Computational Discrete Mathematics: Handout 01

Curtis Bright

September 9, 2021

1 Sage Introduction

This document aims to give a crash-course to Sage. There are many additional resources for help, including the built-in documentation (discussed below), the **official Sage tutorial**, and the (highly recommended) open textbook **Computational Mathematics with SageMath**.

Sage is free and open source. Information on running a local installation can be found on the **Sage installation guide**. Alternatively, Sage can be run "in the cloud" by making a (free) account on the **CoCalc website** or by uploading a Jupyter notebook to a public git repository and using **mybinder.org**.

This document is written as a **Jupyter notebook**, the most common (and convenient) way to write and execute Sage code. A notebook is composed of *cells*. Most of the cells in this notebook consist of an Input section (containing Sage code) and (potentially) an output section (containing the result of evaluating that Sage code) — some code cells simply perform a computation without returning anything (for instance, updating the values of variables). A few cells (including the current one) consist of formatted text and LaTeX equations, written using the Markdown markup language. A third type of cell contains plain, unformatted text.

To execute a piece of Sage code, click on the Input section of the corresponding code cell and hit Shift + Enter (only hitting Enter simply adds a new line). The reader should execute each statement as they work through the notebook, and is encouraged to modify the code and play around as they go. Note that skipping a cell may result in errors when later cells are executed (for instance, if one skips a code block defining a variable and later tries to run code calling that variable). To add a new cell, click to the left of any cell and press the "a" key (to insert above) or the "b" key (to insert below). To delete a cell, click to the left of a cell and press the "d" key twice. These (and other) tasks can also be accomplished through the menu bars at the top of the page.

This introduction is based off of a worksheet originally written by Steven Melczer.

1.0.1 Part 1: The Basics of Sage

We begin with an explanation of arithmetic in Sage, if statements, for and while loops, and Sage functions (in the programming sense; symbolic mathematical functions are described in Part 2 below).

[ ]:
```
# Any text on a line after a '#' symbol is considered a comment, and is not␣
 ↪evaluated by Sage
# Sage can be used as a calculator using usual math notation
```

```
# (recall you need to click on a cell and hit Shift + Enter to evaluate it)
1 + 3*4
```

[ ]:
```
# Code in a Sage cell should be entered in sequence, with one "instruction" per line
# (multiple instructions can be put on the same line using a semicolon -- see examples below)
# The result of previous executions of Sage cells (both the current cell and other cells) is stored by Sage
# All (non-commented) code is executed, however only the output of the final line is printed by default
1 + 2
2 ^ 3
```

[ ]:
```
# Variables are defined using the "=" operator
# Note that some operations (such as variable assignment) do not have an output
# So we add another line to print the value of our variable
mult = 11 * 12
mult
```

[ ]:
```
# randint(a, b) returns a random integer between a and b inclusive
A = randint(0, 10^9)
B = randint(0, 10^9)
A+B
```

[ ]:
```
# Sage can compute with large integers
A*B
```

[ ]:
```
# Even very large integers can be computed exactly
A^100
```

[ ]:
```
# To print other lines, use the print command
print(2^3) # This is an integer
print(3/9) # This is an exact rational number
print(3.0/9.0) # This is a floating point number
print(11 % 3) # This is 11 mod 3
```

[ ]:
```
# Multiple instructions can be put on the same line using a semicolon
# Again, only the output of the final line is displayed by default
2+2; print(1+2); 3*5
```

[ ]:
```
# The "show" command outputs a latex-like pretty output
# Sage knows common math constants such as pi (lower case), e, and I (or the alternative i)
# Most common mathematical functions are supported by default
show(5/10 + 1/pi)
```

```
show(sin(pi/3))
show(log(2))
show(log(2).n(1000)) # Adding ".n()" gives a numerical approximation (can use ".n(k)" to get k bits)
show(exp(i*2*pi/3))
show(exp(I*2*pi/101))
```

[ ]:
```
# Sage computes with *exact* rational numbers
# For example, the following computes the sum of 1 + 1/2 + ... + 1/100
show(add(1/n for n in (1..100)))
```

[ ]:
```
# Sage also has support for matrices and vectors
A = random_matrix(ZZ, 5)
b = random_vector(ZZ, 5)
show(A)
show(b)
```

