IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 23, 2017

1 INTRODUCTION TO python

1.1 USE OF NUMERICAL LIBRARIES

We will be using `python 2.7` as the programming language in this course, but will take advantage of previous programming experience with Matlab. For this reason, frequent reference will be made between the two programming languages. For most of the projects in this course, we will work with the numerical libraries `numpy` and `scipy` which contain a wide range of numerical functions and algorithms. To include these two libraries, we start our code with:

```
1  import numpy as np
2  import scipy as sp
```

We refer to any function from these two libraries by their abbreviated prefix `np` and `sp`, respectively.
Before developing python-programs, it is important to point out key differences between `python` and Matlab. Some of the are:

1. Having its origin in linear algebra applications, the basic data type in Matlab is a matrix. In `python`, the basic data type is an array; matrices can be defined as subclasses of arrays.

2. Matlab uses indexing starting with 1. `python` uses indexing starting with 0.

3. Array equivalence ($A = B$) in Matlab will produce an independent copy of B in A. The same command in `python` will produce a pointer from B to A; changing a value in B will change the associated value in A.

1.2 MATLAB VERSUS python: A COMPARISON OF COMMON COMMANDS

For most of our applications, we will deal with arrays from the numpy library. The list below gives a few of the equivalences between Matlab and python. Note that '#' is the prefix for adding comments.

```
# Matlab: a & b, a | b
  a and b, a or b
# Matlab: size(a)
  shape(a), a.shape
# Matlab: a(end)
  a[-1]
# Matlab: a(2,5)
  a[1,4]
# Matlab: a(1:5,:)
  a[0:5,:], a[:5,:]
# Matlab: a(1:3,5:9)
  a[0:3,4:9], a[0:3][:,4:9]
# Matlab: a(1:2:end,:)
  a[::2,:]
# Matlab: a', a.'
  a.conj().T, a.T
# Matlab: a*b, a.*b, a./b, a.^3
  a.dot(b), a*b, a/b, a**3
# Matlab: a=b, a=b(2,:), a=b(:)
  a=b.copy(), a=b[1,:].copy(), a=b.flatten()
# Matlab: a=eye(3), a=zeros(3,4), a=ones(3,4)
  a=np.eye(3), a=np.zeros((3,4)), a=np.ones((3,4))
# Matlab: [a b], [a; b]
  hstack((a,b)), vstack((a,b))
# Matlab: max(max(a)), max(a), max(a,[],2)
  a.max(), a.max(0), a.max(1)
```

1.3 VARIABLE TYPES

Care must be taken when dividing variables to avoid unintended integer division.

```
  4/5
  4./5
  4/5.
```

The result to the first line is zero, while the result to the second and third line is the (probably) intended 0.8. Similarly,

```
  9/5
  9./5
```

```
3  9/5.
4  float(9)/5
```

produces 1, 1.8, 1.8, 1.8, respectively.
To check the type of a variable a, we simply use

```
1  type(a)
```

Complex numbers are dealt with as follows

```
1  a = 1.5 + 4j
2  b = complex(1.5,4)
3  a.real
4  a.imag
5  a.conjugate()
```

1.4 Loops

Loops are a quintessential element of any programming language, allowing and controlling the repeated execution of code segments. We distinguish **for**-loops, where the number of repetition is prescribed, and **while**-loops, where a loop is executed as long as a logical condition is satisfied. The `python` syntax follows closely the Matlab syntax, with the exception that the **end**-statement is missing in `python` and is replaced by indentation. For a **while**-loop we have

```
1  while a <= 10:
2      c = np.sin(b)
3      d = np.cos(c)
4      a = a-d
5  print d
```

In the above code segment, we execute the indented lines, as along as the condition ($a \leq 10$) is satisfied. The print statement is executed after the **while**-loop terminates. The 4th line is often replaced with an abbreviation for expressions that adjust a variable; we have

```
1  a += b   # same as:  a = a+b
2  a -= b   # same as:  a = a-b
3  a *= b   # same as:  a = a*b
4  a /= b   # same as:  a = a/b
```

For Boolean expressions, we have

```
1  a == b   # a equal b
2  a != b   # a not equal b
3  a > b    # a greater than b
4  a <= b   # a less than or equal b
5  not(a<b) # a not less than b
```

Before talking about for-loops, we need to consider a variable type in `python` referred to as lists. They consist of a sequence of numbers and are defined by the operations that act on them. It is important to distinguish lists from arrays (this distinction is absent in Matlab). Lists are defined by square brackets, with each element separated by a comma. For example,

```
L = [10, 11, 12, 13, 14]
```

