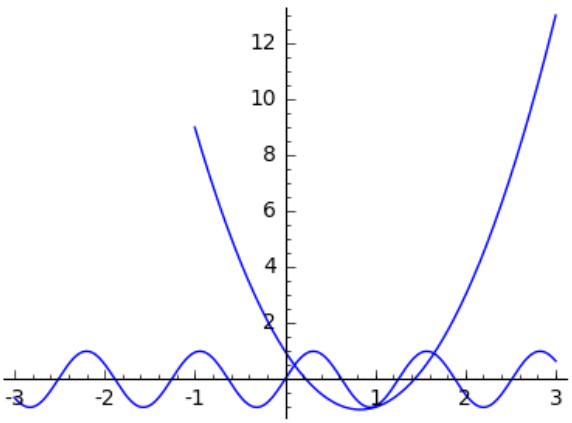
2.1.2 2개 이상의 함수 그래프를 동시에 그리기

2개 이상의 그래프를 그리기 위해 **plot**명령에서 지원하는 다음 구문을 먼저 알아보겠다.

```
plot((f(x),g(x)),x,a,b)
```

이 명령은 함수 $f(x)$와 $g(x)$를 구간 $[a,b]$에서 한꺼번에 그리라는 명령이다. 하지만 여러 개의 함수의 그래프를 유연하게 관리하기 위해 다음의 방법을 많이 사용한다.

```
sage : p=plot(3*x^2-5*x+1,x,-1,3)
sage : q=plot(sin(5*x),x,-3,3)
sage : p+q
```



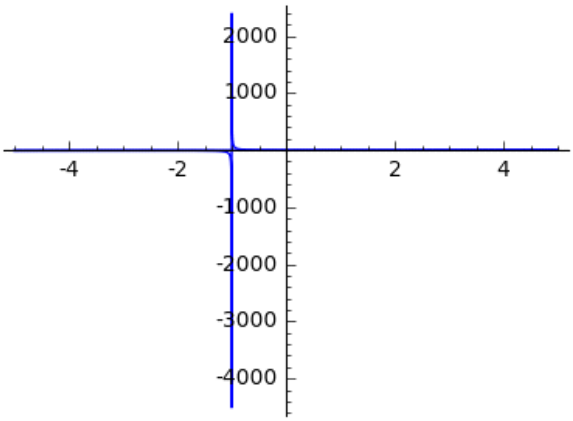
이 명령은 함수 $3x^2-5x+1$을 $[-1,3]$에서 그린 그래프를 **p**라고 지정하고 함수 $\sin(5x)$를 구간 $[-3,3]$에서 그린 그래프를 **q**라고 지정한 다음 2개의 그림을 한꺼번에 그리라는 의미로 연산자 **+**를 사용하였다. 이처럼 Sage에서는 실수, 함수, 그래프 등 실제 수학에서 사용하는 개념의 객체를 구성할 수 있게 정의해 놓고 객체 사이에 명령어와 연산을 정의하여 사용할 수 있게 해 놨다. 어떤 객체의 종류를 확인하는 방법은 다음과 같다.

```
parent(object)
```

```
sage : p=plot(3*x^2-5*x+1,x,-1,3); parent(p)
<class 'sage.plot.graphics.Graphics'>
```

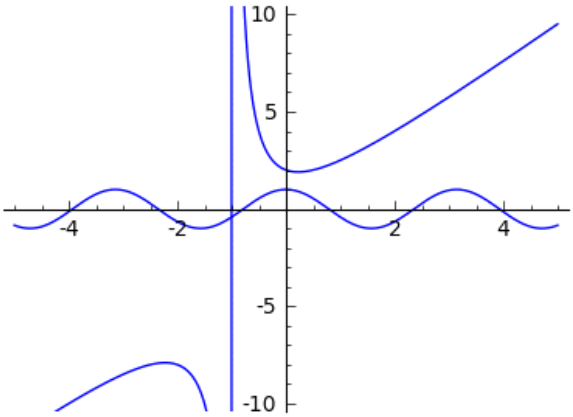
2.1.3 그래프에서 치역의 범위를 조정하기

```
sage : plot( ((2*x^2+x+2)/(x +1),cos(2*x)), x,-5,5)
```



두 함수 $\frac{2x^2+x+2}{x+1}$와 $\cos 2x$를 범위 $[-5,5]$에서 동시에 그리라는 명령을 위와 같이 실행해 보면 그래프를 보기가 불편함을 알 수 있다. 과도하게 넓게 구성되어 있는 치역의 영역을 확장하기 위해 옵션 **ymin**과 **ymax**를 사용하겠다.

sage : **plot(((2*x^2+x+2)/(x +1),cos(2*x)), x,-5,5, ymin=-10, ymax=10)**



치역의 범위가 -10부터 10까지 설정되고 이에 맞게 그래프가 그려진 것을 알 수 있다.

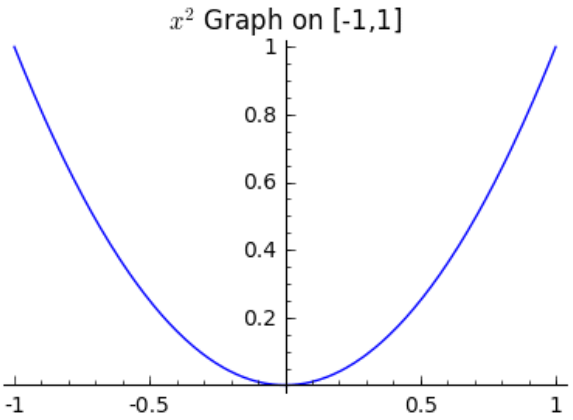
2.1.4 plot의 대표적인 옵션

plot명령과 함께 쓰이는 대표적인 옵션에 대해 알아보자.

- **title**: 그래프에 제목을 표시할 때 사용 된다.

```
plot(f(x),x,a,b,title='Title')
```

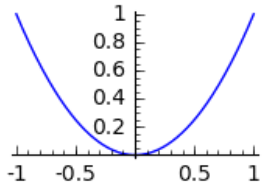
sage : **plot(x^2,x,-1,1, title='x^2 Graph on [-1,1]')**



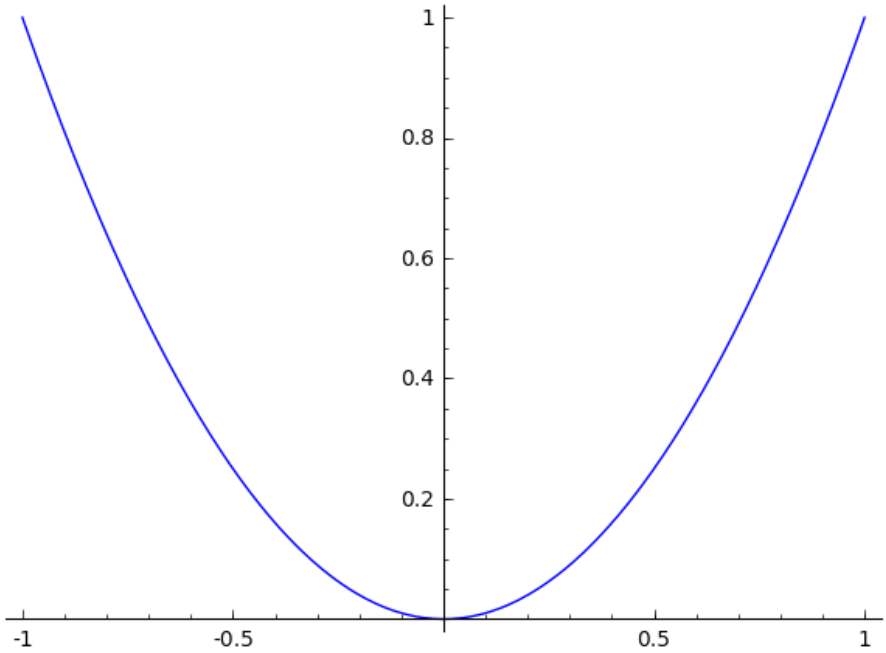
- **figsize**: 그래프의 크기를 조정한다.

```
plot(f(x),x,a,b, figsize='size number')
```

sage : **plot(x^2,x,-1,1, figsize='2')**



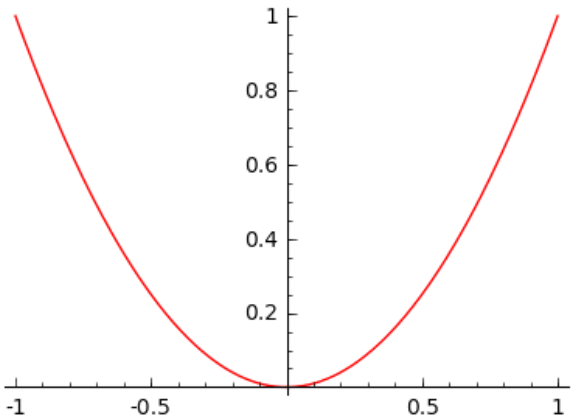
sage : **plot(x^2,x,-1,1, figsize='6')**



- **color**: 그래프의 색을 지정한다.

```
plot(f(x),x,a,b, color='color')
```

sage : **plot(x^2,x,-1,1, color='red')**



Sage에서는 다양한 색을 지원한다. 먼저 'red' 'black' 'purple'처럼 이미 지정되어 있는 색을 사용할 수도 있고 RGB값 (r,g,b) (여기서 각각의 값은 0과1 사이)를 입력하거나 html 색 지정 방법 '#aaff0b'을 사용할 수도 있다.

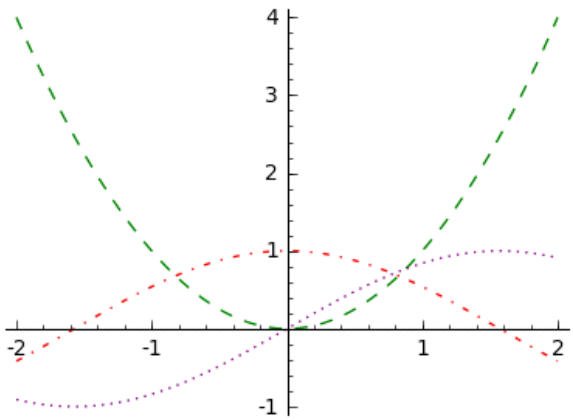
- **linestyle**: 그래프에 사용되는 선의 종류를 결정한다.

```
plot(f(x),x,a,b, linestyle='MATPLOTLIB line option')
```

선에 사용되는 옵션은 MATPLOTLIB에 사용되는 옵션을 그대로 차용하였는데 그 종류는 다음과 같다.

- 실선 : '-' 또는 'solid' (기본값이다.)
- 대쉬선 : '--' 또는 'dashed'
- 점선 : ':' 또는 'dotted'
- 대쉬점선 : '-.' 또는 'dash dot'

```
sage : p=plot(x^2,x,-2,2, color='green', linestyle='--', figsize='4')
sage : q=plot(sin(x),x,-2,2, color='purple', linestyle=':', figsize='4')
sage : r=plot(cos(x),x,-2,2, color='red', linestyle='-.', figsize='4')
sage : p+q+r
```



- **thickness**: 그래프에 사용되는 선의 굵기를 결정한다.

```
plot(f(x),x,a,b, thickness='thickness number')
```

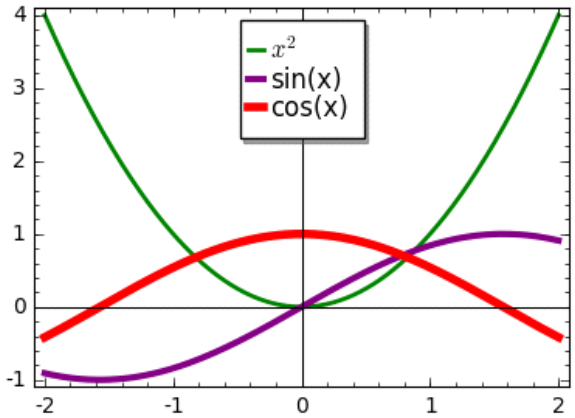
- **frame**: 그래프 주위에 박스로 프레임을 구성한다.

```
plot(f(x),x,a,b, frame=True)
```

- **legend_label**: 그래프의 이름을 붙인다.

```
plot(f(x),x,a,b, legend_label='legend')
```

```
sage : p=plot(x^2,x,-2,2, color='green', legend_label='$x^2$', thickness='2')
sage : q=plot(sin(x),x,-2,2, color='purple', legend_label='sin(x)', thickness='3')
sage : r=plot(cos(x),x,-2,2, color='red', legend_label='cos(x)', thickness='4')
sage : (p+q+r).show(figsize=4, frame=True)
```

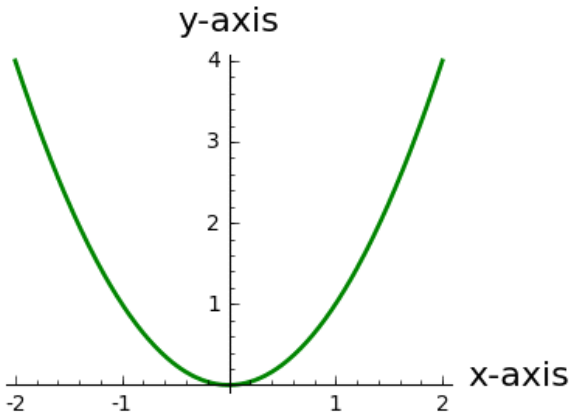


위의 명령을 살펴보면 먼저 x^2의 그래프를 굵기2의 녹색으로 그리는 객체를 **p**라고 지정하고 $\sin x$의 그래프를 굵기3의 보라색으로 그리는 객체를 **q**라고 지정했으며 $\cos x$의 그래프를 굵기4의 빨간색으로 그래는 객체를 **r**이라고 지정했다. 이후에 **(p+q+r)** 덧셈 연산자를 이용하여 3개의 그래프를 동시에 그리라는 명령을 했고 이 이후에 **.show()**라는 그리기 객체의 옵션을 지정할 수 있는 명령을 사용하여 **figsize=4, frame=True**의 옵션을 부여하였다.

● **axes_labels**: 그래프 축에 이름을 표시한다.

```
plot(f(x),x,a,b, axes_labels=['x-axis', 'y-axis'])
```

```
sage : p=plot(x^2,x,-2,2, color='green', thickness='2')
sage : p.show(figsize=4, axes_labels=['x-axis', 'y-axis'])
```



2.1.5 점을 표시하기

함수의 그래프를 그리다 보면 특정 위치의 점을 표시해야 하는 경우가 있다. 특정 위치에 점을 표시하는 방법은 명령어 **point**를 이용한다.

```
point((x,y), option)
```

point명령의 대표적인 옵션은 다음과 같다. (이 옵션들은 **plot**명령에서도 사용 가능하다).

- **pointsize**: 점의 크기를 정한다.

```
point((x,y), pointsize='size number')
```

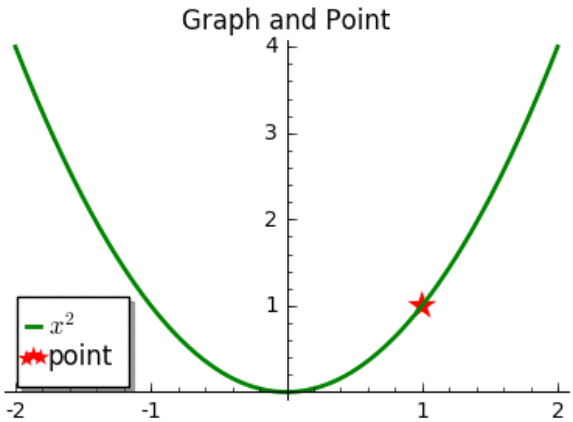
- **marker**: 점의 종류를 정한다. (아무것도 적지 않으면 일반 점을 표현한다.)

```
point((x,y), marker='maker')
```

종류는 다음과 같다.