[ ]:
```
# Sage can solve the linear system Ax = b for x using the solve_right method of matrices
# (named because x appears to the right of A in the system to solve)
show(A.solve_right(b))
```

[ ]:
```
# There are many useful built-in functions on matrices (e.g., to compute the determinant)
A.determinant()
```

[ ]:
```
# Sage can even deal with matrices with symbolic entries
B = Matrix(SR, A)
B[4,4] = x
show(B)
show(B.solve_right(b))
```

[ ]:
```
# Mathematical objects and variables can be inserted in print commands using commas,
# or using empty curly braces {} and .format(math1,math2,...)
print("The area of a circle of radius 1 is approximately", pi.n())
print("The square-root of {} is approximately {}".format(log(2), sqrt(log(2)).n()))
```

[ ]:
```
# To access the help page for a function, type the name of the function and then add a question mark
# For instance, evaluating the expression "sin?" (without quotes) gives the help page for sine
# THIS WILL OPEN A NEW WINDOW AREA
# Using two question marks (e.g. "sin??") shows the source code of the function
```

```
# (Note that for many low level functions like sin Sage relies on outside␣
 ↪packages
# and the source code is not very informative)

# Similar information is provided by the "help" command, which prints below the␣
 ↪current cell
print?
```

[ ]:
```
# The values "_", "__", and "___" (using one, two, and three underscores)
# are the last three output values. Printing displays content but does *not*
# return a value, so a block ending with a print statement has no output
print(_)
print(__)
```

[ ]:
```
# The output of previous cells can be directly accessed using the syntax Out[k]
# NOTE: cells are numbered by execution order --  running cells repeatedly or␣
 ↪in
# different orders will change their numbering!
Out[13]
```

[ ]:
```
# Sage is built on Python, and uses much of the same syntax.
# In particular, indentation is extremely important.
# If statements are defined with indentation
a = 2

if a<0:
    print("Negative")
elif a==0:
    print("Zero")
else:
    print("Positive")
```

[ ]:
```
# In addition, an if statement can be written in one line
a = 2
if a>0: print("Positive")
```

[ ]:
```
# Functions are also defined with indentation
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
print(fact(10))
print(fact) # Prints reference to the function
```

[ ]:
```
# running this code displays the source code for the function fact
fact??
```

[ ]:
```
# Lists in Sage are defined with square brackets
LST = [1,2,3,4]

print(LST)
print(len(LST)) # Print the length of the list
print(LST[0]) # Access elements from the left (starting at index 0)
print(LST[1]) # Print second element of the list
print(LST[-2]) # Negative indices access elements from the right
print(LST[1:3]) # Define sublist
print(LST[1:-1]) # Define sublist using negative index
print(LST[1:]) # Define sublist from a fixed index to end of list
```

[ ]:
```
# Lists can be concatenated with '+'
# Strings can be concatenated similarly (they are lists of characters)
print([1,2,3] + ["a","b","c"])
print("hello" + " " + "world")
```

[ ]:
```
# For loops work over lists or generators, and are indented similarly to if␣
 ↪statements and functions
LST = [0,1,2,3,4,5]
for k in LST:
    print(k^2)
```

[ ]:
```
# The notation a..b can be used to build the list of numbers between integers a␣
 ↪and b
for k in (0..6):
    print(k^2)
```

[ ]:
```
# As in Python, Sage contains the function range(k), which encodes the numbers␣
 ↪0,1,...,k-1
# In Sage versions 8.x and lower, which rely on Python 2, range(k) returns the␣
 ↪list [0,1,...,k-1]
# In Sage 9.0 and higher, which use Python 3, range(k) returns an *iterator*␣
 ↪that can be converted to a list
# (think of an iterator as a method to construct elements of a list, one by␣
 ↪one, which can be more efficient)
# We assume in our discussions that the user of this document is using Sage 9.0␣
 ↪or higher
# More details on this change can be found here: https://docs.python.org/3.0/
 ↪whatsnew/3.0.html#views-and-iterators-instead-of-lists

print(range(5)) # printing an iterator just returns the iterator
print(list(range(5))) # the iterator can be converted to a regular list
```

```
for k in range(5): # loops can range directly over iterators, which can be more␣
 ↪efficient
    print(k^2)
```