Various operations are defined on lists; below are the most common ones.

```
L.append(15)     # append 15 at the end of the list
L.insert(0,9)    # insert 9 at the first position of the list
L += [16,17,18] # add two lists together
del L[5]         # delete the sixth element in the list
len(L)           # length of the list (# of elements)
L.index(12)      # find the index of element 12
L = range(0,10) # create a list from 0 to 9 (!)
```

We can now state the format for a **for**-loop. We have

```
L = [10, 11, 12, 13, 14, 15]
for a in L:
    c = np.sin(a)
    d = np.cos(c)
print d
```

The first line specifies a list of elements. The **for**-loop then scans through the elements of the list L. Again, indentation determines the body of the loop (i.e., the print-statement is executed after the loop terminates).
For the special case of a for-loop over integers, we have the construction

```
for i in range(start, stop, step):
    ...
```

but keep in mind that the element 'stop' is not part of the list. For example,

```
for i in range(1,9,2):
    ...
```

will run over $i = 1,3,5,7$ but not 9.
Many times we want to create a new list by traversing an old list. For this special task there is an inline-abbreviation in `python`

```
Lnew = [np.cos(np.sin(c)) for c in L]
```

1.5 Decision making

It is often the case that we have to execute certain parts of a code, depending on the value of a variable. In this case, we have to introduce different branches in the code that will be

run based on a Boolean expression. A construct that accomplishes this is the **if**-statement. In python, it is constructed as follows

```
if x >= 0:
    y = x**2
elif (x < -1):
    y = (x+1)**2
else:
    y = 0
print x,y
```

Note again that indentation of the various branches avoids the use of an **end**-statement. There can be more than one **elif**-statement.
For short and simple **if**-statements, python allows an inline **if**-statement of the form

```
y = (np.sin(x) if (x>=0) else np.cos(x))
```

1.6 FUNCTIONS

Functions are used in programming to relegate a specific task to a subunit of the code. They are used to keep the code structure orderly and transparent. Functions execute a set of commands that, in the remainder of the code, is used (perhaps with different parameters) repeatedly.
The python syntax for functions follows the example

```
def myfunc(x,y):
    z1 = np.sin(x)*np.exp(-y)
    z2 = np.cos(x)*np.exp(-y*y)
    return z1,z2
```

In this case, the function has two input arguments (x, y) and returns two output values (z_1, z_2). Calling this function would involve

```
a = 1.
b = -3.
v1,v2 = myfunc(a,b)
```

For functions with multiple and optional input values, we can specify variable names and assign to them default values that will be used in the function if the function is called without a specific argument. For example, we can have

```
def myfunc(x,y=1.):
    z1 = np.sin(x)*np.exp(-y)
    z2 = np.cos(x)*np.exp(-y*y)
    return z1,z2
```

and call the function

```
v1,v2 = myfunc(0.5)
```

in which case only the variable x is specified as $x = 0.5$, while y takes on its default value of $y = 1$.

1.7 Exercise

We want to write a set of subroutine/functions that convert a list of temperatures in Fahrenheit to an equivalent list in Celsius and vice versa.
The following code accomplishes this.

```
import numpy as np
import scipy as sp

def F2C(F):
    # conversion from Fahrenheit to Celsius
    C = []      # start with empty list
    for degF in F:
        degC = 5.*(degF-32.)/9.
        C.append(degC)   # append new value to list
    return C

def C2F(C):
    # conversion from Celsius to Fahrenheit
    F = []      # start with empty list
    for degC in C:
        degF = 9.*degC/5. + 32.
        F.append(degF)   # append new value to list
    return F

if __name__ == '__main__':
    # main code

    F  = range(50,200,10)
    C  = F2C(F)
    FF = C2F(C)

    # doing it 'inline'
    F_inl = [cdeg*9./5.+32. for cdeg in C]
    C_inl = [5.*(fdeg-32.)/9. for fdeg in F]
```

The code is stored in a file called `FCconversion.py` and can be called in `python` by using

```
run FCconversion.py
```

The above file `FCconversion.py` is treated as a module and can be imported into other programs, just as we import the libraries `numpy` and `scipy`. For example, in another `python`-code we can write one of the following commands

```
1  import FCconversion
2  from FCconversion import *
3  import FCconversion as FC
```

and then have access to all functions in `FCconversion.py`.
The piece at the end, starting with the **if**-statement (line 20), contains the main-code or driver routine and is invoked when typing the 'run FCconversion.py' command.