- 가로선 : '-', 세로선 : '|'
- 원 : 'c', 오각형 : 'p', 사각형 : 's', 엑스 : 'x', 더하기 : '+', 별 : '*',
- 다이아몬드 : 'D', 얇은 다이아몬드 : 'd',

```
sage : p=plot(x^2,x,-2,2, color='green', legend_label='$x^2$', thickness='2')
sage : q=point((1,1), pointsize='200', color='red', marker='*', legend_label='point')
sage : (p+q).show(figsize=4, title='Graph and Point')
```



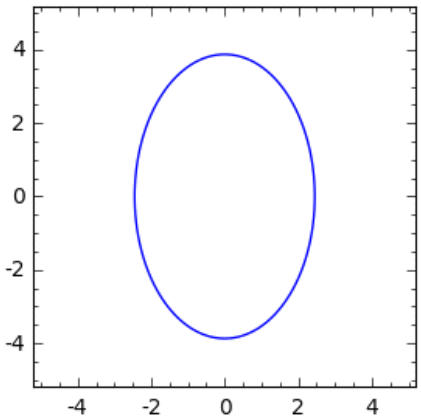
2.1.6 음함수의 그래프 그리기

음함수 형태로 되어 있는 식에 관해서도 그래프를 그릴 수 있다. 이때는 **implicit_plot** 명령을 사용한다.

```
implicit_plot(equation, (x,a,b), (y,c,d))
```

타원 $\frac{x^2}{2}+\frac{y^2}{5}=3$의 그래프를 그려보자.

```
sage : x,y=var('x,y')
sage : implicit_plot (x^2/2 + y^2/5==3 , (x,-5,5) , (y,-5,5)).show(figsize='4')
```

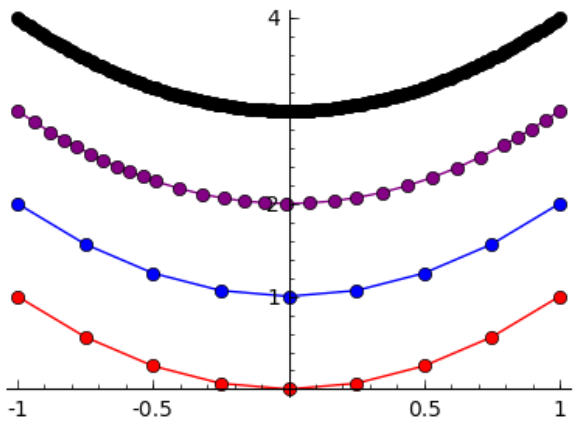


Sage에서 문자 **x**는 실수나 복소수를 의미하는 변수로 이미 지정되어 있다. 하지만 문자 **y**는 그렇지 않으므로 이 문자가 변수임을 지정하는 명령이 첫 번째 줄이다.

2.1.7 컴퓨터는 어떻게 함수의 그래프를 표현하는가?

컴퓨터는 함수의 그래프를 표현하기 위해 많은 점을 표시한 다음에 이를 직선으로 연결하는 방법을 주로 사용한다. 이를 확인해 보기 위해 다음의 명령을 살펴보자.

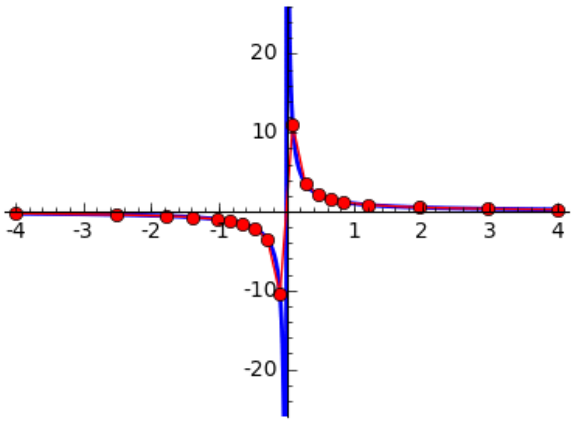
```
sage : p=plot(x^2,x,-1,1,plot_points=1, marker='o', color='red')
sage : q=plot(x^2+1,x,-1,1,plot_points=2, marker='o', color='blue')
sage : r=plot(x^2+2,x,-1,1,plot_points=10, marker='o', color='purple')
sage : s=plot(x^2+3,x,-1,1, marker='o', color='black')
sage : (p+q+r+s).show(figsize='4')
```



plot_points함수의 그래프를 표현하는데 이용하는 점의 단위를 표현하고 있다. 그리고 이 점들 사이를 직선으로 연결한다. 이 옵션을 입력하지 않으면 기본 값은 200으로 그래프 객체 **s**를 보면 많은 점이 붙어 마치 굵은 선처럼 되어 있는 것을

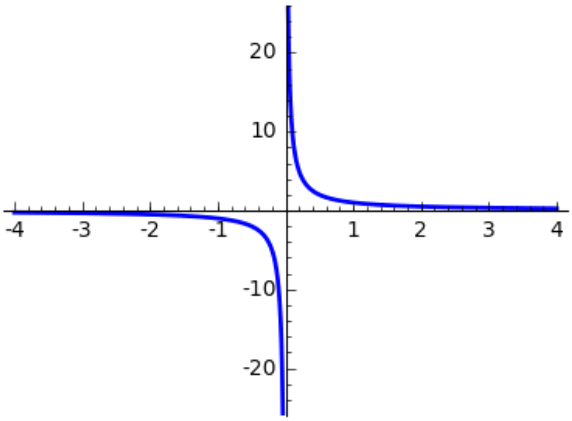
확인할 수 있다. 이러한 이유 때문에 함수의 그래프를 그리다 보면 종종 이런 일이 발생한다.

```
sage : p=plot(1/x,x,-4,4, ymin=-25, ymax=25, thickness='2', )
sage : q=plot(1/x,x,-4,4, ymin=-25, ymax=25, plot_points=3, color='red', marker='o')
sage : (p+q).show(figsize='4')
```



그래프 객체**p**는 $1/x$그래프를 표현한 것으로 y축에 점근선 같은 선이 있는 것을 확인할 수 있고 객체**q**는 같은 함수를 **plot_points**을 3로 설정하였다. 함수 위에 표시한 점의 차이로 인해 모양이 다른 것을 확인할 수 있고 객체**p**도 두 점 사이를 선으로 잇다 보니 y축에 선이 하나 생긴 것이다. 엄밀한 의미로 얘기하자면 두 그래프 모두 함수 $1/x$의 그래프라고 얘기 할 수 없을 것이다. 이를 해결하기 위해 다음의 옵션을 사용한 그래프를 확인하자.

```
sage : plot(1/x,x,-4,4, ymin=-25, ymax=25, thickness='2', exclude=[0], figsize='4')
```



옵션 **exclude**는 그래프의 정의역에서 제외하라는 명령으로 위 그래프는 점 $x=0$이 정의역에서 제외되었으므로 이를 지나는 직선은 그리지 않는다. 따라서 우리가 생각하는 모양을 얻을 수 있었다.

2.2 극한

2.2.1 리스트

극한과 연속에 대한 여러 가지 명령을 구현하기 위해 먼저 여러 개의 데이터를 저장하는 리스트라는 개념에 대해 알아보자. 1장에서 잠시 소개된 리스트는 다른 프로그래밍 언어에서 얘기하는 배열과 비슷한 개념이다. 여러 개의 데이터를 하나의 변수에 지정하는 것으로 예를 들어 **v=[1,2,3,4,5,6,7]**이라고 **v**를 정의하면 **v**는 총 7개의 데이터가 있는 리스트가 된다.

```
sage : v=[1,2,3,4,5,6,7]; v
 [1,2,3,4,5,6,7]
sage : parent(v)
 <type 'list'>
```

리스트 안의 데이터는 0을 시작으로 각각의 주소를 이용해 반환 가능하다. **v**안의 첫 번째와 세 번째 값을 반환하기 위해서는 다음과 같이 사용 한다.

```
sage : v[0]; v[2]
 1
 3
```

리스트 안의 특정 영역을 반환하려면 :(콜론) 기호를 사용한다. 예를 들면 다음과 같다.

```
sage : v[2:4]; v[2:]; v[:4]; v[:]
 [3, 4]
 [3, 4, 5, 6, 7]
 [1, 2, 3, 4]
 [1, 2, 3, 4, 5, 6, 7]
```

리스트 안의 주소는 0부터 시작하므로 **2:4**로 지정하면 2를 의미하는 3번째부터 4를 의미하는 5번째 바로 앞까지, 즉 세 번째와 네 번째 데이터를 반환하라는 의미이다. **2:**는 세 번째부터 마지막까지 **:4**는 처음부터 다섯 번째 전까지 (즉 네 번째까지) 그리고 **:**를 단독으로 사용하면 데이터의 전부를 의미한다. 리스트는 이중으로 사용 가능하다. 아래 예를 보자.

```
sage : w=[[1,2,3],[4,5,6]]; w; w[0][1]; w[1][2]
 [[1, 2, 3], [4, 5, 6]]
 2
 6
```

리스트 **w**는 대괄호 안에 또 다른 대괄호가 있는 이중으로 되어있는 구조이고 **w[0][1]**는 리스트 **w**안 첫 번째 리스트의 두 번째 데이터 **w[1][2]**는 리스트 **w**안 두 번째 리스트의 세번째 데이터를 의미하므로 각각 2와 6이 반환되었다. 이런 방식으로 행렬과 같은 구조를 만들 수 있지만 리스트는 행렬이 아니다. 왜냐하면 행렬에 사용되는 연산과 리스트에서 사용하는 연산은 다르게 정의되어 있기 때문이다. 참

고로 행렬은 만드는 방법은 명령 **matrix**을 이용 한다.

다음은 리스트의 값을 깔끔하게 출력해주는 **table**명령을 알아보자.

```
sage : A=[['a','b','c'],[1,2,3],[6,7,8]]
sage : table(A)
  a  b  c
  1  2  3
  6  7  8
```

table은 리스트의 구조를 행렬처럼 이해하기 쉽게 보여준다. 다음과 같은 옵션도 사용 가능하다.

```
sage : table(A, header_row=True, frame=True, align='center')
```

a	b	c
1	2	3
6	7	8

리스트를 만드는 방법 중에는 ..기호를 이용하는 방법도 있다. 다음의 예를 살펴보자.

```
sage : v=[3..12]; v
 [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

a..b기호를 이용하면 **a**부터 **b**까지의 정수를 쭉 나열하라는 의미이다. 비슷하게 명령어 **range**를 사용 가능하다.

```
range(stop)
range(start, stop ,step])
```

```
sage : range(10); range(3,11,2)
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
 [3, 5, 7, 9]
```

range(10) 명령은 0부터 자연수 9까지의 값을 반환하라는 의미이고
range(3,11,2) 명령은 3부터 10까지 2씩 증가하는 수를 반환하라는 의미이다.

2.2.2 table 사용과 기초 반복문

컴퓨터 프로그래밍에서 가장 중요한 반복문에 대해 알아보자. Sage에는 다양한 반복문 명령이 있고 이를 익히기 위해서는 많은 연습이 필요하지만 지금은 간단하게 **for**명령을 이용하여 우리가 원하는 함숫값을 **table**로 반환하는 방법에 대해 알아보자. 구문은 다음과 같다.

```
table( [f(x), g(x)] for x in [List] ])
```

위의 명령은 **[List]**안에 있는 데이터를 차례로 불러와 **[f(x), g(x)]**의 형태로 반환하라는 의미이다. **for**명령은 일반적으로 다음과 같이 사용하는데

```
for x in [List]:
    command
```

[List]안에 있는 데이터를 하나하나 불러오는데 **command**를 수행 하는 의미로 사용한다. 예를 들어

```
sage : v=range(3,11,2)
sage : a=0
sage : for x in v:
     :     a=a+x
sage : a
 24
```

처음 0이 저장되어 있는 **a**에 [3, 5, 7, 9]가 저장되어 있는 리스트에서 숫자를 하나씩 뽑아내어 더하라는 명령이다. 따라서 24가 반환되었다. **for**명령을 **table**과 함께 사용한 예는

```
sage : table([ (2*a, 3*a) for a in range(1,4)])
 2   3
 4   6
 6   9
```

range(1,4)로 만들어지는 값 1 2 3을 1부터 하나하나씩 반환한 값을 **a**라고 한 다음 **a**와 관련된 다음의 형태 **(2*a, 3*a)**로 **table**형태로 출력하라는 명령이다.

2.2.3 극한 개념의 수치적 접근

극한의 개념에서 한없이 접근함을 수치적으로 표현해 이해하곤 한다. 다음의 극한값을 찾는다고 생각 하자.

$$\lim_{x\to 0}\frac{\sin x}{x}$$

함수 $f(x)=\frac{\sin x}{x}$은 $x=0$에서 정의역이 아니다. 먼저 우극한 값을 유추하기 위해 함수 $f(x)$에 x값이 0.1부터 0.09 0.08 0.07순으로 접근하는 리스트를 만들려고 한다. 이것은 다음의 명령으로 만들 수 있다.

```
sage : rval=[a*0.01 for a in range(10,0,-1)]; x
 [0.100000000000000,
  0.0900000000000000,
  0.0800000000000000,
  0.0700000000000000,
  0.0600000000000000,
  0.0500000000000000,
  0.0400000000000000,
  0.0300000000000000,
  0.0200000000000000,
```

0.0100000000000000]

이 값에 대응하는 함수 $f(x)=\frac{\sin x}{x}$의 함숫값을 **table**명령을 이용해서 출력하면

sage : **f(x)=sin(x)/x**
sage : **table([(x, f(x)) for x in rval],frame=True, align='center')**

0.100000000000000	0.998334166468282
0.0900000000000000	0.998650546644567
0.0800000000000000	0.998933674614659
0.0700000000000000	0.999183533393325
0.0600000000000000	0.999400107990743
0.0500000000000000	0.999583385413567
0.0400000000000000	0.999733354665854
0.0300000000000000	0.999850006749855
0.0200000000000000	0.999933334666654
0.0100000000000000	0.999983333416666

비슷하게 좌극한 값을 유추할 수 있다. –0.1부터 –0.09 –0.08 –0.07순으로 접근하는 리스트를 만들고 이 값에 대한 함숫값을 찾으면 된다.

sage : **lval=[a*(-0.01) for a in range(10,0,-1)]**
sage : **table([(x, f(x)) for x in lval],frame=True, align='center')**

좌극한 우극한 추정 값을 동시에 출력하고 위에 관련 값을 표시해주는 명령을 수행해 보자

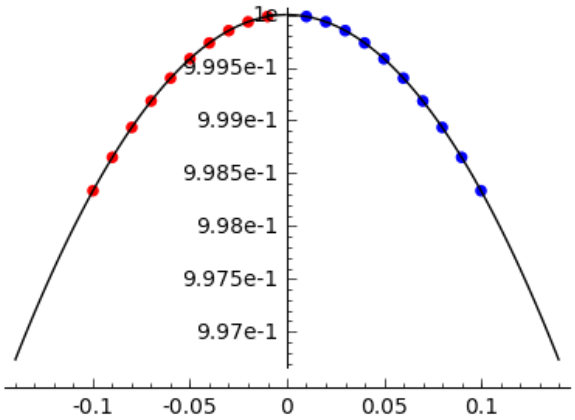
sage : **A=[['left x','f(x)','right x', 'f(x)']]**
sage : **val=[a*0.01 for a in range(10,0,-1)];**
sage : **B=[[-1*x,f(-1*x),x,f(x)] for x in val]**
sage : **table(A+B,header_row=True, frame=True, align='center')**

left x	f(x)	right x	f(x)
–0.100000000000000	0.998334166468282	0.100000000000000	0.998334166468282
–0.0900000000000000	0.998650546644567	0.0900000000000000	0.998650546644567
–0.0800000000000000	0.998933674614659	0.0800000000000000	0.998933674614659
–0.0700000000000000	0.999183533393325	0.0700000000000000	0.999183533393325
–0.0600000000000000	0.999400107990743	0.0600000000000000	0.999400107990743
–0.0500000000000000	0.999583385413567	0.0500000000000000	0.999583385413567
–0.0400000000000000	0.999733354665854	0.0400000000000000	0.999733354665854
–0.0300000000000000	0.999850006749855	0.0300000000000000	0.999850006749855
–0.0200000000000000	0.999933334666654	0.0200000000000000	0.999933334666654
–0.0100000000000000	0.999983333416666	0.0100000000000000	0.999983333416666

이 표는 x가 0으로 접근할 때 $f(x)$의 값이 1에 가까워 짐을 보여준다.

리스트 **A**는 표에서 위에 제목에 해당하는 부분이고 데이터에 해당되는 부분은 **B**이다. 그리고 이를 **+**연산을 이용하여 함께 출력하였다. 이와 같이 리스트에서 덧셈 연산은 두 표를 연결하라는 의미를 가지고 있다. 위 표에 있는 값과 함수를 그래프로 표현해 보자.

```
sage : f(x)=sin(x)/x
sage : p=plot(f(x),x,-0.14,0.14, exclude=[0], color='black')
sage : q=point([[c,f(c)] for c in val], pointsize='30')
sage : r=point([[-1*c,f(-1*c)] for c in val], pointsize='30', color='red')
sage : (p+q+r).show(figsize='4')
```



그래프 객체 **p**는 검은색으로 표시된 함수 $f(x)$의 그래프이고 우극한 추정 값들은 파란색 점으로 좌극한 추정 값들은 빨간색으로 표시했다.

2.2.4 극한 계산

Sage에서 극한 계산은 수학적으로 증명된 극한 법칙을 이용하여 잘 계산한다. 함수 $f(x)$의 x가 a로 한없이 가까이 갈 때의 극한값 $\lim_{x\to a} f(x)$를 구하는 명령은 다음과 같다.