[ ]:
```
# For loops can also be defined over one line
for k in range(5): print(k^2)
```

[ ]:
```
# There is a powerful alternate syntax for building lists using a
# function f(x) on the elements of a list LST: [f(k) for k in LST]
print([k^2 for k in range(5)])
print([cos(k*pi) for k in [1..5]])
```

[ ]:
```
# While loops are defined similarly to for loops
k = 0
while k<5:
    print(k^2)
    k = k+1
```

[ ]:
```
# While loops can be broken using 'break'
k = 0
while True:
    if k>= 5:
        break
    print(k^2)
    k = k+1
```

[ ]:
```
# The map operator applies a function to each element of a list
# Similar to range, in Sage 9.0+ map returns a "map object" / iterator
# The list command can be used to obtain an honest list from a map object
list(map(abs,[-1,2,-3,4]))
```

[ ]:
```
# User defined functions can also be mapped, where appropriate
def double(k):
    return 2*k

list(map(double,[-1,2,-3,4]))
```

[ ]:
```
# Can also use map with 'lambda expressions' to define a function in place
print(list(map(lambda t: t*2, [-1,2,-3,4])))
print(lambda t: t^2) # Defines the function f(x) = x^2 in place
```

[ ]:
```
# Can filter a list using 'filter'
# Similar to range and map, in Sage 9.0+ filter returns a "filter object" /␣
 ↪iterator
# The list function can be applied to obtain an honest list from a filter␣
 ↪object
```

```
print(list(filter(lambda t: t>0, [-1,2,-3,4])))
```

[ ]:
```
# Can also use the list 'comprehension form' to filter elements when building a list
[p^2 for p in [1..10] if is_prime(p)]
```

[ ]:
```
# Can sort lists, when appropriate.
# Sort is a function of a list (see Part 2 below for additional details)
L = [1,4,3,2,7,6]
L.sort() # Modifies the list L in place
print(L)
L = ["a","acb","abc","ab"]
L.sort() # Sort in lexicographical order
print(L)
```

[ ]:
```
# Lists are stored by reference. Simply writing L2 = L1, makes L1 and L2 point to the *same* list
# Use L2 = copy(L1) to make an independent copy
# Note copy only works at one level of lists
# (use the "deepcopy" command to copy a list of lists, and make everything independent)

L1 = [1,2,3]
L2 = L1 # L2 points to the same list as L1
L3 = copy(L1) # L3 points to a new list, initialized with the same values as L1
L2[0]=-1
print(L1); print(L2); print(L3)
```

1.0.2 Part 2: The Symbolic Ring and Sage Types

We now see how to manipulate symbolic variables and abstract functions, including basic calculus operations and plotting, and how to determine the type and parent of an object.

[ ]:
```
# Before running this section, we reset all variables
reset()
```

[ ]:
```
# By default, when opening a notebook the variable "x" can be used to define a symbolic function / expression
poly = x^2 - 1
print(poly)
```

[ ]:
```
# Using any other (undeclared) variable will give an error
# This behaviour can cause frustration for first-time Sage users
poly2 = y^2 - 1
```

[ ]:
```
# You can "undeclare" a variable with the restore command
restore(x)
show(x)
```

[ ]:
```
# If the variable x is assigned a different value, this does not change the␣
 ↪value of our symbolic expression!
# However, any new expressions containing x will use the updated value of x.
x = 2
print(poly)
poly2 = x^2 - 1
print(poly2)
```

MAKE SURE YOU UNDERSTAND THIS CRUCIAL POINT: This behaviour occurs because *Sage variables* (for instance, x on the left hand side of x = 2) are distinct from the underlying *symbolic variables* used to define symbolic expressions. By default the *Sage variable* x is initialized to a *symbolic variable* "x", and the expression poly above is defined in terms of this *symbolic variable*. Changing the *Sage variable* x to a new value does nothing to the underlying symbolic variable "x", which is why the value of poly does not change after setting x = 2.