Question: The two temperature scales cross at -40 degrees, i.e., $-40°F = -40°C$. Verify the code using this result.

Question: Run the code `FCconversion.py` and compute the maximum difference between the two lists C (line 24) and C_{inl} (line 29).

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 22, 2017

1 PATTERN SEARCHING (RABIN-KARP ALGORITHM)

Pattern searching is an important discipline of computer science and applied mathematics. It is concerned with the extraction of a given pattern p of size m from a (commonly very large) text t of length n. The pattern and text can consist of numbers, letters or other distinct structures or tokens. Typical applications of pattern searching are encountered in plagiarism detection software or in the sequencing of genetic material.
Here we will consider sequential pattern searching, where a large string of text is searched for occurrences of shorter patterns. For example, in a long sequence of nucleotides within a DNA molecule we may be interested in searching for locations and frequency of a short pattern made up of the DNA alphabet $\{A, C, G, T\}$ which denotes the four building blocks, i.e., the nucleic acids **A**denine, **C**ytosine, **G**uanine and **T**hymine.

1.1 NAIVE SEARCH

A naive approach to pattern searching is the sequential comparison of the pattern to text snippets of equal length, then sliding along the entire text sequence. While this naive pattern searching algorithm often works better than expected, it exhibits its worst-case performance (and run times) when looking for rather long patterns in very long search strings. With a text of length n and a pattern of length m, we have to make a comparison of m entries in the pattern and substring for each of the positions in the text t. The run time of the algorithm thus scales as $\mathcal{O}(nm)$.
An implementation of a naive search algorithm is given below, written as a python function.

```
1 def naive_search (string,pattern):
```

```
    # naive string matching algorithm
    n  = len(string)
    m  = len(pattern)
    ic = 0
    # element-by-element matching
    for i in range(n-m+1):
        # call the (logical) match function
        if match(string,pattern,i):
            ic += 1
            print i
    print "number of matches: ", ic
```

This function uses another python function `match` that we will use later in the more advanced Rabin-Karp algorithm (see below). It is given by

```
def match(string,pattern,i):
    # element-by-element matching of a pattern
    # in a string; i: position in string
    for k in range(len(pattern)):
        if pattern[k] == string[i+k]:
            pass
        else:
            return False
    return True
```

1.2 Speeding up the naive-search algorithm by hashing

One possibility of speeding up the naive-search algorithm is by appling more sophisticated manners of traversing and skipping through the text, rather than the slow sliding-window approach. Another and alternative possibility is the speed-up of the matching procedure. This latter technique is what the Rabin-Karp algorithm proposes. The Rabin-Karp algorithm, named after Michael Rabin and Richard Karp, speeds up the matching section of the naive-search algorithm by introducing **hash functions**.
In general, a hash function is a mapping of a string or number sequence onto a numerical value, the **hash value**. Based on this hash function, the search pattern and a string segment match when their respective hash values match. This hash value is akin to the familiar checksum, when comparing two data files. The Rabin-Karp algorithm thus replaces a direct comparison of a given pattern and an extracted string segment by a comparison of their associated hash values. In other words, in the text sequence we search for substrings that have the same hash value as the pattern's hash value.
When introducing hash functions into the naive-search algorithm, two issues arise.
First, the mapping of a pattern onto a hash value may produce false positives, i.e., the hash values of the pattern and the text substring coincide, even though the actual substrings do not match. This is also referred to as **hash collision**. It is a consequence of the fact that

different text-substrings can map to the same hash value. While this non-uniqueness does not pose a problem, we simply need to compare the pattern and the real text segment after we receive a flag from the matching hash values. In other words, hash value matching does a fast preprocessing step, before direct comparison confirms or rejects the potential match. Since a direct comparison is relatively costly, we have to choose a proper hash function that does not produce too many false positivesm but rather keeps direct comparisons a rare event.
Second, the computing of the hash value has to be efficient, otherwise there will not be any savings over a direct comparison. Ideally, the comparison should not scale with the pattern size, but instead be constant in time. In this latter case, the run time $\mathcal{O}(nm)$ of the naive search would reduce to $\mathcal{O}(n)$, which is as good as one can expect from a sequential search algorithm.
With the hash function introduced, the Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each m-character subsequence of the text. If the hash values are unequal, the algorithm will calculate the hash value for the next m-character sequence. If the hash values are equal, the algorithm will compare the pattern and the m-character sequence directly. In this way, there is only one hash comparison per text subsequence, and character matching is only needed when hash values match.