```
limit(f(x),x=a)
```

우극한과 좌극한을 구하는 명령은 각각 다음과 같다.

```
limit(f(x),x=a,dir='+')
limit(f(x),x=a,dir='-')
```

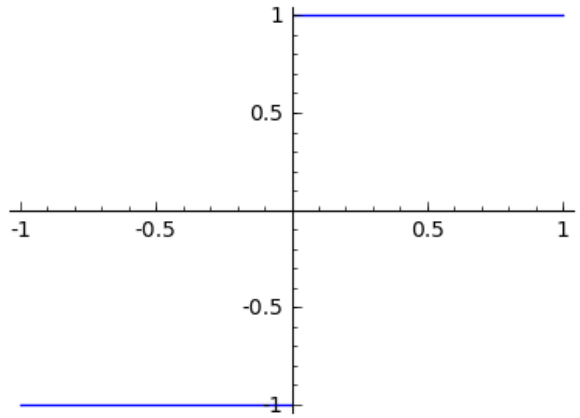
여기서 옵션 **dir**의 값 **'+'** 대신에 **'plus'** 혹은 **'right'**를 사용 가능하고 **'-'** 대신에 **'minus'** 혹은 **'left'**를 사용 가능하다. 몇 가지 극한 계산의 예를 들어 보자.

```
sage : limit(sin(x)/x,x=0)
  1
```

$\lim_{x\to 0}\frac{\sin x}{x}=1$ 임을 잘 계산하고 있다. 다음은 불연속점에 대한 계산이다.

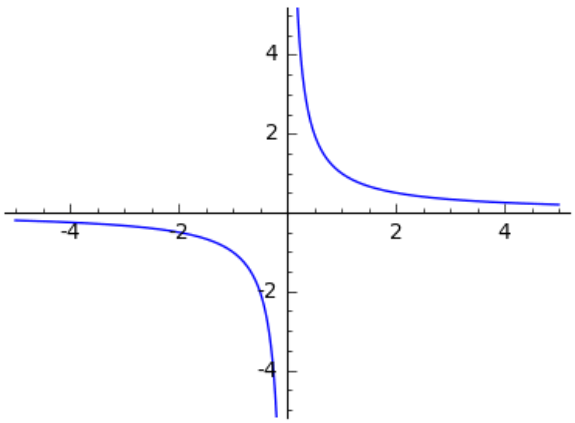
```
sage : limit(abs(x)/x,x=0,dir='+')
```

```
  1
sage : limit(abs(x)/x,x=0,dir='-')
  -1
sage : limit(abs(x)/x,x=0)
  und
sage : plot(abs(x)/x,x,-1,1,exclude=[0],figsize='4')
```



함수$f(x)=\frac{|x|}{x}$는 $x=0$에서 정의되지 않으며 좌극한과 우극한이 다름을 알고 있다. 이를 Sage에서도 잘 계산하며 $x=0$의 극한값 반환 명령어에는 정의 할 수 없다(Undefined)는 **und**표현을 사용하고 있다. 또 다른 예를 살펴보자.

```
sage : limit(1/x,x=0,dir='+')
  +Infinity
sage : limit(1/x,x=0,dir='+')
  -Infinity
sage : limit(1/x,x=0)
  Infinity
sage : plot(1/x,x,-5,5,exclude=[0],figsize='4',ymin=-5, ymax=5)
```



함수 $f(x)=\frac{1}{x}$는 $x=0$에서 정의되지 않고 극한 값은 발산함을 알고 있다. 우극한은 양의 방향으로 발산하고 좌극한은 음의 방향으로 발산한다. 이를 Sage에서는

+Infinity와 -Infinity로 표현하고 있으며 방향을 정의할 수 없는 발산에 대해서는 Infinity로 표현하여 극한이 수렴하지 않음을 표시하고 있다.

2.2.5 무한대와 관련된 극한 계산

함수 $f(x)$의 $x \to \infty$로의 극한 계산은

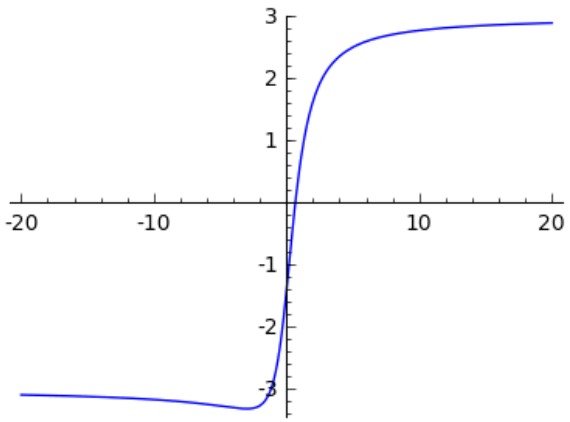
```
limit(f(x),x=infinity)
```

를 이용하고 $x \to -\infty$의 극한 계산은 명령

```
limit(f(x),x=-infinity)
```

을 이용한다. 먼저 함수 $f(x)=\frac{3x-2}{\sqrt{x^2+2}}$의 무한대 극한을 계산 하고 함수의 그래프를 그려본 명령이다.

```
sage : f(x) = (3*x-2)/sqrt(x^2+2)
sage : limit(f(x),x=Infinity)
  3
sage : limit(f(x),x=-Infinity)
  -3
sage : plot(f(x),x,-20,20,figsize='4')
```



1과 -1사이에서 진동하는 코사인 함수의 극한을 구하는 명령을 해보자.

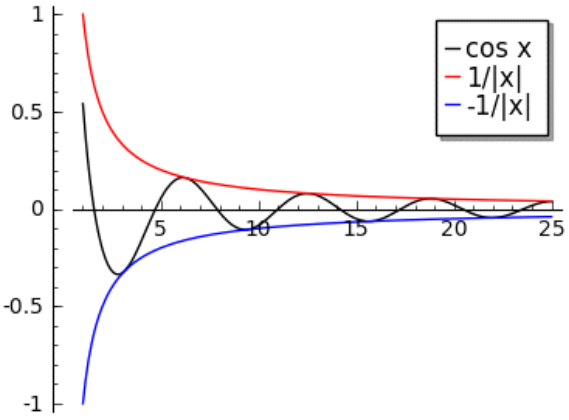
```
sage : limit(cos(x),x=Infinity)
  ind
```

수렴 하지 않는 극한 $\lim_{x\to\infty}\cos x$을 계산하는 명령에 대한 결과는 극한 값이 정의 되지 않지만 함숫값이 발산하지는 않는다는 ind(indefinite but bounded) 결과를 보여준다.

```
sage : limit(cos(x)/x, x=Infinity)
  0
```

$\lim_{x\to\infty}\frac{\cos x}{x}=0$에 대한 근거로 주로 조임 정리를 사용한다. $\frac{\cos x}{x}$함수가 $\frac{1}{|x|}$와$\frac{-1}{|x|}$ 사이에 존재하기 때문인데 이 관계를 그래프를 통해 확인해 보자.

```
sage : p=plot(cos(x)/x,x,1,25, color='black', legend_label='cos x')
sage : q=plot(1/abs(x),x,1,25, color='red', legend_label='1/|x|')
sage : r=plot(-1/abs(x),x,1,25, color='blue', legend_label='-1/|x|')
sage : (p+q+r).show(figsize='4')
```



2.3 연속

2.3.1 연속

정의역과 공역이 실수인 어떤 함수 f가 $x=a$에서 연속인 것을 $\lim_{x\to a}f(x)=f(a)$로 정의 한다. 기하학적으로는 함수의 그래프가 점$(a,f(a))$을 중심으로 끊어지지 않았다는 것을 의미한다. 함수의 연속임을 밝히기 위해 Sage에서는 등식의 참 거짓을 밝히는 함수 **bool**을 이용할 수 있다.

```
bool(argument)
```

예를 들어 함수 $f(x)=\begin{cases}\frac{x^2-1}{x-1} & \text{if } x\neq 1\\ 6 & \text{if } x=1\end{cases}$ 에 대해 다음의 명령을 사용해 보자.

```
sage : def f(x):
     :     if x<>1:
     :         return (x^2-1)/(x-1)
     :     else:
     :         return 6
sage : f(1)
  6
sage : limit(f(x),x=1)
```

```
2
```

함숫값과 극한값이 다르므로 이 함수는 $x=1$에서 연속이 아님을 알 수 있다. 또한 다음과 같이 확인해 볼 수도 있다.

```
sage : bool(limit(f(x),x=1)==f(1))
  False
```

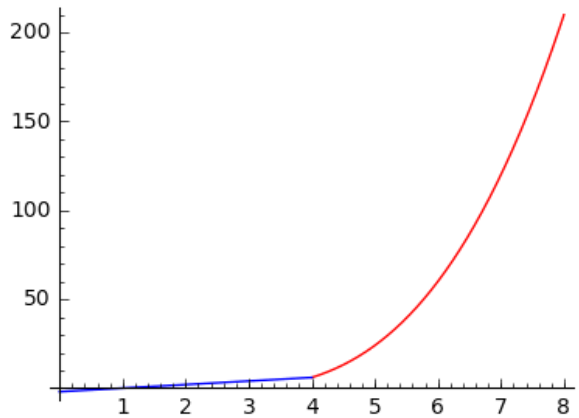
$x=1$에서 함숫값과 극한값이 같은지 아닌지를 **bool**명령으로 확인해 봤을 때 같지 않다는 결과 **False**결과를 반환했다. 따라서 함수는 $x=1$에서 연속이 아니다.

주어진 함수 $f(x)=\begin{cases} x^3-6x^2+kx-6 & x<4 \\ 2x-13+k & x\geq 4 \end{cases}$가 연속함수가 되게 k를 지정하는 문제를 생각해 보자. 먼저 함수 $x^3-6x+kx-6$와 $2x-13+k$는 어떠한 실수 k에 대해서도 다항함수이므로 연속함수이다. 따라서 우리는 $x=4$일 때 위 함수가 연속함수가 되게 하는 k를 찾으면 된다.

```
sage : var('k')
sage : f1(x)=x^3-6*x^2+k*x-6
sage : f2(x)=2*x-13+k
sage : solve(f1(4)==f2(4),k)
  [k == 11]
```

두 함수를 정의하고 $x=4$를 넣은 함숫값이 서로 같게 하는 k값은 11임을 알 수 있다. 이 함수의 그래프를 그려 보면

```
sage : p=plot(f1(x).substitute(k=11), x,4,8, color='red')
sage : q=plot(f2(x).substitute(k=11), x,0,4, color='blue')
sage : (p+q).show(figsize='4')
```



참고로 함수를 구간별로 정의했을 때 **def f(x):** 명령을 이용했다면 이 함수를 **plot(f(x))**을 이용하여 그릴 수는 없다.

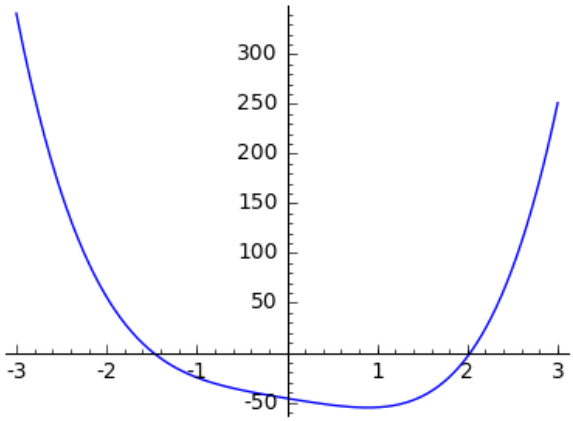
2.3.2 중간값 정리

중간값 정리는 함수 $f(x)$가 구간 $[a,b]$에서 연속이라면 $f(a)$와 $f(b)$사이에 있는 값 d에 대해 $f(c)=d$를 만족하는 c가 $[a,b]$구간 안에 존재한다는 정리이다. 이 정리는 연속 함수에 대한 중요한 정리 중 하나로 어떤 방정식의 근의 유무에 대해서 알려주곤 한다. 방정식 $4x^4+2x^2-15x+4=50$이 구간 $[-3,3]$에서 적어도 하나의 근을 가제가 됨을 살펴보자.

```
sage : f(x) = 4*x^4 + 2*x^2 -15*x + 4 - 50
sage : f(-3); f(3)
 341
 251
```

이를 살펴보기 위해 함수 $f(x)=4x^4+2x^2-15x+4-50$를 설정하고 $f(c)=0$이 되는 지점을 찾기 위해 양 끝점에 대한 함숫값을 조사하였지만 두 값이 모두 양수로 중간값 정리를 만족하지 못한다. 이에 이 함수의 그래프를 그려보겠다.

```
sage : plot(f(x),-3,3, figsize='4')
```



그래프를 통해 우리는 2개 근이 있음을 유추 가능하다. 이를 명확하게 하는 다음의 함숫값을 이용하자

```
sage : f(-2); f(-1)
 56
 -25
sage : f(-3); f(3)
 -55
 251
```

함수 $f(x)=4x^4+2x^2-15x+4-50$는 다항함수이므로 모든 점에서 연속이고 $f(-2)=56$이고 $f(-1)=-25$이므로 중간값 정리에 의해 $[-2,-1]$구간에서 적어도 하나의 해가 있고 마찬가지로 $f(1)=-55$ $f(3)=251$이므로 중간값 정리에 의해 구간 $[1,3]$에 적어도 하나의 해가 있다. 그래프의 모양에 따른 중간값 정리의 의미를 확인할 수 있다.

2.3.3 이분법 (Bisection Method)

5차 이상의 다항식을 비롯한 우리가 접하는 수많은 방정식은 대수적으로 그 해를 찾기 어렵다. 이 때 우리는 그 해를 근사적으로 계산하게 되는데 중간값 정리를 이용한 간단한 알고리즘인 이분법을 적용하여 해를 찾아보도록 하겠다.

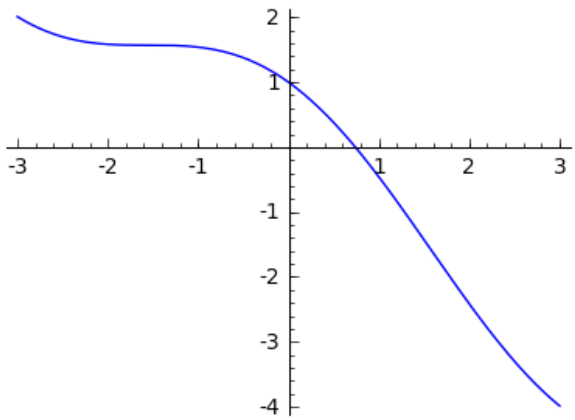
- **알고리즘(이분법)** (연속함수f, 왼쪽끝점a, 오른쪽끝점b, 오차범위ϵ)
  1. $c=(a+b)/2$로 지정한다.
  2. 만약 $|b-c| \leq \epsilon$이면 해를 c로 지정하고 알고리즘을 마친다.
  3. 만약 $f(b)$와 $f(c)$의 곱이 음수면 $a=c$로 양수면 $b=c$로 지정한다.
  4. 1번으로 되돌아간다.

이분법을 이용하여 방정식 $\cos x = x$을 풀어보자 먼저 이 방정식을 대수적으로 푸는 명령을 실행하면

```
sage : solve(cos(x)==x,x)
 [x == cos(x)]
```

수식을 그대로 반환하면서 해를 찾지 못한다. $f(x)=\cos x - x$로 지정하고 함수의 그래프를 그려보면

```
sage : f(x)=cos(x)-x
sage : plot(f(x),x,-3,3,figsize='4')
```



$x=0$과 $x=1$사이에 근이 하나 있음을 확인할 수 있다. 이 해를 이분법으로 찾아보자

```
sage : ep=0.0000001; a=0; b=1.0;
sage : iter=0
sage : while True:
     :     iter=iter+1
     :     c=(a+b)/2
     :     if abs(b-c)<ep: break
```

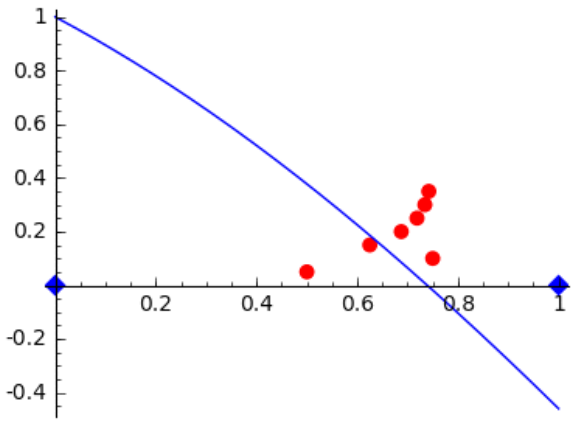
```
        :     if f(b)*f(c)<=0: a=c
        :     else: b=c
sage : c; f(c); iter
 0.739085137844086
 -7.74702468842037e-9
 24
```

먼저 결과를 살펴보면 근사해는 0.7390851378440860이 나왔고 이 값을 $f(x)=\cos x-x$에 대입한 결과는 –7.74702468842037e-9가 나왔다. 총 반복 횟수는 24이다. 명령어에 대해 설명하면 먼저 a, b, ϵ을 변수 **a b ep**로 지정하고 반복 횟수를 확인하기 위해 변수 **iter**를 도입하였다. 알고리즘을 반복하기 위한 반복문은 **while**을 이용 하였는데 이 명령어의 사용법은

```
while condition:
    command
```

와 같고 **condition**조건을 만족하면 계속 반복하라는 의미이다. 우리는 **while True**라고 지정하였는데 **True**에 의미가 없으므로 반복 내의 **break**를 dnlgsk 요건이 만족될 때까지 계속 반복하게 된다. 반복문 안의 명령은 이분법 알고리즘 내용과 같다. 단 반복문이 시작될 때마다 **iter=iter+1**명령을 통해 총 반복 횟수를 알 수 있게 구성하였다. 이분법이 반복될 때마다 해를 찾아가는 방법을 알아보기 위해 다음의 명령을 살펴보자.