[ ]:
```
# The easiest way to define a new symbolic variable having
# the same name as a Sage variable is with the "var" command
x = 2
print(x) # Prints the value of the Sage variable x
var('x') # Makes the Sage variable x point to the symbolic variable "x"
print(x) # Prints the value of the Sage variable x, which is now the symbolic␣
 ↪variable "x"
```

[ ]:
```
# Multiple variables can be defined at the same time
var('a b c') # Initialize Sage variables a, b, and c to symbolic variables "a",␣
 ↪"b", and "c"
poly2 = (a + b*c)^2
print(poly2)
```

[ ]:
```
# The commands "type" and "parent" illustrate the domains in which Sage objects␣
 ↪live.
# Symbolic expressions, defined with symbolic variables, live in the Symbolic␣
 ↪Ring.
# Sage automatically determines where objects defined with "=" should live.
var('a')
poly = a
print(type(poly))
print(parent(poly))
```

[ ]:
```
# Some additional examples
poly = 10
print(type(poly))
print(parent(poly))
```

```
print(type(1/2))
print(parent(1/2))
print(type(0.5))
print(parent(0.5))
```

[ ]:
```
# In Sage (as in Python) objects can have their own functions
# Type "poly2." (without quotes) then hit the *Tab* key to see the available␣
 ↪functions for poly2
# Run the command "poly2.subs?" to see the help page for the function poly2.
 ↪subs
# Run the command "poly2.subs??" to see the source code for the function poly2.
 ↪subs
var('a b c')
poly2 = (a + b*c)^2
print(poly2.subs(a=1,b=2,c=3))
print(poly2.subs)
print(parent(poly2.subs))
```

[ ]:
```
# Sage has many commands for manipulating expressions, such as simplify and␣
 ↪expand.
# Be careful -- in the Symbolic Ring, Sage simplifies without checking␣
 ↪restrictions on variables
var('x')
print(((x-1)*(x+1)/(x-1)).simplify())
print(simplify((x-1)*(x+1)/(x-1))) # simplify(p) is a built-in shortcut for␣
 ↪type(p).simplify()
```

[ ]:
```
# Expanding an expression
print(expand((x-1)*(x+1)))
```

[ ]:
```
# Factoring an expression
pol = x^2-1
print(pol.factor())
```

[ ]:
```
# Factoring a larger expression
f = x^23-x^16-2*x^13+3*x^20-2*x^10+2*x^11-2*x^4+2*x+5*x^3-5
show(f)
show(f.factor())
```

[ ]:
```
# Equations are also symbolic expressions, defined using "=="
eq1 = (x^3 == 1)
print(eq1)
print(eq1.lhs())
print(eq1.rhs())
```

[ ]:
```
# The solve command works with equations
show(solve(eq1, x))
```

[ ]:
```
# Symbolic functions can be defined with symbolic variables
f = sin(x)+2*cos(x)
print(f)
```

You can also define f via `f(x) = sin(x)+2*cos(x)`.

This format allows you to specify the order of the arguments if you want to call a multivariate function, e.g., `F(x, y, z) = sin(x) + 2*cos(y) + 3*z`.

[ ]:
```
# The find_root command can be used to approximate roots numerically
# (additional details on finding roots of polynomials are given in the next␣
 ↪section)
f(x).find_root(-10, 10)
```

[ ]:
```
# Symbolic functions can be plotted with the plot command
plot(f(x),x,0,10) # Syntax is "plot(function, variable, xmin value, xmax␣
 ↪value)"
```

[ ]:
```
# Plots can be "added" to overlap -- run "plot?" or "help(plot)" for plotting␣
 ↪options
p1 = plot(sin(x),x,0,pi)
p2 = plot(cos(x),x,0,pi,color="red")
p1+p2
```

[ ]:
```
# Common plot options include
# plot_points (default 200)
# xmin and xmax
# color
# detect_poles (vertical asymptotes)
# alpha (line transparency)
# thickness (line thickness)
# linestype (dotted with ':', dashdot with '-.', solid with '-')
# Use commands list p.set_aspect_ratio, etc.

pp = plot([]) # Define empty plot
cols = rainbow(20) # Define 20 evenly spaced colors
for k in range(20):
    pp = pp + plot(log(x+k),1,10,color=cols[k],thickness=2)

pp # Print superimposed graphs
```

[ ]:
```
# 3d plots are similar (user may need to load the 3D viewer after running this␣
 ↪code)
var('x y')
```

```
f(x,y) = x^2 + sin(x)*cos(y)
plot3d(f, (x,-1,1), (y,-1,1))
```

[ ]:
```
# The series command computes power series of symbolic expressions representing␣
 ↪functions
show(log(1/(1-x)).series(x,10))
```

[ ]:
```
# The series command can also compute Laurent series
show(tan(x).series(x==pi/2,10))
```

Some methods available to symbolic expressions may not be available for symbolic series as they are of the type "symbolic series" rather than "symbolic expressions".