1.3 A BIT OF MATHEMATICS

Consider an m-character sequence as an m-digit number expressed in base b, where b is at least the number of letters in the alphabet used in our text t. The text subsequence $t[i,\ldots,i+m-1]$ can then be mapped to the number x_i according to

$$x_i = t[i]\cdot b^{m-1} + t[i+1]\cdot b^{m-2} + \cdots + t[i+m-1]. \tag{1.1}$$

This expression already acts as a hash function, assigning a number x_i to a string $t[i,\ldots,i+m-1]$. If the text to be processed consists of letters (such as in plagiarism detection, or in DNA pattern searching), an additional mapping from the letters to the $t[i]$'s has to be introduced (see the example below). The above way of expressing a text sequence as a number using a polynomial in b is known as the **Rabin fingerprint**. It has the considerable advantage that the subsequent hash value x_{i+1} of the right-shifted m-digit subsequence $t[i+1,\ldots,i+m]$ can be computed in constant time. This can be easily shown by

$$\begin{aligned}
x_{i+1} &= t[i+1]\cdot b^{m-1} + t[i+2]\cdot b^{m-2} + \ldots + t[i+m], && \text{(1.2a)}\\
&= x_i \cdot b \qquad \text{(shift left one digit)} && \text{(1.2b)}\\
&\quad - t[i]\cdot b^m \qquad \text{(subtract leftmost digit)} && \text{(1.2c)}\\
&\quad + t[i+m]. \qquad \text{(add new rightmost digit)} && \text{(1.2d)}
\end{aligned}$$

In this manner, we avoid the explicit computation of a new hash value; rather, we simply adjust the current hash value as we shift our substring in the text by one character to the right. Hash functions that allow this type of shortcut are referred to as **rolling hash** functions; there are many others besides the Rabin fingerprint.

In the above expression, there still remains one problem. If the size m of the pattern gets large and the alphabet of the text is rather rich (and thus requires a large base b), the value of b^m that appears in the expression above will become exceedingly high. To counteract this trend, we limit the size of the hash value by introducing the modulo function, i.e., we clip the hash values to a finite number range. More specifically, we compute our hash values by taking the modulo (with an appropriate number q) of each component in the above expression.
The modulo function $\mod$ is particularly convenient since it satisfies the mathematical expressions

$$[(x \mod q) + (y \mod q)] \mod q = (x+y) \mod q, \tag{1.3a}$$
$$(x \mod q) \mod q = x \mod q. \tag{1.3b}$$

Applied to our rolling hash expression, and introducing the notation $h_i = x_i \mod q$, we obtain

$$\begin{aligned} h_{i+1} &= \left(t[i+1]\cdot b^{m-1} \mod q + t[i+2]\cdot b^{m-2} \mod q + \ldots + t[i+m] \mod q\right) \mod q \\ &= (h_i \cdot b \mod q \quad \text{(shift left one digit)} && (1.4a)\\ &\quad - t[i]\cdot b^m \mod q \quad \text{(subtract leftmost digit)} && (1.4b)\\ &\quad + t[i+m] \mod q) \mod q \quad \text{(add new rightmost digit)} && (1.4c) \end{aligned}$$

This latter expression is at the heart of the Rabin-Karp algorithm.
It leaves the final questin of what value to choose for q. A small number will increase the frequency of hash collisions, the necessity to perform an excessive number of direct comparisons and, consequently, a serious degradation in performance. For example, when choosing $q = 3$, we only produce hash values of $h_i = 0, 1, 2$, which may be far too limited to properly represent all the different text substrings. Instead, a large prime number is commonly chosen for q.