```
sage : n=7; a=0; b=1.0; Cvals=[]; Yvals=[];
sage : for i in [1..n]:
        :     c=(a+b)/2
        :     Cvals.append(c)
        :     if abs(b-c)<ep: break
        :     if f(b)*f(c)<=0: a=c
        :     else: b=c
sage : Yvals=[ i*0.05 for i in [1..n]]
sage : Points=zip(Cvals,Yvals)
sage : f(x)=cos(x)-x
sage : p=plot(f(x),x,0,1)
sage : q=point(Points, pointsize='50', color='red')
sage : r=point([(0,0),(1,0)], pointsize='50', marker='D')
sage : (p+q+r).show(figsize='4')
```



변수 **n**은 총 반복 횟수를 의미하고 **Cvals**은 반복하면서 생기는 **c**값을 차례대로 저장하는 리스트이다. **Cvals=[]**로 지정하면 비어있는 리스트를 만들게 된다. **Yvals**은 그림에서 빨간색 점을 찍기 위해 **c**값이 업데이트 될수록 점의 위치를 위로 이동시키는 역할을 한다. **append()**명령은

```
List.append(x)
```

리스트의 맨 마지막에 **x**를 삽입하라는 명령이다. 또한 명령어 **zip**은

```
zip(List1, List2)
```

List1과 **List2**의 값을 차례로 하나씩 가져와 순서쌍(turple) 형태로 저장하는 명령어이다. 따라서 순서쌍 리스트인 **Points**의 값을 출력해보면

sage : **table(Points)**

0.500000000000000	0.0500000000000000
0.750000000000000	0.100000000000000
0.625000000000000	0.150000000000000
0.687500000000000	0.200000000000000
0.718750000000000	0.250000000000000
0.734375000000000	0.300000000000000
0.742187500000000	0.350000000000000

와 같이 구성된다. 함수의 그래프를 그린 객체를 **p**라 하고 시작점 **a**와 **b**를 다이아몬드 점으로 그린 객체를 **r**이라 하였고 순서쌍 리스트 **Points**의 값을 빨간색 점으로 그린 객체를 **q**라 하여 이를 표현하였다.

제 3장

미분과 그 응용

3.1 미분

3.1.1 미분 계수와 접선

함수 $f(x)$의 $x=a$에서의 미분계수는 극한 $\lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$로 정의하고 존재하면 $f'(a)$라 표시한다. 미분계수의 기하학적인 의미는 함수 $f(x)$의 $x=a$에서 접선의 기울기를 의미한다. 먼저 함수 $f(x)=x^3+5x^2-3x+2$의 $x=5$에서의 미분계수를 정의 $\lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$를 이용하여 h가 점점 작아지는 숫자를 넣어 수치적으로 알아보자.

```
sage : f(x)=x^3+5*x^2-3*x+2; a=5;
sage : A=[ ['h','limit value'] ]
sage : val=[ N(1/(10^i)) for i in [1..6] ]
sage : B=[ [h, N((f(a+h)-f(a))/h)] for h in val]
sage : table(A+B)
```

h	limit value
0.100000000000000	124.010000000000
0.0100000000000000	122.200099999998
0.00100000000000000	122.020001000038
0.000100000000000000	122.002000009616
0.0000100000000000000	122.000199993977
$1.00000000000000\times 10^{-6}$	122.000020013502

h의 값을 $1/10$씩 점점 줄여 나가니 극한 값이 122에 가까워지는 것으로 추정이 된다. 실제 극한을 계산해 보면

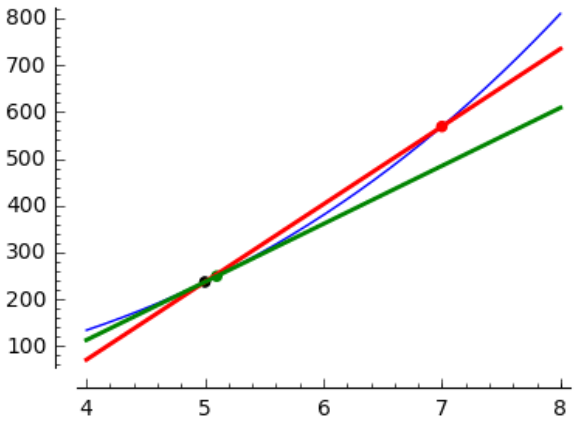
```
sage : var('h')
sage : limit( (f(a+h)-f(a))/h, h=0)
 122
```

미분계수는 122가 나옴을 알 수 있다. 또한 실제 손으로 계산하는 것과 비슷하게 $\frac{f(a+h)-f(a)}{h}$의 계산이 가능하다면 이를 먼저 계산한 뒤에 h를 0에 근접시키는 방법을 사용할 수도 있다. 이 방법은 다음과 같다.

```
sage : var('h')
sage : ((f(a+h)-f(a))/h).simplify_full()
 h^2 + 20*h + 122
```

$\frac{f(a+h)-f(a)}{h}$값을 계산하여 정리하였더니 $h^2+20h+122$가 나왔고 이 값에 h를 0으로 근접시키면 122가 나옴을 알 수 있다. 위에서 제시한 과정을 그래프와 함께 살펴보자. $h=2.0$일 때의 두 점 $(5,f(5))$와 $(7,f(7))$을 지나는 직선을 나타내는 함수 $y=\frac{f(7)-f(5)}{2}(x-5)+f(5)$와 $h=0.1$일때의 두 점 $(5,f(5))$와 $(5.1,f(5.1))$을 지나는 직선을 나타내는 함수 $y=\frac{f(5.1)-f(5)}{0.1}(x-5)+f(5)$을 그려 보자.

```
sage : f(x)=x^3+5*x^2-3*x+2;
sage : g(x)=(f(7)-f(5))/2.0*(x-7)+f(7);
sage : h(x)=(f(5.1)-f(5))/0.1*(x-5)+f(5);
sage : p=plot(f(x), x,4,8)
sage : q=plot(g(x), x,4,8, color='red', thickness='2')
sage : r=plot(h(x), x,4,8, color='green', thickness='2')
sage : s=point([(5,f(5))],pointsize='40', color='black')
sage : t=point([(7,f(7))],pointsize='40', color='red')
sage : u=point([(5.1,f(5.1))],pointsize='40', color='green')
sage : (p+q+v+s+t+r).show(figsize='4')
```



검은색 점은 $(5,f(5))$이고 녹색 점은 $(5.1,f(5.1))$ 빨간색 점은 $(7,f(7))$이고 각 선은 두 점을 연결한 직선이다. 빨간색 직선에 비해 녹색 직선이 검은색 점 위의 접선과 더 유사함을 확인할 수 있다.

3.1.2 도함수

Sage에서 정의한 함수 **f(x)**에 대한 도함수를 구하는 명령은 다음의 2가지 방법이 있다.

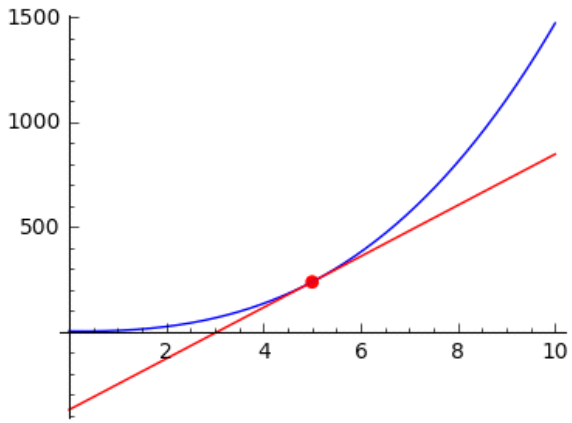
```
diff(f(x))
f(x).derivative()
```

또한 $x=a$에서의 미분계수는 도함수의 $x=a$에서 값이므로 함수의 $x=a$에서 미분계수를 다음과 같이 구할 수 있다.

```
diff(f(x)).substitute(x=a)
```

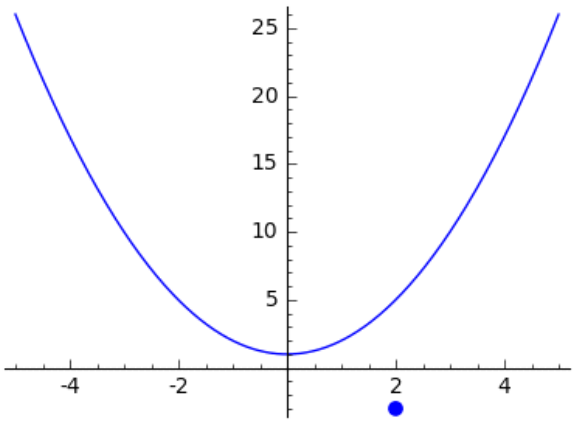
앞의 절에서 이용한 함수 $f(x)=x^3+5x^2-3x+2$의 $x=5$에서 미분계수와 접선을 그려 보자 접선의 방정식은 $y=f'(a)(x-a)+f(a)$임을 잘 알고 있다.

```
sage : f(x)=x^3+5*x^2-3*x+2;
sage : slop=diff(f(x)).substitute(x=5)
sage : slop
 122
sage : p=plot(f(x), x,0,10)
sage : q=plot(slop*(x-5)+f(5), x,0,10, color='red')
sage : r=point([(5,f(5))],pointsize='40', color='red')
sage : (p+q+r).show(figsize='4')
```



함수 $f(x)=x^2+1$의 접선 중에 점 $P(2,-3)$을 지나는 선을 찾는 문제를 Sage에 적용해 보자. 먼저 함수와 점을 그래프로 표현해 보자.

```
sage : p=plot(x^2+1,x,-5,5)
sage : q=point((2,-3),pointsize='50')
sage : (p+q).show(figsize='4')
```



위와 같은 문제를 직접 푼다면 함수 위에 있는 어떤 점 $(a, f(a))$와 점 $P(2,-3)$과의 직선의 기울기와 점 $(a, f(a))$에서의 미분계수가 같게 되는 값 a를 찾으려고 할 것이다. 이 사실을 이용하자.

```
sage : var('a')
sage : f(x)=x^2+1
sage : m=(f(a)-(-3))/(a-2)
sage : sol=solve(m==diff(f(x)).substitute(x=a),a); sol
 [a == -2*sqrt(2) + 2, a == 2*sqrt(2) + 2]
```

임의의 값 a는 변수로 지정하고 함수 $f(x)$를 정의 하였다. 두 점 $P(2,-3)$와 $(a, f(a))$의 기울기를 **m**으로 지정하고 이 값과 $x=a$일때의 $f(x)$의 미분계수가 서로 같게 되는 a를 찾은 값을 변수 **sol**로 지정했더니 2개의 값이 리스트로 출력되었다. 접선의 방정식은

$$y = f'(a)(x-a)+f(a)$$

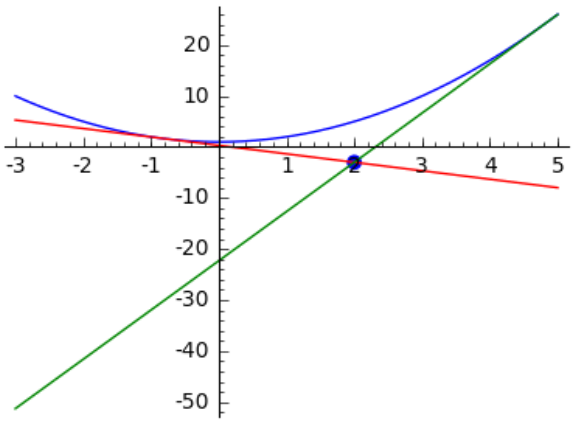
임을 알고 있으므로 이를 이용하는 방법은 다음과 같다.

```
sage : a1=sol[0].rhs(); a2=sol[1].rhs()
sage : y1=diff(f(x),x).substitute(x=a1)*(x-a1)+f(a1);
sage : y2=diff(f(x),x).substitute(x=a2)*(x-a2)+f(a2);
sage : y1.simplify_full(); y2.simplify_full()
 -4*x*(sqrt(2) - 1) + 8*sqrt(2) - 11
 4*x*(sqrt(2) + 1) - 8*sqrt(2) - 11
```

먼저 리스트 **sol**에 있는 각 항목의 우변이 기울기 값이므로 이 2개의 값을 **a1**, **a2**라고 지정하고 점선의 방정식 식을 이용하여 두 직선 **y1**, **y2**를 지정한 다음 **simplify_full()**명령을 이용해서 이 식을 최대한 간단히 표현하도록 했다.

마지막으로 이 모든 항목을 그래프로 표현하면

```
sage : p=plot(x^2+1,x,-3,5)
sage : q=point((2,-3),pointsize='50')
sage : r=plot(y1,x,-3,5,color='red')
sage : t=plot(y2,x,-3,5,color='green')
sage : (p+q+r+t).show(figsize='4')
```



3.2 다양한 미분 기법

3.2.1 고계 도함수

함수 $f(x)$를 n번 미분한 n계 도함수를 찾는 명령은 다음과 같다.

```
diff(f(x), x, n)
```

예를 들어 $f(x)=\sin x$의 300번 미분한 함수는 자기 자신이다.

```
sage : diff(sin(x),x,300)
 sin(x)
```

또한 도함수를 리스트로 표현할 수도 있다. 예를 들어 함수 $x\sin x$를 1번부터 5번 미분한 함수를 나열해 보자.

```
sage : [diff(x*sin(x),x,i) for i in [1..5]]
 [x*cos(x) + sin(x),
 -x*sin(x) + 2*cos(x),
 -x*cos(x) - 3*sin(x),
 x*sin(x) - 4*cos(x),
 x*cos(x) + 5*sin(x)]
```

3.2.3 음함수의 미분법

음함수 $f(x,y)=c$가 주어졌을 때 $\frac{dy}{dx}$를 구하기 위해서 우리는 변수 y를 x에 관한 함수로 취급하고 계산한다. Sage에서도 이와 유사하게 이용한다. 먼저 주어진 음함수 $x^3+y^3=6xy$에 대한 $\frac{dy}{dx}$를 구해보는 예를 들겠다. 먼저 음함수 미분을 하기 위해 음함수를 정의하는 다음의 방법에 대해 알아보자.

```
sage : var('x,y')
sage : y=function('y')(x)
sage : eq=x^3+y^3==6*x*y
sage : eq
 x^3 + y(x)^3 == 6*x*y(x)
```

먼저 음함수를 Sage에서 인식시키기 위해서는 문자 **y**를 사용해야 하므로 **y**를 변수로 선언해 준다. 여기서 우리는 음함수의 미분을 위해 **y**를 **x**에 관한 함수로 지정하는 명령인 **y=function('y')(x)**를 이용했고 음함수를 **eq**라는 이름의 식으로 정의하였다. 결과를 보면 **eq**는 음함수의 형태로 반환되는데 여기서 **y**가 **y(x)**의 형태임이 확인 가능하다. 이식을 미분하면 다음과 같이 된다.

```
sage : diff(eq,x)
 3*y(x)^2*D[0](y)(x) + 3*x^2 == 6*x*D[0](y)(x) + 6*y(x)
```

여기서 **D[0](y)(x)**이 의미하는 것은 우리가 손으로 계산할 때 표현하는 y' $y'(x)$ $\frac{dy}{dx}$를 의미한다. 이 형태로는 $\frac{dy}{dx}$형태로 알아보기 쉽지 않으므로 다음과 같지 정리해 준다.