[ ]:
```
# To go from a series expression to the polynomial defined by its terms, use␣
 ↪the truncate command
show(arctan(x).series(x,10))
show(arctan(x).series(x,10).truncate())
```

Sage can also compute derivates, integrals, and limits from Calculus such as:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sin x \qquad \int \sin(x)\cos(x)\,\mathrm{d}x \qquad \int_{-\infty}^{\infty} e^{-x^2}\,\mathrm{d}x \qquad \lim_{x\to 0}\frac{\sin x}{x}$$

[ ]:
```
# The diff/differentiate command computes derivatives
# The integral/integrate command is similar
show(diff(sin(x),x))
show(integrate(sin(x)*cos(x),x))
show(integrate(exp(-x^2),x,-infinity,infinity))
show(limit(sin(x)/x,x=0))
```

[ ]:
```
# This also works with abstract symbolic functions
var('x y')
function('f')(x)
show(diff(sin(f(x)),x,x))
function('g')(x,y)
show(diff(g(sin(x),f(x)),x))
```

[ ]:
```
# Here we set up and solve a differential equation
x = var('x')
y = function('y')(x)
desolve(y*diff(y,x) == x, y)
```

Sage can even find closed-form expressions of some interesting sums such as these:

$$\sum_{k=0}^{n} a^k \qquad \sum_{k=0}^{n}\binom{n}{k} \qquad \sum_{n=1}^{\infty}\frac{1}{n^2}$$

[ ]:
```
# Sage can calculate some symbolic sums
var('a k n')
show(sum(a^k,k,0,n))
show(sum(binomial(n,k), k, 0, n))
show(sum(1/n^2, n, 1, infinity))
```

[ ]:
```
# You can use the assume to command to put restrictions on variables
# This can be useful for infinite series
# Just running sum(a^k,k,0,infinity) throws a "ValueError" -- need to *assume*␣
 ↪|a|<1
assume(abs(a)<1)
show(sum(a^k,k,0,infinity))
```

[ ]:
```
# Assumptions stay with variables until cleared using forget
# Run this cell directly after the previous cell
print(bool(a<1)) # Test if a < 1, under assumption abs(a) < 1
forget(abs(a)<1)
print(bool(a<1)) # Test if a < 1, under no assumptions
```

[ ]:
```
##################################################
# EXERCISE: Make a function to compute the nth Fibonacci number (defined by␣
 ↪fib(n+2) = fib(n+1) + fib(n)
# and fib(0)=fib(1)=1). What is the highest number you can compute in 5 seconds?
#
# Adding the line "@cached_function" directly before the function definition␣
 ↪tells Sage to store the
# result of each function return, which will speed up the computation. Add this␣
 ↪line then see how high
# you can go in 5 seconds.
##################################################
```

[ ]:
```
##################################################
# EXERCISE: Compute the sum of the first 100 perfect squares.
# (Use the add function -- type "add?" or "help(add)" for its documentation)
##################################################
```

[ ]:
```
##################################################
# EXERCISE: Create a list containing the first 100 primes of the form 4*k+1
##################################################
```

[ ]:
```
##################################################
# EXERCISE: Guess the result of uncommenting and running the following code.
# Explain the output that does occur.
##################################################
# var('p q r'); r = q; q = p; p = r
# print(p); print(q); print(r)
```

[ ]:
```
##################################################
# EXERCISE: Uncomment and hit <Tab> with the cursor on the far right of the␣
 ↪following line see the
# available functions for poly2 which begin with "s". Select one of the␣
 ↪functions, look up its
# documentation, then run the function (with appropriate arguments if␣
 ↪necessary)
##################################################
# poly2.s
```