1.4 Algorithm

With this we can state the Rabin-Karp algorithm.
The Rabin-Karp pattern search algorithm computes a hash value for a search pattern p of length m and for each m-character substring of a text t. The two hash values are compared, rather than the actual strings. Only after the two values match does the algorithm compare the two patterns (search pattern and substring) to confirm or reject a match, thus resolving a hash collision. If no match is found, the algorithm shifts to the next substring of t. When a rolling hash is used, this latter shift can be computed very efficiently.
We have the following python functions, implementing the Rabin-Karp string search algorithm.

```
1 import sys
2 import time
```

```
3
4  def RabinKarp(str,pat,base=256,modu=16647133):
5      # Rabin-Karp string matching algorithm
6      #    str:  text to be searched
7      #    pat:  pattern to match
8      #    base: base of the Rabin-fingerprint
9      #    modu: modulus of the hash-function
10     ic = 0
11     n  = len(str)
12     m  = len(pat)
13     # compute the hash values for pattern
14     # and initial string (Horner's scheme)
15     hp = 0
16     for i in pat:
17         hp = (base*hp + ord(i))%modu
18     hs = 0
19     for i in str[:m]:
20         hs = (base*hs + ord(i))%modu
21     # check for initial match
22     if hs == hp and match(str[:m],pat):
23         print 0
24         ic = 1
25     # compute b^m
26     bm = 1
27     for i in range(m-1):
28         bm = (bm*base)%modu
29     # main loop
30     for i in range(1, n - m + 1):
31         hs = (base*(hs - bm*ord(str[i-1])) + \
32               ord(str[i+m-1]))%modu
33         if (hs == hp):
34             if match(str[i:i+m],pat):
35                 print i,
36                 ic += 1
37     print "\n number of matches found: ", ic
```

Finally, the main code contains material that applies the Rabin-Karp algorithm and the naive search algorithm to two applications: a data-base of genetic material from the DNA of the fruit fly (*Drosophila melanogaster*) which contains more than 23 million nucleotide, and the first 100000 digits of π.

```
if __name__ == '__main__':

    # read 'Genetic Material' file
    f     = open('GeneticMaterialLarge','r')
    strGM = f.read()
    f.close()
    strGM = strGM.replace("\n","")

    # patterns to search for
    p1  = 'CACAATATATGATCGC'
    p2  = 'GTGCCAACATATTGTGCTCTCTATATAATGACTGCCTCT'
    p3  = 'GCGAATATGGAAAAAAAGGCAGACACACACTTGTTGATTTCTTTATGCG'
    pa  = 'ATTTCTACTGTAAAAATCTTTCTTTGTCCAAAAAATCTTTGATAGTGTTT'
    pb  = 'CCCTATGTTTGTGTGCACGGACACGTGCTAAAGCCAAATATTTTGCTAGT'
    pc  = 'CAATATATAACAAAAAATATTTTTCGGGCTTGCGTACTGTAAAGATGAAT'
    p4  = pa + pb + pc
    patGM = p1

    # read 'Digits of Pi' file
    f     = open('DigitsOfPi','r')
    strPi = f.read()
    f.close()

    # pattern to search for
    p1 = '8888'
    p2 = '2479114016957900338356'
    patPi = p1

    # scan the Genetic Material file
    t0    = time.time()
    RabinKarp(strGM,patGM)
    t1    = time.time()
    total = t1-t0
    print 'Genetic Material (Rabin-Karp algorithm): ',total

    t0    = time.time()
    naiveSearch(strGM,patGM)
    t1    = time.time()
    total = t1-t0
    print 'Genetic Material (naive search): ',total

    # scan the Digits of Pi file
    t0    = time.time()
```

```
44      RabinKarp(strPi,patPi)
45      t1    = time.time()
46      total = t1-t0
47      print 'Digits of Pi (Rabin-Karp algorithm): ',total
48
49      t0    = time.time()
50      naiveSearch(strPi,patPi)
51      t1    = time.time()
52      total = t1-t0
53      print 'Digits of Pi (naive search): ',total
```

The main code includes a time comparison between the naive search and the Rabin-Karp algorithm. Even though the difference may not appear this drastic, one has to keep in mind that the algorithm is commonly applied to far larger data-bases and to larger patterns. In this case, the difference in run-time between naive searching and Rabin-Karp will be more pronounced.

1.5 Improvements and applications

The Rabin-Karp algorithm has been presented in its most basic form. However, even in this rudimentary form it is commonly applied in bioinformatics (when searching for patterns in nucleotides), text processing (such as plagiarism detection), music prototyping (for search engines) or data compression.
Nonetheless, there are improvements to the algorithms that increase its efficiency and applicability to a wide range of data. One of the most common is the use of prefix and suffix trees. Prefix and suffix strings are smaller substrings attached in front or at the end of a string that allow the early dismissal of mismatching strings and the skipping ahead by more than one record. Taking advantage of prefix and suffix strings can produce substantial speedup in the overall search for patterns.
The Rabin-Karp algorithm can quite easily be extended to multiple-pattern searches. In this case, we are looking for occurrences in the form of a vector-valued shift (s_1, s_2) (in the case of two patterns). Facial recognition and medical imaging are areas where two-dimensional pattern searches are commonly encountered.
Another extension of the Rabin-Karp algorithm involves the occurrence of wild cards, which are taken as matching *any* pattern. This situation is often encountered when dealing with DNA transcription factors. These factors need to be ignored (labelled as wild cards) when searching for valid patterns.