```
sage : solve(diff(eq,x),diff(y))
 [D[0](y)(x) == -(x^2 - 2*y(x))/(y(x)^2 - 2*x)]
```

$\frac{dy}{dx}$형태로 정리하기 위해 **solve**명령을 사용했으며 **D[0](y)(x)**을 의미하는 **diff(y)**를 사용하였다.

음함수의 미분 기능을 응용하기 위해 위에서 살펴보았던 음함수에서 접선의 기울기가 0인 점을 찾는 방법에 대해 알아보자 이를 위해 $\frac{dy}{dx}=0$과 음함수 $x^3+y^3=6xy$를 만족하는 점 (x,y)를 찾아야 한다. 명령어를 살펴보자.

```
sage : var('z')
sage : dydx=solve(diff(eq,x),diff(y))
sage : con1=(dydx[0].rhs()).substitute(y(x)==z)
sage : eqz=eq.substitute(y(x)==z)
sage : sys1; eqz
 -(x^2 - 2*z)/(z^2 - 2*x)
 x^3 + z^3 == 6*x*z
```

변수 **z**는 **solve**명령을 사용하기 위해 이미 함수로 지정된 변수 **y**대신 사용하고자 정의하였다. 왜냐하면 **solve**명령은 변수 사이의 해를 찾아주는 명령인데 문자 **y**는 변수가 아닌 **x**에 관한 함수로 바뀌었기 때문이다. 우리가 위에서 찾은 dy/dx를 변수 **dydx**에 지정하고 이 식에 **y(x)**의 함수를 변수 **z**로 변경한 식을 **con1**이라고 지

정하였다. 그리고 위에서 정의한 음함수를 **y(x)**대신에 **z**로 변환하여 이를 **eqz**이라고 지정 하였다.

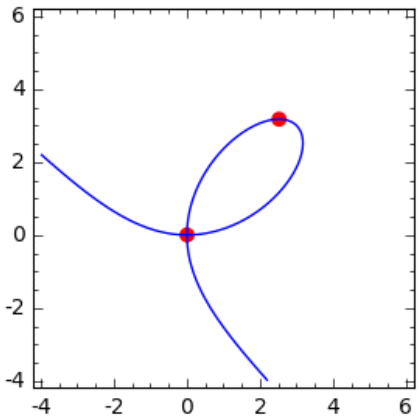
```
sage : sols=solve([con1==0,eqz],x,z)
sage : sols
 [[x == 2.519842139881605, z == 3.174802110817942],
  [x ==(-1.259921049894873 + 2.182247271943443*I),
   z == (-1.5874010519682 -2.749459273997206*I)],
  [x == (-1.259921049894873 - 2.182247271943443*I),
   z == (-1.5874010519682 + 2.749459273997205*I)],
  [x == 0, z == 0]]
```

총 4개의 해가 나왔는데 이 중에는 허근도 포함되어 있다. 이 해 중에서 실수해만 따로 뽑아 보자.

```
sage : real_sols=[ [a,b] for [a,b] in sols if imag(a.rhs())==0 and imag(b.rhs())==0]
sage : real_sols
 [[x == 2.519842139881605, z == 3.174802110817942], [x == 0, z == 0]]
```

위의 명령은 어떤 값에서 허수 부분을 찾는 **imag()**명령을 사용했으며 이 허수 부분이 0과 동일한 값만 출력하라는 명령을 이용해 2개의 실수해를 찾아냈다. 마지막으로 이를 그래프위에 나타내 보자.

```
sage : p=implicit_plot(eqz,(x,-4,6),(z,-4,6))
sage : q=point(Points,color='red',pointsize='50')
sage : (p+q).show(figsize='4')
```

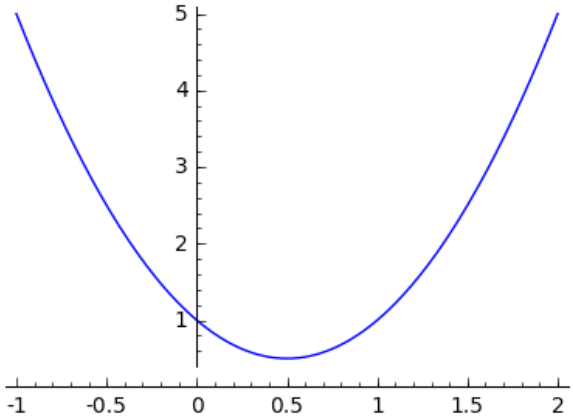


3.2.4 역함수와 미분

$g(x)$가 함수 $f(x)$의 역함수라는 것의 정의는 $f(g(x))=g(f(x))=x$를 만족할 때 이다. 그리고 $g(x)=f^{-1}(x)$라고 표현 한다. 함수 $f(x)$가 가역함수인 조건은 전단사 함

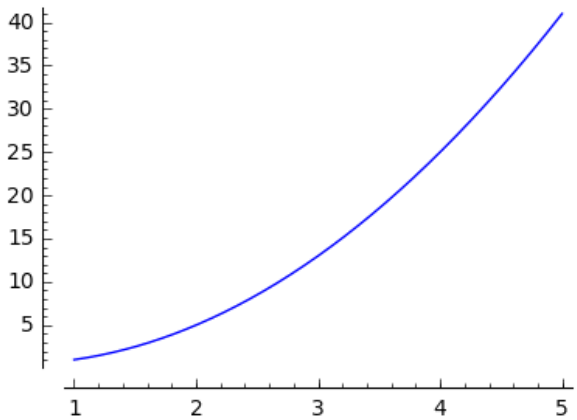
수 임을 잘 알고 있다. 또한 실수에서 정의된 연속 함수에서 전단사 함수는 함수 $f(x)$가 순증가 함수 또는 순 감소함수임도 알고 있다. 먼저 Sage를 이용해 역함수를 구하는 방법에 대해 알아보자. 만약 함수를 $f(x)=2x^2-2x+1$로 정한다면 이 함수는 2차 함수 이므로 순증가함수 혹은 순감소함수가 아니며 따라서 전단사 함수가 아니다. 그래프를 그려 확인 하자.

```
sage : f(x)=2*x^2-2*x+1
sage : plot(f(x),x,-1,2,figsize='4')
```



여기서 전단사 함수를 만들기 위해 정의역을 제한하자. 위의 그래프에서 순증가하는 구간을 생각하여 정의역을 $[1,\infty)$로 두고 그래프를 그려보자.

```
sage : plot(f(x),x,1,5,figsize='4')
```



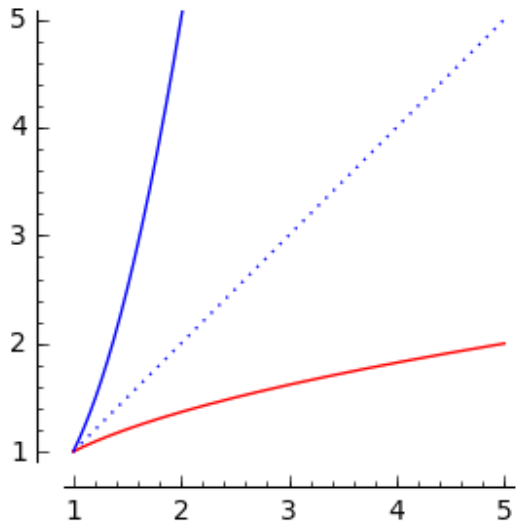
정의역을 제한한 함수는 순증가함수 형태임을 확인할 수 있다. 이제 이 함수의 역함수를 생각해보자. 우리가 역함수를 구할 때 역함수 $y=f^{-1}(x)$는 $x=f(y)$를 만족하므로 이를 이용하면 함수 $f(y)=x$를 y에 관해 정리하면 된다. 이를 Sage에서 구현한다면

```
sage : var('x,y')
sage : inv=solve(f(y)==x,y)
sage : inv
 [y == -1/2*sqrt(2*x - 1) + 1/2, y == 1/2*sqrt(2*x - 1) + 1/2]
```

우리가 제한한 함수는 정의역과 치역이 모두 양수이므로 음수가 나올 수 있는 첫

번째 결과는 역함수가 될 수 없고 리스트의 두 번째 결과가 역함수임을 알 수 있다. 이를 함수로 지정하고 그래프를 그려보자

```
sage : g(x)=inv[1].rhs()
sage : p=plot(f(x),x,1,5)
sage : q=plot(g(x),x,1,5,color='red')
sage : r=plot(x,x,1,5,linestyle=':')
sage : (p+q+r).show(figsize='4', aspect_ratio='1', ymax=5)
```



뮈 명령에서는 x축과 y축을 같은 비율로 그리는 옵션 **aspect_ratio='1'**을 사용하였다.

우리가 역함수의 미분을 배울 때 다음의 정리를 이용한다. 만약 f가 미분 가능한 일대일 함수로 역함수는 f^{-1}이고 $f'(f^{-1}(a)) \neq 0$이면

$$(f^{-1})'(a) = \frac{1}{f'(f^{-1}(a))}.$$

Sage에서 이 정리를 사용하기 위해서는 $f^{-1}(a)$을 **solve**명령을 이용하여 구해야 하는데 이 과정은 우리가 위에서 계산한 역함수를 직접 찾는 과정과 비교했을 때 계산과정에서 큰 차이가 없다. 따라서 역함수의 도함수는 역함수를 직접 계산한 다음에 그 함수의 도함수를 구하는 방식을 이용해도 충분하다.

3.3 미분의 응용

3.3.1 서로 관련된 변화율

먼저 Sage에서는 함수 **S(t)**의 도함수를 다음과 같이 표현함을 기억하자.

```
D[0](S)(t)=diff(S(t))
```

Sage에서 서로 관련된 변화율 문제를 풀기 위해 다음의 문제를 살펴보자. 맑은 날 야외에 있는 완벽한 구체 형태의 고무공 겉넓이가 $3cm^2/h$의 비율로 열을 받아 구의 형태를 유지하며 점점 팽창하고 있다. 반지름이 $2cm$일 때 고무공의 반지름은 얼마나 빨리 증가하는가?

이 문제를 풀기 위해 먼저 구의 겉넓이에 관련된 공식 $S=4\pi r^2$을 이용하자. 여기서 S는 구의 겉넓이이고 r은 구의 반지름 그리고 t는 시간이다. 그리고 r은 시간에 따라 점점 증가하므로 t에 관한 함수로 생각할 수 있고 겉넓이는 r에 관련된 함수이므로 따라서 t에 관한 함수가 된다.

```
sage : var('t,S,r')
sage : r(t)=function('r')(t)
sage : S(t)=function('S')(t)
sage : sa=S(t)==4*pi*(r(t))^2
sage : dsa=diff(sa,t)
sage : dsa
 D[0](S)(t) == 8*pi*r(t)*D[0](r)(t)
```

r과 **S**를 모두 **t**에 관한 함수로 지정하고 구의 겉넓이에 관한 식을 **sa**라고 지정하였다. 그리고 이 식에 대한 시간의 변화율을 알고 싶으므로 이 관계식을 **t**로 미분한 식을 **dsa**라고 지정하였다.

```
sage : sol=solve(dsa,diff(r(t)))
sage : sol
 [D[0](r)(t) == 1/8*D[0](S)(t)/(pi*r(t))]
```

반지름 **r**의 시간에 따른 증가 비율이 우리가 알고자 하는 값이므로 반지름의 시간 **t**에 관한 도함수를 중심으로 정리를 하면 위와 같은 결과를 얻을 수 있다. 이를 수식으로 적으면

$$r'(t)=\frac{1}{8}\frac{S'(t)}{\pi r(t)}$$

가 된다.

```
sage : Dr(t)=sol[0].rhs()
sage : n(Dr(t).substitute(r(t)==2,diff(S(t),t)==3),digits=5)
 0.059683
```

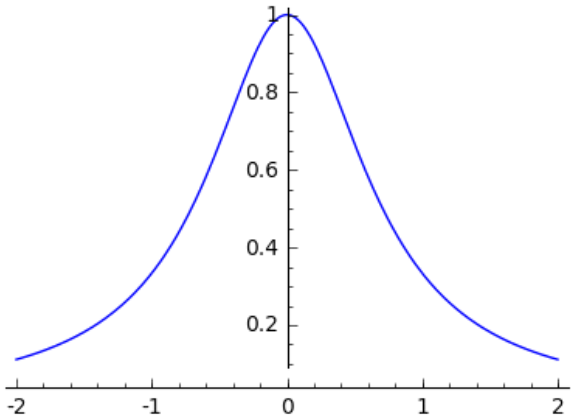
마지막으로 위에서 얻은 반지름의 시간에 따른 도함수 식을 **Dr(t)**라고 지정하고 우리가 알고 있는 정보인 반지름 $r(t)=2$ 그리고 겉넓이의 증가 비율 $S'(t)=3$을 대입하면 우리가 원하는 값을 얻을 수 있다.

반지름이 $2cm$일 때 반지름의 증가 비율은 시간당 $0.059683cm$가 된다.

3.3.2 극값과 변곡점

Sage를 이용해 극값을 찾는 방법에 대해 알아보자. 먼저 $x=a$에서 극값을 가지면 $f'(x)=0$이거나 $f'(x)$가 존재하지 않는다. 하지만 역은 성립하지 않는다. 또한 점 $(x,f(x))$가 변곡점이라면 $f''(x)=0$을 만족하지만 $f''(x)=0$이 된다고 해서 그 곳이 변곡점이 되는 것은 아니다. 함수 $f(x)=1/(x^2+1)$의 극값과 변곡점을 Sage를 이용해 찾아보자. 먼저 함수의 그래프를 그리자.

```
sage : f(x)=1/(2*x^2+1)
sage : plot(f(x),x,-2,2,figsize='4')
```



이 함수의 극값을 찾기 위해 먼저 $f'(x)$가 0이 되거나 존재하지 않는 x를 찾아보자

```
sage : Df(x)=diff(f(x),x)
sage : Df(x)
 -4*x/(2*x^2 + 1)^2
sage : solve(Df(x)==0,x)
 [x == 0]
```

함수 $f(x)$의 도함수를 **Df(x)**로 지정하고 도함수를 살펴보니 유리함수의 분모에 어떤 x를 넣어도 정의가 되므로 미분계수가 정의되지 않는 곳은 없다. 그리고 $f'(x)=0$이 되는 x를 찾아보았더니 $x=0$ 1개가 나왔다.

```
sage : diff(f(x),x,2).substitute(x=0);  f(0)
 -4
 1
```

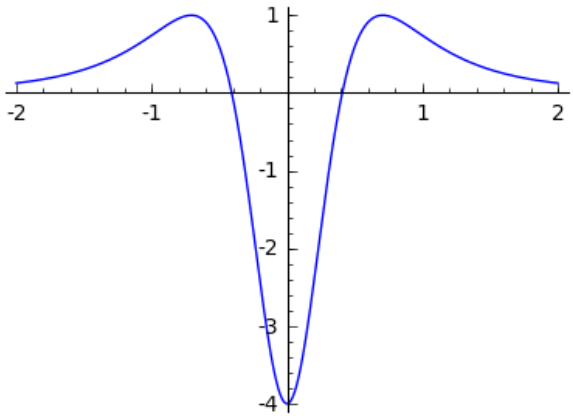
2계 도함수 판정법에 의해 $x=0$에서 $f'(0)=0$이고 $f''(0)=-4$이므로 $x=0$에서 극댓값을 가지고 극댓값은 1임을 알 수 있다. 변곡점을 찾기 위해 $f''(x)=0$이 되는 x를 찾아보자.