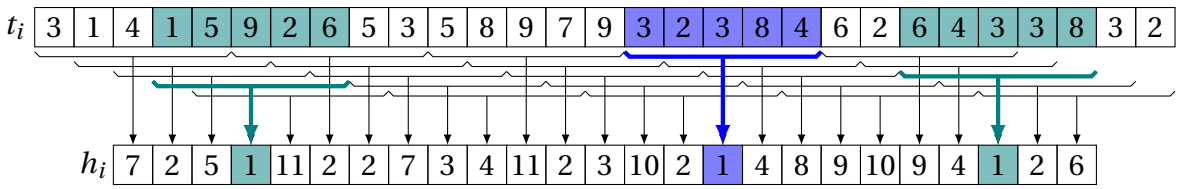

Figure 1.1: Demonstration of the Rabin-Karp algorithm: search for the string '32384' in the digits of π. We employ the algorithm with radix 10 (i.e., a representation of the digits in base 10) and a modulo 13 function. The hash value of the pattern is 1. We recognize the string in position 16 in the text (highlighted in blue), but we also encounter two hash collisions (in green) in positions 4 and 23, where the hash value matches, but not the actual substring.

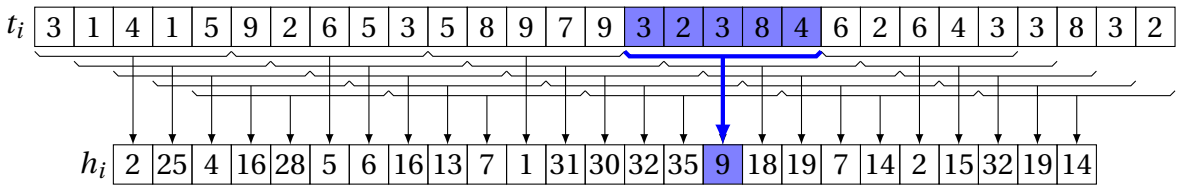

Figure 1.2: Demonstration of the Rabin-Karp algorithm: search for the same string '32384' in the digits of π. This time, we choose a modulo 37 function. The hash value of the pattern is 9, and we do not encounter hash collisions.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 31, 2017

1 SHORTEST PATH IN GRAPHS (DIJKSTRA ALGORITHM)

A graph is a mathematical construct consisting of a set of vertices that are connected by a set of edges. The edges can contain directional information, i.e. pointing from vertex to vertex, or just simply connecting two vertices. In former case, we refer to the graph as a directed graph; in the latter case, the graph is undirected. Often, the edges in an undirected or directed graph carry a weight which, in applications, is related to distance, capacity or cost of travel.

Graphs appear in many applications, such as traffic flow through cities, telephone or power networks, or work flow charts in manufacturing, to name but a few.

In complex graphs we are often interested in the path through the graph that (i) connects two specified vertices and (ii) has a minimum total weight. For example, in planning a route through a city from point A to point B using a GPS device, we are interested in a set of connected edges that start at A and end at B such that the total sum of the edge weights is as small as possible. The edge weight in this case could be the length of the edge (yielding the shortest path from A to B) or the travel time along the edge (resulting in the fastest path from A to B). The algorithm that efficiently determines the minimal-cost path in a graph is Dijkstra's algorithm. It was conceived and formulated in 1956 by Edsger Dijkstra, a Dutch computer scientist. In its original form it can compute the shortest path from a point A in the graph to a point B, or, more generally, it can calculate the shortest path from a point A to any other vertex in the graph, thus generating a shortest-graph tree structure.

1.1 ALGORITHM

For computing the shortest distance through a graph, we first have to decide on a starting place, i.e., an initial or source node. Dijkstra's algorithm can then be stated in the following three steps.

1. We set the distance to the initial node to zero, and the distance to all other nodes to infinity. Furthermore, we set the initial node to the `current` node and label all other nodes as `unvisited`.

2. From the `current` node, determine all its `unvisited` neighbors. Then compute the distance to all these `unvisited` neighbors by adding the distance to the `current` node to the distances to all `unvisited` neighbors. Only update the distances to the `unvisited` neighbors, if they are smaller than their present values (otherwise, keep the present value).

3. Select from the `unvisited` nodes the one with the smallest distance, record its parent node as the `current` node. Then remove it from the `unvisited` list and select it as the new `current` node. Go to step 2.

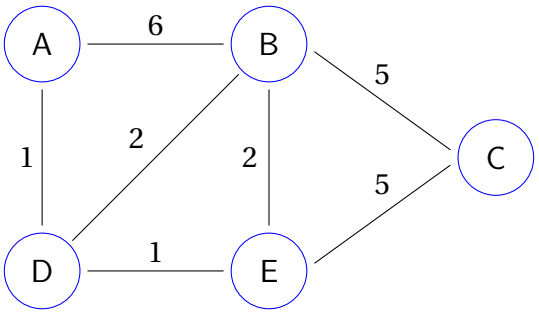

Figure 1.1: Example of a simple graph.

This algorithm terminates when no `unvisited` nodes can be found. The above algorithm is best explained with the help of an example (see figure 1.1). It is based on a simple graph with five nodes and seven edges. Based on the connectivity of the graph, we can formulate a **connectivity** or **adjacency matrix** as follows

$$\mathsf{W} = \begin{pmatrix} 0 & 6 & 0 & 1 & 0 \\ 6 & 0 & 5 & 2 & 2 \\ 0 & 5 & 0 & 0 & 5 \\ 1 & 2 & 0 & 0 & 1 \\ 0 & 2 & 5 & 1 & 0 \end{pmatrix} \tag{1.1}$$

with $\mathsf{W} = \{w_{ij}\}$, where w_{ij} denotes the weight of the edge between node i and node j. The ordering of the matrix rows and columns is following the sequence ABCDE. Furthermore, we take a value of $w_{ij,i\neq j} = 0$, if the node i is not connected to node j. We wish to determine the shortest path from node A to node C.

The tableau below (see figure 1.2) outlines the steps of Dijkstra's algorithm to compute the shortest path for the graph shown in figure 1.1.

`step 0:` We take A as our `current` node and set $d[A] = 0$ and $d[B], d[C], d[D], d[E] = \infty$.

`step 1:` We then determine the `unvisited` neighbors of A : the two nodes B and D. The distance to these nodes is the distance to the `current` node A, i.e., $d[A] = 0$, plus the respective weights of the edges from A to B and D. We update the distance of these two nodes as $d[B] = 6$ and $d[D] = 1$. As we update the distance to these two nodes, we also store the parent-node information for the two nodes, marked by the subscript $_A$, which indicates that the path to B and D comes from A. We proceed by choosing the `unvisited` node with the minimum distance, in our case, node D (see the teal square), as the new `current` node and remove it from the `unvisited` list.

	$d[A]$	$d[B]$	$d[C]$	$d[D]$	$d[E]$
`step 0`	0	∞	∞	∞	∞
`step 1`	0	6_A	∞	1_A	∞
`step 2`	0	3_D	∞	1_A	2_D
`step 3`	0	3_D	7_E	1_A	2_D
`step 4`	0	3_D	7_E	1_A	2_D

Figure 1.2: Dijkstra tableau

`step 2:` We repeat the same process with node D as the `current` node. Its `unvisited` neighbors are nodes B and E. The distances to these nodes are computed as 3 and 2, respectively, simply by adding the associated edge values 2 and 1 to the distance of the `current` node D. The smaller of the two distances corresponds to node E, which will become our next `current` node (teal square).

`step 3:` With E as the new `current` node, its `unvisited` neighbors consist only of node C. The distance to C is the sum of the `current` node's distance ($d[E] = 2$) and the edge weight from E to C, which is 5. The distance to C is thus 7. We denote by subscript $_E$ the parent node E. With E the only `unvisited` neighboring node, we set the `current` node to E.

`step 4:` A final application of a Dijkstra step does not produce any `unvisited` neighboring nodes, and the algorithm terminates.

The final line in figure 1.2 contains the information about shortest-distance paths from node A to all other nodes in the graph: node B can be reached at a cost of 3, node C at a cost of 7, node D at a cost of 1, and node E can be reached with a cost of 2. The stored subscripts allow us to reconstruct the shortest path. Starting at C, the cost 7_E identifies node E as the parent node. The parent node of E is node D whose parent node is A, our

starting node. The shortest path from A to C thus follows: A → D → E → C. See the red path in figure 1.2 and the critical path (in red) in figure 1.3.