```
sage : sols=solve(diff(f(x),x,2)==0,x)
sage : sols
```

```
[x == -1/6*sqrt(6), x == 1/6*sqrt(6)]
```

2개의 x값에서 $f''(x)=0$이 됨을 알 수 있다. $x=a$ 위에서 변곡점이 있다면 $f''(x)$의 값이 $x=a$를 중심으로 부호가 바뀌어야 하므로 이를 확인하기 위해 2계도함수의 그래프를 그려보자.

sage : **plot(diff(f(x),2),x,-2,2,figsize='4')**



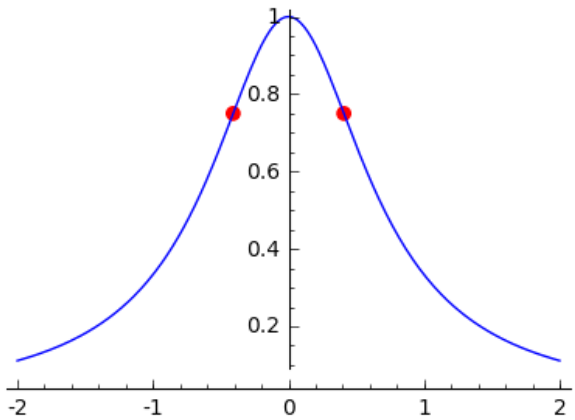
위에서 찾은 $f''(x)=0$이 되는 2개의 점과 이 그래프를 비교해 보면 2개의 점을 중심으로 양에서 음으로 혹은 음에서 양으로 부호가 바뀌는 것을 확인할 수 있다. 따라서 위에서 찾은 2개의 점 위에 변곡점이 존재하게 된다. 마지막으로 변곡점을 그래프로 표현하자.

sage : **p=plot(f(x),x,-2,2)**

sage : **Points=[(a.rhs(),f(a.rhs())) for a in sols]**

sage : **q=points(Points, color='red', pointsize='50')**

sage : **(p+q).show(figsize='4')**

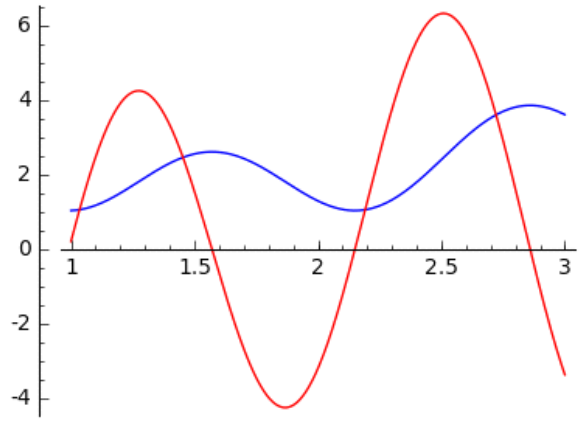


3.3.3 연속함수의 폐구간에서 최대 최소

다음은 폐구간 $[a,b]$에서 연속인 함수 $f(x)$의 최대 최소를 구하는 방법에 대해 알아보자. 최대 최소를 구하기 위해 사용하는 방법은 구간 양 끝점에 있는 함숫값과 임

계수의 함숫값을 비교하여 가장 큰 값이 최댓값이고 가장 작은 값이 최솟값이다. 예를 들어 함수 $f(x)=3\cos(\sin(x))+\sin(5x)$의 구간 $[1,3]$에서 최대 최소를 구해보자. 먼저 함수를 그래프로 그려 어림하자.

```
sage : f(x)=3*cos(sin(x))+sin(5*x)
sage : p=plot(f(x),x,1,3)
sage : q=plot(diff(f(x)),x,1,3,color='red')
sage : (p+q).show(figsize='4')
```



파란색은 함수 $f(x)$의 그래프이고 빨간색은 도함수 $f'(x)$의 그래프이다. 임계수의 위치를 어림해보면 1.5와 1.6사이에 $f'(x)=0$이 되는 x가 있고 비슷하게 2.1과 2.2 사이에 그리고 2.8과 2.9사이에도 임계수가 있는 것으로 생각된다. 먼저 이를 **solve** 명령을 이용해 찾아보자.

```
sage : solve(diff(f(x))==0,x)
 [sin(sin(x)) == 5/3*cos(5*x)/cos(x)]
```

Sage에서는 해를 대수적으로 찾을 수 없다고 나타낸다. 따라서 근사해를 구해야만 한다. 이를 위해 위에서 사용했던 이분법을 사용해보자.

```
sage : def bisect_method(f, a, b, ep):
     :     c=a
     :     if f(a)*f(b)<0:
     :         while True:
     :             c=(a+b)/2
     :             if abs(b-c)<ep: break
     :             if f(b)*f(c)<=0: a=c
     :             else: b=c
     :     return c
sage : g(x)=diff(f(x),x)
sage : cand=[1.5, 2.1, 2.8]
sage : candx=[]
sage : candx.append(1.0)
sage : for a in cand:
```

```
        :       candx.append(bisect_method(g, a, a+0.1, 1.0*10^(-10)))
    sage : candx.append(3.0)
    sage : candx
     [1.00000000000000,
     1.57079632682726,
     2.14985157521442,
     2.85926165049896,
     3.00000000000000]
```

def는 우리가 함수를 정의할 때 다룬 적이 있다. def 명령은 함수뿐만 아니라 명령을 비롯한 다양한 객체를 생성할 때 쓰는 명령이다. 위 명령에서는 def 명령을 이용하여 새로운 명령 bisect_method을 만들었으며 def를 이용해 명령을 만드는 구조는 다음과 같다.

```
def new_command(input1, input2, input3):
    command
    return result
```

따라서 이분법 명령의 정의할 때 사용한 def bisect_method(f, a, b, ep): 의미는 명령어의 이름은 bisect_method이고 이 명령을 수행하는데 (f, a, b, ep)같은 정보가 필요하며 명령에 대한 출력 값은 c 이라는 것이다. 명령의 내부를 살펴보면 아무 출력 값을 생성하기 위해 c=a를 가장 먼저 수행하게 했고 if f(a)*f(b)<0: 부분을 추가하여 이분법에 사용할 양쪽 끝 점이 중간값정리를 만족해야지 반복구문 안으로 들어갈 수 있게 지정했다. 그리고 나머지 내용은 이분법을 만들 때 쓴 명령과 동일하다.

그래프를 그려 추측한 $f'(x)=0$을 만족하는 x가 존재할 $[1.5,1.6]$, $[2.1,2.2]$, $[2.8,2.9]$의 왼쪽 끝점을 리스트 cand로 지정하고 최소 최대를 비교할 x값의 후보를 모아두는 리스트를 candx로 지정하였다. 그리고 리스트 candx안에 최대 최소를 찾는 구간의 양쪽 끝점과 이분법 명령을 이용해 찾은 $f'(x)=0$을 만족하는 x의 근사해를 넣었다.

```
sage : candy=[f(a) for a in candx]
sage : candy
 [1.04017596151550,
 2.62090691760442,
 1.03930857081941,
 3.87169193389720,
 3.62046509743671]
```

최대 최소가 존재하는 x값의 후보가 있는 candx안의 값을 함수에 넣어 나온 함숫값의 리스트를 candy로 지정하였다. 여기서 직접 최대 최소를 확인 가능하지만 리

스트의 양이 많을 것을 생각하여 리스트에서 가장 큰 값과 가장 작을 값을 반환하는 **max min**명령을 수행하겠다. 이 명령들은 리스트 안에서 가장 큰 수와 가장 작은 수를 반환 한다.

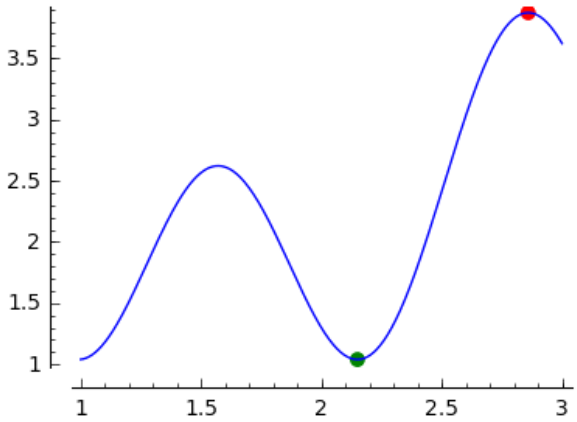
```
sage : max(candy); min(candy)
 3.87169193389720
1.03930857081941
```

함수의 최댓값 최솟값을 근사하는 방법을 알아보았다. 마지막으로 최대 최소가 존재하는 x를 찾아 그래프로 표현해 보자. **max min**명령은 리스트에서 가장 큰 값과 가장 작은 값을 알려주지만 그 위치가 어디인지 알려주지는 못한다. 이를 위해 다음의 명령을 수행하자.

```
sage : max_i=candy.index(max(candy))
sage : min_i=candy.index(min(candy))
sage : max_i; min_i
 3
 2
```

리스트에 **.index(a)** 명령을 이용하면 **a**값이 있는 위치를 반환한다. 만약 해당되는 값이 없으면 아무것도 반환하지 않는다. 가장 큰 값과 가장 작은 값의 위치를 반환하라고 지정했더니 큰 값의 위치는 3 작은 값의 위치는 2를 반환하였다.

```
sage : (maxx,maxy)=(candx[max_i], candy[max_i])
sage : (minx,miny)=(candx[min_i], candy[min_i])
sage : p=plot(f(x),x,1,3)
sage : q=point( (maxx,maxy), color='red', pointsize='50')
sage : r=point( (minx,miny), color='green', pointsize='50')
sage : (p+q+r).show(figsize='4')
```



3.3.4 뉴턴의 방법

방정식의 해를 근사적으로 찾는데 사용하는 방법인 이분법은 중간값정리를 만족하면 그 역할을 잘 수행하지만 이 방법에도 단점이 있다. 첫 번째로 해가 중근일 경

우 해는 존재하지만 해를 중심으로 중간값정리를 만족하지 않으므로 이분법을 사용할 수 없다. 또 한 가지 단점은 이분법은 반복 횟수를 늘려갈수록 오차가 $1/2$씩 줄어드는데 이러한 방법은 수렴 속도가 낮다고 얘기한다. 즉 이것보다 더 빠르게 해로 수렴하는 방법이 존재한다는 의미이다. 이분법의 단점을 극복하고 더 빠른 수렴 속도를 가지는 뉴턴의 방법을 알아보겠다.

간단하게 뉴턴의 방법을 유도해 보자. 뉴턴의 방법은 $f(x)=0$의 해를 구하기 위해 근사해 x_n에 대해 p에 대한 방정식 $f(p+x_n)=0$을 세우고 p에 대해 차수가 높은 항을 제거하여 p에 대한 '일차식'으로 만드는 것이 주요 아이디어다. 미분을 이용하여 $f(p+x_n)$을 선형근사하면

$f(p+x_n)\approx f(x_n)+f'(x_n)p$ 이므로 $p=-\dfrac{f(x_n)}{f'(x_n)}$이 되고 뉴턴의 방법은 다음 점화식의 극한값이 주어진 방정식의 해라고 주장하는 방법이다.

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}.$$

하지만 만약 $f'(x_n)=0$이 되는 x_n이 존재한다면 뉴턴의 방법은 제대로 역할을 못하게 된다.

- **알고리즘(뉴턴의 방법)** (미분가능함수f, 초기값x_0, 반복횟수N)
  1. 해에 근접한 초기값 x_0을 설정한다.
  2. 점화식 $x_{n+1}=x_n-\dfrac{f(x_n)}{f'(x_n)}$을 N만큼 반복하여 해를 추정한다.

뉴턴의 방법을 이용하여 이분법에서 풀었던 방정식 $\cos x=x$을 풀어보자. 이분법을 이용할 때 초기 구간을 $[0,1]$로 정하였는데 비슷하게 $x_0=0$으로 놓자.

```
sage : def newton_method(f, x0, N):
     :     c=x0
     :     for i in range(N):
     :         d=c-f(c)/diff(f(x)).substitute(x=c)
     :         c=d
     :     return c
sage : f(x)=cos(x)-x
sage : newton_method(f, 0.0, 10)
 0.739085133215161
```

이분법과 뉴턴의 방법을 비교하기 위해 다음을 생각해보자. 방정식 $\cos x=x$의 정확한 해는 모르므로 먼저 반복 횟수를 충분히 하여 오차가 매우 작은 근사해를 하나 지정하고 이분법과 뉴턴의 방법의 반복 횟수를 증가시키면서 이 근사해에 얼마나

가깝게 접근하는지 알아보자.

```
sage : exact=bisect_method(f, 0.0, 1.0, 100)
sage : exact
```

0.739085133215161

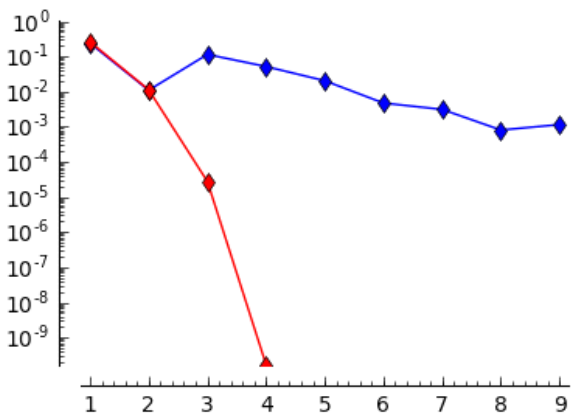
충분히 정확한 해를 이분법으로 100번 반복한 값을 채택한 근거는 이분법 명령에서 $\epsilon = 1.0e^{-15}$정도로 계산했더니 반복횟수가 50회 나왔고 따라서 100회 정도면 머신오차에 (컴퓨터가 인지할 수 없는 오차)에 충분히 가까울 거라고 추정하였다. (실제로 화면에 나오는 숫자도 소수점 15자리까지 나온다.)

```
sage : N=range(1,10)
sage : bisect_val=[ abs(exact-bisect_method(f,0.0,1.0,a)) for a in range(1,10) ]
sage : newton_val=[ abs(exact-newton_method(f,0.0,a)) for a in range(1,10) ]
sage : table(zip(N,bisect_val,newton_val))
```

1	0.239085133215161	0.260914866784839
2	0.0109148667848393	0.0112787346250832
3	0.114085133215161	0.0000277576962010029
4	0.0515851332151607	$1.70123359843899 \times 10^{-10}$
5	0.0203351332151607	0.000000000000000
6	0.00471013321516067	0.000000000000000
7	0.00310236678483933	0.000000000000000
8	0.000803883215160672	0.000000000000000
9	0.00114924178483933	0.000000000000000

이분법과 뉴턴의 방법을 1번부터 9번까지 반복하여 나온 값을 정확히 근사한해와의 절댓값 차이를 각각 **bisect_val**, **newton_val**이라고 지정하고 이를 **table**로 표현하였다. 2가지 방법 모두 오차가 0으로 수렴한다고 추정 가능하지만 이분법의 방법은 천천히 수렴하는 반면 뉴턴의 방법은 5번 반복했을때 이미 컴퓨터에서 오차를 계산하지 못할 만큼 수렴하였다. 마지막으로 이를 그래프로 비교해보자.

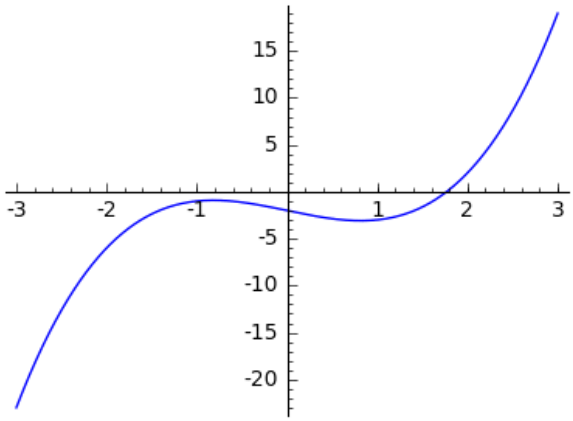
```
sage : p=line(zip(N,bisect_val),marker='d')
sage : q=line(zip(N,newton_val),marker='d',color='red')
sage : (p+q).show(scale='semilogy', ymax=1.0, figsize='4')
```



그래프에서 x축은 반복횟수로 정하였고 y축은 오차이며 **table**에서 얻은 값을 점으로 표시하고 이를 선으로 연결하였다. 특이한 점은 그리기 옵션에서 **scale='semilogy'**으로 지정하여 y축의 스케일을 로그스케일로 지정하였다. 로그스케일로 지정한 이유는 일반적으로 그래프를 그릴 때 사용하는 선형스케일은 0으로 접근하는 두 가지 방법에 대한 차이가 잘 드러나지 않기 때문이다. 그래프를 살펴보면 이분법은 천천히 0으로 접근하는 반면 뉴턴의 방법은 상당히 빨리 접근하고 있다.

뉴턴의 방법을 사용하는데 유의할 점을 알아보자. 먼저 뉴턴의 방법을 사용하기 위해서는 함수가 미분 가능해야 한다. 또한 초기값 x_0의 위치가 해와 상당히 가까워야 하는데 그렇지 않다면 수렴속도가 떨어지게 된다. 함수 $f(x)=x^3-2x-2$와 초기값을 1.0과 0.7로 선택했을 때를 비교해보자.

```
sage : f(x)=x^3-2*x-2
sage : plot(f(x),x,-3,3,figsize='4')
```



해는 1.5와 2사이에 있을 것이라 예측 가능하다.

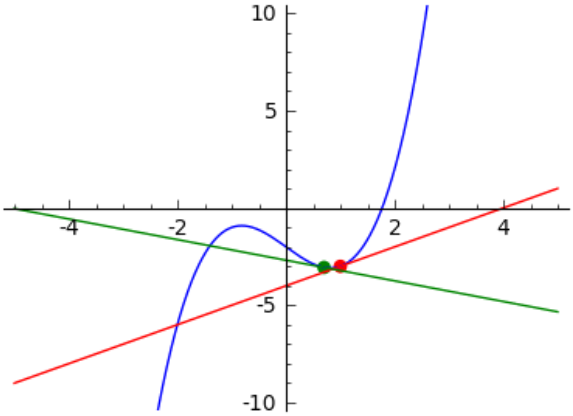
```
sage : N=range(1,10)
sage : newton_10=[ newton_method(f,1.0,a) for a in range(1,10) ]
sage : newton_07=[ newton_method(f,0.7,a) for a in range(1,10) ]
sage : table(zip(N,newton_10,newton_07))
```

1	4.00000000000000	−5.06792452830188
2	2.82608695652174	−3.44200253127823
3	2.14671901373924	−2.37186487111257
4	1.84232627714009	−1.65938189154669
5	1.77284763643924	−1.14019825288424
6	1.76930139743645	−0.507660466004837
7	1.76929235429736	−1.41691560042678
8	1.76929235423863	−0.917073589137094
9	1.76929235423863	0.874522717071314

같은 뉴턴의 방법이지만 초기값을 $x_0=1.0$으로 지정했을 때와 $x_0=0.7$로 지정했을

때의 반복하면서 생기는 값은 상당히 다른 것을 확인할 수 있다. 그 이유는 $x_0 = 1.0$과 $x_0 = 0.7$일 때 접선의 모양 때문이다.

```
sage : p=plot(f(x), x,-5,5)
sage : q=plot(diff(f(x)).substitute(x=1.0)*(x-1)+f(1), x,-5,5, color='red')
sage : r=plot(diff(f(x)).substitute(x=0.7)*(x-0.7)+f(0.7), x,-5,5, color='green')
sage : s=point([(1,f(1))],pointsize='40', color='red')
sage : t=point([(0.7,f(0.7))],pointsize='40', color='green')
sage : (p+q+r+s+t).show(figsize='4', ymax=10, ymin=-10)
```



$x = 1.0$일 때 접선의 x절편과 $x = 0.7$일 때 접선의 x절편이 뉴턴의 방법을 한번 적용했을 때 반환하는 값이 되는데 이 값이 서로 다름을 알 수 있다.

제 4장

적분과 그 응용

4.1 적분과 리만합

4.1.1 부정적분과 정적분 기본명령

$g(x)=f'(x)$에 대한 $f(x)$를 $g(x)$의 역도함수라 하고 이를 찾는 과정을 부정적분이라 한다. Sage에서는 부정적분을 다음의 명령어를 이용하여 쉽게 찾을 수 있다.

```
integral(f(x), x)
integrate(f(x), x)
```

몇 가지 예를 살펴보자.

$$\int \frac{x+1}{x^2+2x+1}dx$$

```
sage : integrate((x+1)/(x^2+2*x+1),x)
 1/2*log(x^2 + 2*x + 1)
```

$$\int \frac{2x}{\sqrt{x+1}}dx$$

```
sage : integrate(2*x/sqrt(x+1),x).simplify_full()
 4/3*sqrt(x + 1)*(x - 2)
```

$$\int \sec^3 x\,dx$$

```
sage : integrate(sec(x)^3,x)
 -1/2*sin(x)/(sin(x)^2 - 1) + 1/4*log(sin(x) + 1) - 1/4*log(sin(x) - 1)
```

$$\int e^x \sin x\,dx$$

```
sage : integrate(exp(x)*sin(x),x)
 -1/2*(cos(x) - sin(x))*e^x
```

Sage에서는 부정적분의 결과로 적분상수 C를 따로 표시하지 않으며 모든 유리함수를 비롯해 손으로 계산 가능한 초월함수는 Sage에서 결과 값을 보여준다. 또한 유리함수의 적분을 계산할 때 이용하는 유리함수의 부분분수 변환을 Sage에서 지원한

다.

```
function.partial_fraction(x)
```

```
sage : f1(x)=1/( (x+1)^2*(x^2+x+1)^3 )
sage : f1.partial_fraction(x)
 x |--> -(3*x + 1)/(x^2 + x + 1) - (2*x + 1)/(x^2 + x + 1)^2
 + 3/(x + 1) + 1/(x + 1)^2 - (x + 1)/(x^2 + x + 1)^3
```

그러나 Sage에서 모든 적분을 계산할 수 있는 것은 아니다.

$$\int e^{\sin x}\,dx$$

```
sage : integrate(exp(sin(x)),x)
 integrate(e^sin(x), x)
```

Sage에서 계산할 수 없는 부정적분은 적분형태 그래도 반환되며 이러한 함수는 손으로도 계산할 수 없다.

Sage에서 정적분 $\int_a^b f(x)\,dx$를 계산하는 명령은 다음과 같다.

```
integral(f(x), x,a,b)
integrate(f(x), x,a,b)
```

또한 수치근사 명령 **N()**을 이용하면 정적분을 바로 수치근사 가능하다.

```
sage : integrate(exp(sin(x)),x,0,1)
 integrate(e^sin(x), x, 0, 1)
sage : N(integrate(exp(sin(x)),x,0,1))
 1.6318696084180513
```

4.1.2 리만합과 표본점을 이용한 근사적분 개요

정적분의 수치근사 방법을 알아보기 전에 리만합을 정리해보자. 폐구간 $[a,b]$의 임의의 분할 $P=\{x_0,x_1,\dots,x_n\}$ $(a=x_0<x_1<\dots<x_n=b)$에 대해 $\triangle x_i=x_i-x_{i-1}$과 $[x_{i-1},x_i]$사이에 있는 임의의 점 x_i^*(포본점)을 이용한 다음을 리만합이라 한다.

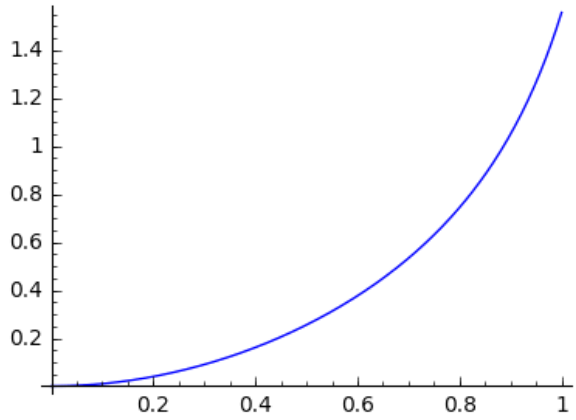
$$\sum_{i=1}^{n} f(x_i^*)\triangle x_i.$$

그리고 분할에서 $\triangle x_i$의 가장 큰 값이 0으로 가까워지는 어떠한 분할에서 임의의 표본점을 이용한 리만합이 하나의 값으로 수렴하면 우리는 함수 $f(x)$를 구간 $[a,b]$에서 적분가능이라 하고 다음의 형태로 표현한다.

$$\int_a^b f(x)\,dx=\lim_{\max\triangle x_i\to 0}\sum_{i=1}^{n} f(x_i^*)\triangle x_i$$

함수 $f(x)$가 구간 $[a,b]$에서 적분 가능할 때 임의의 분할과 표본점에 대해 리만합의 극한이 한 값으로 수렴하므로 우리는 분할과 표본점을 적당히 선택하여 적분값을 근사할 수 있다. 먼저 사용의 편의를 위해 앞으로 분할은 균등분할을 이용하도록 하자. 정적분 $\int_0^1 \tan x^2\,dx$를 리만합을 통해 근사해 보자. 먼저 이 함수의 그래프를 그려보고 **integrate**명령을 이용해 정적분을 수행해 보자.

```
sage : f(x)=tan(x^2)
sage : plot(f(x),x,0,1,figsize='4')
```




```
sage : integrate(f(x),x,1,2)
 integrate(tan(x^2), x, 1, 2)
```

정적분 $\int_0^1 \tan x^2\,dx$는 역도함수와 미분적분학의 기본정리에 의해서는 정확한 값을 구할 수 없다. 먼저 균등분할의 개수 n과 표본점을 왼쪽 끝점을 이용하여 리만합을 구해보자.

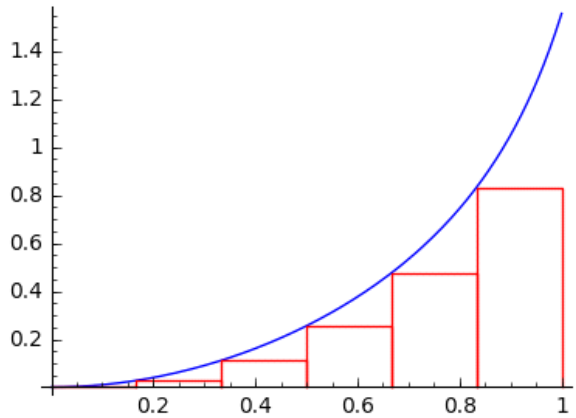
```
sage : var('k'); intval=[];
sage : for n in [2..10]:
     :     d=1.0/n
     :     xstar=f((k-1)/n)
     :     intval.append(N(sum(xstar*d,k,1,n)))
sage : intval
 [0.127670960610518,
 0.195930687019432,
 0.237090275655799,
 0.264469496318743,
 0.283959080062198,
 0.298525530445907,
 0.309818494341476,
 0.318826592780326,
 0.326177681481724]
```

리만합이 $\sum_{k=1}^{n} f(x_k^*)\triangle x_k$임을 생각하자. 분할은 균등분할 이므로 적분 구간 $[0,1]$을 n으로 나눈 것이 분할의 길이가 될 것이고 이를 **d**라고 지정하였다. 그리고 각 분할의 왼쪽 끝점은 첫 번째 분할에서 $f(0/n)$ 두 번째 분할에서 $f(1/n)$ k번째 분할에서 $f((k-1)/n)$이 되므로 이를 **xstar**라고 지정하였다. 그리고 마지막으로 k가 1부터 n까지 증가하는 동안 $f(k)$를 모두 더하라는 $\sum_{k=1}^{n} f(k)$의 명령어 **sum**을 사용하였다. 이 명령어의 사용은 다음과 같다.

```
sum(f(k), k,1,n)
```

다음은 특정 n에 대해 왼쪽 끝점의 리만합을 그래프로 비교해 보자.

```
sage : f(x)=tan(x^2)
sage : p=plot(f(x),x,0,1)
sage : n=6
sage : for k in [1..n]:
     :     xpt1=(k-1)/n
     :     xpt2=k/n
     :     ypt=f( N((k-1)/n) )
     :     p = p + line([ (xpt1,0.0), (xpt1,ypt), (xpt2, ypt), (xpt2,0.0) ], color='red')
sage : p.show(figsize='4')
```

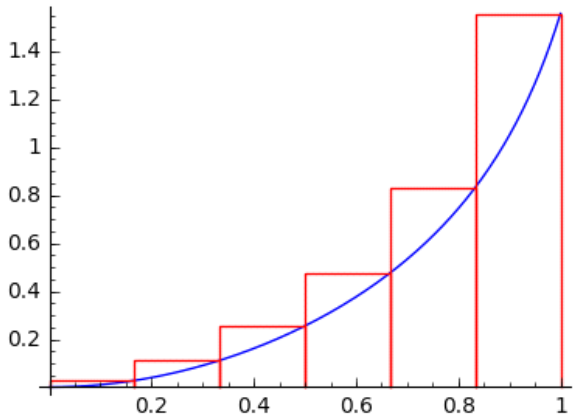


$n=6$으로 선택했고 **xpt1 xpt2 ypt**는 각각 분할에서 왼쪽 끝 x점 오른쪽 끝 x점 그리고 표본점의 값 $f(x^*)$를 의미한다. 이를 이용하여 막대를 그리는 명령을 입력하였다.

표본점을 오른쪽 끝점으로 선택한 리만합의 근사 적분을 알아보기 전에 먼저 그래프로 이를 그려보자.

```
sage : f(x)=tan(x^2)
sage : p=plot(f(x),x,0,1)
```

```
sage : n=6
sage : for k in [1..n]:
     :     xpt1=(k-1)/n
     :     xpt2=k/n
     :     ypt=f( N(k/n) )
     :     p = p + line([ (xpt1,0.0), (xpt1,ypt), (xpt2, ypt), (xpt2,0.0) ], color='red')
sage : p.show(figsize='4')
```



표본점을 왼쪽끝점과 오른쪽끝점으로 선택했을 때의 리만합 그래프를 그려보니 적분값 $\int_0^1 \tan x^2 dx$는 표본점을 왼쪽끝점으로 선택했을 때의 리만합 보다는 항상 크고 표본점을 오른쪽끝점으로 선택했을 때의 리만합 보다는 항상 작다는 것을 알 수 있다. n을 상당히 크게 하고 리만합을 비교하면 다음과 같다.

```
sage : var('k'); n=2000; d=1.0/n
sage : xstar_l=f((k-1)/n)
sage : xstar_r=f(k/n)
sage : intval_l=(N(sum(xstar_l*d,k,1,n)))
sage : intval_r=(N(sum(xstar_r*d,k,1,n)))
sage : [intval_l, intval_r]
 [0.398025235395902, 0.398803939258229]
```

n을 2000으로 크게 분할했음에도 불구하고 두 리만합의 차이가 있는 것을 확인할 수 있다. 따라서 더 정확한 적분값을 찾기 위해서는 n을 더 크게 잡아야 하고 이는 계산하는데 더 많은 컴퓨터 자원이 필요하다는 것을 의미한다. 적분값을 리만합을 통해 근사할 때 표본점을 왼쪽끝점이나 오른쪽끝점으로 이용하는 것은 비효율적인데 그 이유를 다음 절에서 알아보자.

4.2 다양한 근사적분 방법

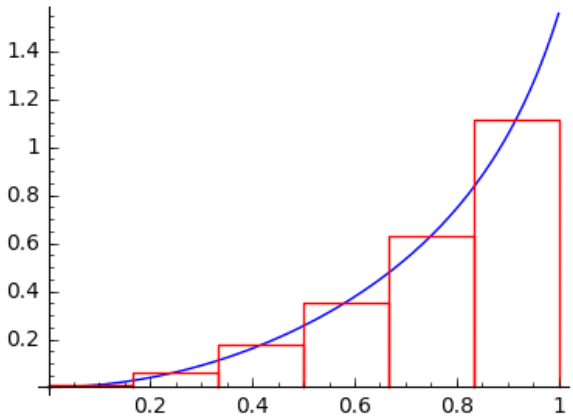
4.2.1 중점 법칙과 사다리꼴 법칙

리만합에서 왼쪽끝점과 오른쪽끝점에 대한 표본점의 대체방법으로 분할의 가운데 점을 사용할 수 있는데 이를 중점 법칙이라 한다. 즉 리만합 $\sum_{k=1}^{n} f(x_k^*)\triangle x_k$에서

$$x_k^* = \frac{x_{k-1} + x_k}{2}$$

로 지정한 것이다. 먼저 앞의 절에서 살펴본 문제에 중점 법칙을 적용하여 그래프로 표현해 보자.

```
sage : f(x)=tan(x^2)
sage : p=plot(f(x),x,0,1)
sage : n=6
sage : for k in [1..n]:
     :     xpt1=(k-1)/n
     :     xpt2=k/n
     :     ypt=f( N((2*k-1)/(2*n)) )
     :     p = p + line([ (xpt1,0.0), (xpt1,ypt), (xpt2, ypt), (xpt2,0.0) ], color='red')
sage : p.show(figsize='4')
```



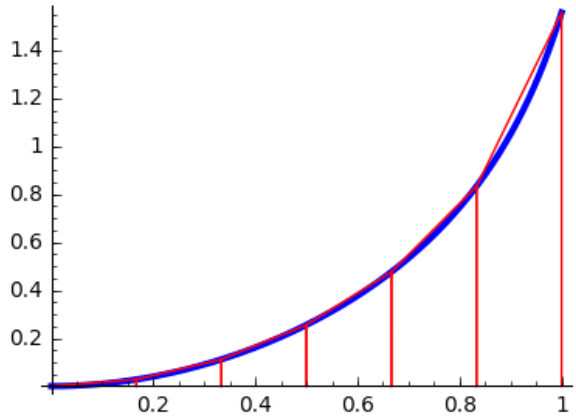
각 분할에서 표본점이 분할의 가운데 점으로 지정되어 있어 왼쪽끝점이나 오른쪽끝점보다는 적분값이 정확할 것이라 예측할 수 있다. 비슷하게 표본점을 왼쪽끝점과 오른쪽끝점으로 잡은 표본점의 평균으로 선택하는 사다리꼴 방법이 있다. 즉 리만합 $\sum_{k=1}^{n} f(x_k^*)\triangle x_k$에서

$$f(x_k^*) = \frac{f(x_{k-1}) + f(x_k)}{2}$$

로 지정한 것이다. 비슷하게 사다리꼴 법칙을 그래프로 표현해 보자. 두 표본점을 평균한 값은 두 표본점을 직선으로 연결한 값과 같으므로 직선으로 연결한 형태로

표현하겠다.

```
sage : f(x)=tan(x^2)
sage : p=plot(f(x),x,0,1, thickness='3')
sage : n=6
sage : for k in [1..n]:
     :     xpt1=(k-1)/n
     :     xpt2=k/n
     :     ypt1=f( N( (k-1)/n ) )
     :     ypt2=f( N( k/n ) )
     :     p = p + line([ (xpt1,0.0), (xpt1,ypt1), (xpt2, ypt2), (xpt2,0.0)], color='red')
sage : p.show(figsize='4')
```



그래프를 살펴보면 사다리꼴 법칙이나 중점 법칙이 왼쪽끝점이나 오른쪽끝점을 선택할 때보다 더 정확하게 적분값을 근사할 것이라 예측된다. 이를 확인하기 위해 역도함수와 미분적분학 기본정리에 의해 계산할 수 있는 적분값에 다양한 표본점을 이용해 근사한 값을 비교하자. 먼저 다음의 적분을 살펴보자.

$$\int_0^1 \tan^{-1} x\, dx.$$

```
sage : integrate(arctan(x),x,0,1)
 1/4*pi - 1/2*log(2)
sage : integrate(arctan(x),x,0,1).N()
 0.438824573117476
```

이 적분값에 n을 10, 20, 40, 80으로 하고 왼쪽끝점 오른쪽끝점 중점 법칙 사다리꼴 법칙을 이용한 리만합과의 차를 절댓값(오차)으로 살펴보자.

```
sage : f(x)=arctan(x); var('k')
sage : int_l=[]; int_r=[]; int_mid=[]; int_tra=[];
sage : exact=integrate(arctan(x),x,0,1).N()
sage : nn=[10, 20, 40, 80, 160]
sage : for n in nn:
```

```
        :     d=1.0/n
        :     xstar_l=f( (k-1)/n )
        :     xstar_r=f( k/n )
        :     xstar_mid=f( 0.5*(((k-1)/n)+(k/n)) )
        :     xstar_tra=0.5*(f((k-1)/n)+f(k/n))
        :     int_l.append( abs(exact-sum(xstar_l*d,k,1,n)).N(digits=4) )
        :     int_r.append( abs(exact-sum(xstar_r*d,k,1,n)).N(digits=4) )
        :     int_mid.append( abs(exact-sum(xstar_mid*d,k,1,n)).N(digits=4) )
        :     int_tra.append( abs(exact-sum(xstar_tra*d,k,1,n)).N(digits=4) )
sage : Head=[("n", "Left", "Right", "Mid", "Trapezoidal")]
sage : Result=zip(nn,int_l,int_r,int_mid,int_tra)
sage : table(Head+Result, header_row=True, frame=True, align='center')
```

n	Left	Right	Mid	Trapezoidal
10	0.03968	0.03886	0.0002086	0.0004110
20	0.01974	0.01953	0.00005210	0.0001054
40	0.009844	0.009792	0.00001302	0.00002599
80	0.004910	0.004907	3.255×10^{-6}	1.311×10^{-6}
160	0.002452	0.002457	8.138×10^{-7}	1.609×10^{-6}

위 결과를 통해 5가지 정도를 분석할 수 있다.

(1) n이 커질수록 오차는 작아진다.

(2) 중점 법칙과 사다리꼴 법칙의 경우 왼쪽 끝점이나 오른쪽 끝점보다 오차가 훨씬 적다.

(3) 왼쪽끝점과 오른쪽끝점을 이용할 경우 n이 2배씩 증가할수록 오차는 $1/2$로 줄어든다.

(4) 중점 법칙과 사다리꼴 법칙을 이용할 경우 n이 2배씩 증가할수록 오차는 $1/4$로 줄어든다.

(5) 중점 법칙의 오차 크기는 사다리꼴 법칙의 오차 크기에 약 절반 정도이다.

미분적분학 책에 증명되어 있지는 않지만 중점 법칙과 사다리꼴 법칙의 오차 한계(오차가 생길 수 있는 최대값)는 다음과 같다. $a \le x \le b$일 때 $|f''(x)| \le K$라고 하자. E_T와 E_M을 각각 사다리꼴 법칙과 중점 법칙의 적분값에 대한 오차라면

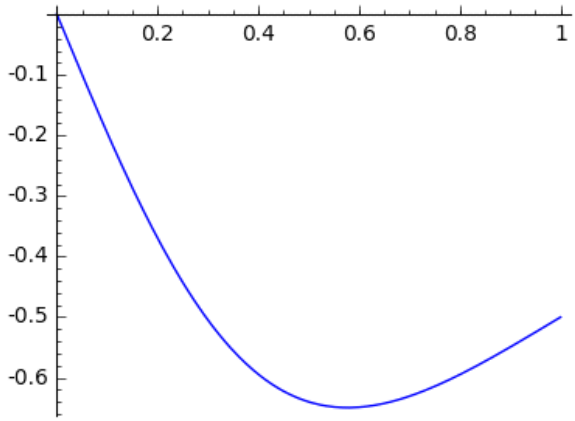
$$|E_T| \le \frac{K(b-a)^3}{12n^2}, \quad |E_M| \le \frac{K(b-a)^3}{24n^2}$$

를 각각 만족한다.

오차 한계를 다음과 같이 활용 가능하다. 먼저 적분 $\int_0^1 \tan^{-1} x\,dx$을 계산하기 위해 사다리꼴 법칙을 이용했을 때 오차가 0.0001이하의 정확한 값을 찾기 위해서는 n의

값을 얼마만큼 크게 해야 하는지 알아보자. 먼저 $|f''(x)| \le K$를 만족하는 K를 찾아보자.

```
sage : f(x)=arctan(x)
sage : g(x)=diff(f(x),x,2)
sage : plot(g(x),x,0,1,figsize='4')
```




```
sage : sol=solve(diff(g(x))==0,x); sol
 [x == -1/3*sqrt(3), x == 1/3*sqrt(3)]
sage : K=abs(g(sol[1].rhs()).N()); K
 0.649519052838329
```

$g(x)=f''(x)$로 놓고 이 함수의 구간 $[0,1]$에서의 최대 최소를 찾기 위해 그래프를 그려보니 $g'(x)=0$인 x에서 함수의 최솟값이 있는 것을 확인할 수 있었고 이 x를 찾아 함수 $g(x)$에 넣어 K값을 찾았다.

```
sage : var('n');
sage : result=solve( 0.0001==K/(12*n^2),n); result
 [n == -20/7022497*sqrt(9502595)*sqrt(7022497),
  n == 20/7022497*sqrt(9502595)*sqrt(7022497)]
sage : result[1].rhs().N()
 23.2651214775525
```

찾은 값을 오차 한계 부등식 $|E_T| \le \frac{K(b-a)^3}{12n^2}$에 넣고 부등식을 등식으로 바꾼 뒤 n을 찾아보니 23.26정도의 값이 나왔다. 즉 $n=23.26$번 정도로 하면 오차는 최대 0.0001정도 된다는 뜻이다. 따라서 n을 24번 이상으로 해야 오차가 최대 0.0001이하로 하는 근사 적분값을 찾을 수 있다.

4.2.2 심프슨 법칙

심프슨 법칙은 중점 법칙과 사다리꼴 법칙 보다 더 정확한 근사적분을 위해 고안되었다. 중점 법칙이나 사다리꼴 법칙이 구간의 양 끝점을 평균하는 직선으로 연결하는 방법을 이용한다면 심프슨 법칙은 이를 곡선으로 연결하여 더 정확한 근사를 시

도한다. 주어진 세 점 (x_0,y_0), (x_1,y_1), (x_2,y_2)을 연결하는 이차곡선은 다음의 형태로 표현할 수 있다.

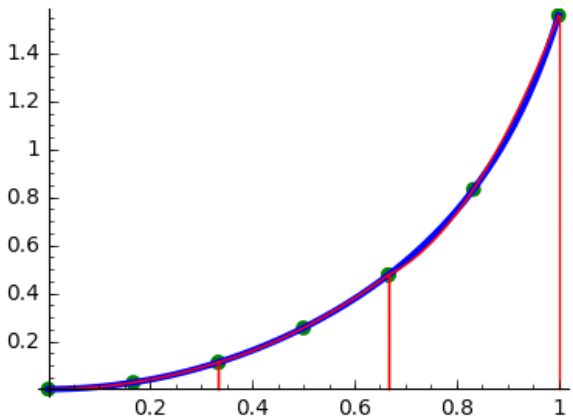
$$f(x)=y_0L_0(x)+y_1L_1(x)+y_2L_2(x)$$

여기서

$$L_0=\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)},\ L_1=\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)},\ L_2=\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

가 된다. 이러한 형태를 라그랑지 형태라고 한다. 이 형태는 연립방정식을 풀면 쉽게 유도할 수 있다.

```
sage : f(x)=tan(x^2)
sage : p=plot(f(x),x,0,1,thickness='3')
sage : n=6
sage : for k in [1..n/2]:
     :     x0=2*(k-1)/n; x1=(2*k-1)/n; x2=2*k/n;
     :     y0=f( N(x0) ); y1=f( N(x1) ); y2=f( N(x2) )
     :     L0(x)=( (x-x1)*(x-x2) )/( (x0-x1)*(x0-x2) )
     :     L1(x)=( (x-x0)*(x-x2) )/( (x1-x0)*(x1-x2) )
     :     L2(x)=( (x-x0)*(x-x1) )/( (x2-x0)*(x2-x1) )
     :     quad(x)=y0*L0(x)+y1*L1(x)+y2*L2(x)
     :     p = p + line([ (x0,0.0), (x0,y0)], color='red')
     :     p = p + line([ (x2,0.0), (x2,y2)], color='red')
     :     p = p + plot(quad(x),x,x0,x2, color='red')
     :     p = p + point([(x0,y0), (x1,y1), (x2,y2)],color='green', pointsize='50')
sage : p.show(figsize='4')
```



다른 근사적분을 다룰 때 사용하던 $\tan x^2$함수에 대해 $n=6$인 균등분할에서 세 점을 한 구간으로 설정하고 라그랑지 형태를 이용해 이를 지나는 이차곡선을 그렸다. 심프슨 법칙은 목표 함수인 파란색 함수의 적분을 위해 구간별로 세 점을 이은 빨간색 이차곡선을 적분하여 이를 근사하자는 것이다. 균등분할이므로 $h=(x_2-x_0)/2$라고 한다면 다음을 쉽게 계산할 수 있다.

$$\int_{x_0}^{x_2}(y_0L_0(x)+y_1L_1(x)+y_2L_2(x))\,dx=\frac{h}{3}[f(x_0)+4f(x_1)+f(x2)].$$

심프슨 법칙을 정리하면 구간 $[a,b]$를 $2n$개로 균등 분할하고 $h=\triangle x_k=(b-a)/2n$와 $m=2n$로 지정한 리만합 $\sum_{k=1}^{m} f(x_k^*)\triangle x_k$에서

$$f(x_k^*)=\frac{f(x_{2k-2})+4f(x_{2k-1})+f(x_{2k})}{3}$$

로 지정한 것이다. 사다리꼴 법칙과 중점 법칙에 사용한 근사적분을 심프슨 법칙에 적용해 보고 이를 비교해 보자.

```
sage : f(x)=arctan(x); var('k')
sage : int_mid=[]; int_tra=[]; int_sim=[];
sage : exact=integrate(arctan(x),x,0,1).N()
sage : nn=[10, 20, 40, 80, 160]
sage : for n in nn:
     :     d=1.0/n
     :     xstar_mid=f( 0.5*(((k-1)/n)+(k/n)) )
     :     xstar_tra=0.5*(f((k-1)/n)+f(k/n))
     :     int_mid.append( abs(exact-sum(xstar_mid*d,k,1,n)).N(digits=4) )
     :     int_tra.append( abs(exact-sum(xstar_tra*d,k,1,n)).N(digits=4) )
     :     xstar_sim=( f((2*k-2)/n) + 4*f((2*k-1)/n) + f(2*k/n) )/3
     :     int_sim.append( abs(exact-sum(xstar_sim*d,k,1,n/2)).N(digits=9) )
sage : Head=[("n", "Mid", "Trapezoidal", "Simpson")]
sage : Result=zip(nn,int_mid,int_tra,int_sim)
sage : table(Head+Result, header_row=True, frame=True, align='center')
```

n	Mid	Trapezoidal	Simpson
10	0.0002086	0.0004110	1.40724296×10^{-6}
20	0.00005210	0.0001054	8.70677468×10^{-8}
40	0.00001302	0.00002599	5.35419531×10^{-9}
80	3.255×10^{-6}	1.311×10^{-6}	3.06044967×10^{-10}
160	8.138×10^{-7}	1.609×10^{-6}	3.68345354×10^{-11}

심프슨 법칙은 중점 법칙이나 사다리꼴 법칙에 비해 훨씬 적은 오차가 발생하고 n이 2배 증가할수록 오차는 $1/16$으로 줄어드는 것을 확인할 수 있다. 또한 심프슨의 법칙에 대한 오차 한계는 다음과 같다.

$a\le x\le b$에서 $|f^{(4)}(x)|\le K$라고 하고 E_S를 심프슨 법칙의 오차 한계라고 하면

$$|E_S|\le K\frac{(b-a)^5}{180n^4}$$

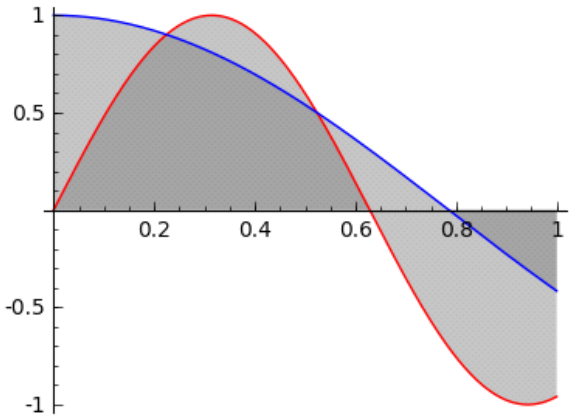
을 만족한다.

4.3 적분의 응용

4.3.1 곡선 사이의 넓이

곡선 사이의 넓이는 구하는데 중요한 계산은 구간에서 두 함수의 차를 어떻게 계산할지이다. Sage에서는 먼저 그래프를 그려 두 함수의 정의역에 따른 함숫값을 비교하고 근을 구하는 명령을 통해 구간을 설정하고 적분 명령을 통해 넓이를 계산할 것이다. $f(x)=\sin 5x$, $g(x)=\cos 2x$, $x=0$, $x=1$로 둘러싸인 영역의 넓이를 구하는 방법을 예로 들자.

```
sage : f(x)=sin(5*x); g(x)=cos(2*x)
sage : p=plot(f(x),x,0,1, color='red', fill=True)
sage : q=plot(g(x),x,0,1, fill=True)
sage : (p+q).show(figsize='4')
```



영역을 구분하기 위해 **fill=True** 옵션을 추가하였다. 우리가 구하는 넓이의 영역은 연한회색으로 구분되어진 곳이다. 구간별로 $f(x)-g(x)$혹은 $g(x)-f(x)$를 이용해 넓이를 구해야 하므로 $f(x)=g(x)$가 되는 x를 먼저 찾자.

```
sage : solve(f(x)==g(x),x)
 [sin(5*x) == cos(2*x)]
```

$f(x)=g(x)$가 되는 x를 Sage에서 대수적으로 구할 수 없으므로 뉴턴의 방법을 이용하여 근사적으로 찾도록 하자.

```
sage : def newton_method(f, x0, N):
     :     c=x0
     :     for i in range(N):
     :         d=c-f(c)/diff(f(x)).substitute(x=c)
     :         c=d
     :     return c
sage : pt1=newton_method(f-g,0.2,5)
sage : pt2=newton_method(f-g,0.5,5)
```

```
sage : f(pt1)-g(pt1); f(pt2)-g(pt2);
 0.000000000000000
 -3.330669073875470e-16
```

$f(x)=g(x)$를 만족하는 x중에서 $x=0.2$부근에 있는 값을 **pt1**이라고 하고 $x=0.5$부근에 있는 값을 **pt2**라고 한 다음에 이를 뉴턴방법으로 찾고 $f(x)-g(x)$를 계산하니 0에 아주 가까운 숫자가 나왔다. 마지막으로 우리가 원하는 영역의 넓이를 구하기 위해 다음의 계산

$$\int_0^{pt1} g(x)-f(x)\,dx+\int_{pt1}^{pt2} f(x)-g(x)\,dx+\int_{pt2}^{1} g(x)-f(x)\,dx$$

를 적용하면 된다.

```
sage : int1=integrate(g(x)-f(x),x,0,pt1)
sage : int2=integrate(f(x)-g(x),x,pt1,pt2)
sage : int3=integrate(g(x)-f(x),x,pt2,1)
sage : int1+int2+int3
 0.399203142999404 35
```

만약 적분을 역도함수와 미분적분학 기본정리에 의해 계산하지 못할 경우 근사 적분을 이용해야 한다.