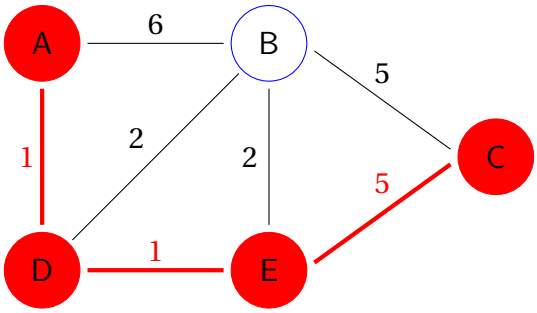

Figure 1.3: graph2

1.2 Implementation

The following python function implements the Dijkstra algorithm. The graph is defined by the weight matrix `wei` which contains its adjacency structure (given by the non-zero elements) and the cost associated with travelling along edges. The `Dijkst`-function specifies the starting (`ist`) and ending (`isp`) node in the network and produces the shortest path through the graph. The function uses an `UnVisited`-array which is 1 for `unvisited` nodes and 0 for visited ones. Line 26 produces a list of neighboring nodes, extracted from the `current` row of the weight matrix; by eliminating the zero elements from this list, only the `unvisited` neighbors remain (see line 27).

```
1  import numpy as np
2  import scipy as sp
3  import math as ma
4  import sys
5  import time
6
7  def Dijkst(ist,isp,wei):
8      # Dijkstra algorithm for shortest path in a graph
9      #    ist: index of starting node
10     #    isp: index of stopping node
11     #    wei: weight matrix
12
13     # exception handling (start = stop)
14     if (ist == isp):
15         shpath = [ist]
16         return shpath
17
18     # initialization
```

```
19      N         =  len(wei)
20      Inf       =  sys.maxint
21      UnVisited =  np.ones(N,int)
22      cost      =  np.ones(N)*1.e6
23      par       = -np.ones(N,int)*Inf
24
25      # set the source point and get its (unvisited) neighbors
26      jj            = ist
27      cost[jj]      = 0
28      UnVisited[jj] = 0
29      tmp           = UnVisited*wei[jj,:]
30      ineigh        = np.array(tmp.nonzero()).flatten()
31      L             = np.array(UnVisited.nonzero()).flatten().size
32
33      # start Dijkstra algorithm
34      while (L != 0):
35          # step 1: update cost of unvisited neighbors,
36          #         compare and (maybe) update
37          for k in ineigh:
38              newcost = cost[jj] + wei[jj,k]
39              if ( newcost < cost[k] ):
40                  cost[k] = newcost
41                  par[k]  = jj
42
43          # step 2: determine minimum-cost point among UnVisited
44          #         vertices and make this point the new point
45          icnsdr     = np.array(UnVisited.nonzero()).flatten()
46          cmin,icmin = cost[icnsdr].min(0),cost[icnsdr].argmin(0)
47          jj         = icnsdr[icmin]
48
49          # step 3: update "visited"-status and determine neighbors of new
50          UnVisited[jj] = 0
51          tmp           = UnVisited*wei[jj,:]
52          ineigh        = np.array(tmp.nonzero()).flatten()
53          L             = np.array(UnVisited.nonzero()).flatten().size
54
55      # determine the shortest path
56      shpath = [isp]
57      while par[isp] != ist:
58          shpath.append(par[isp])
59          isp = par[isp]
60      shpath.append(ist)
61
62      return shpath[::-1]
```

The main code

```
if __name__ == '__main__':

    # starting and stopping node
    ist = 4
    isp = 3

    # adjacency matrix
    wei = np.array([[ 0, 20,  0, 80, 0,  0, 90,  0],
                    [ 0,  0,  0,  0, 0, 10,  0,  0],
                    [ 0,  0,  0, 10, 0, 50,  0, 20],
                    [ 0,  0, 10,  0, 0,  0, 20,  0],
                    [ 0, 50,  0,  0, 0,  0, 30,  0],
                    [ 0,  0, 10, 40, 0,  0,  0,  0],
                    [20,  0,  0,  0, 0,  0,  0,  0],
                    [ 0,  0,  0,  0, 0,  0,  0,  0]])

    shpath = Dijkst(ist,isp,wei)
    print ist,' -> ',isp,' is ',shpath
```

1.3 Example

As a more realistic example, we wish to find the shortest and fastest path through a network of streets in a large city. This is a task that is routinely solved by a navigational GPS system in your car (although with a slightly different algorithm).
Figure 1.4 shows a city map of Rome, with some of the city's attractions marked. In addition, we have "discretized" some of the main streets by introducing vertices and connectig them by edges; a total of 58 nodes have been added. The edges between the nodes carry a weight that includes their distance as well as the speed limit: in this manner, we can compute the shortest path (using distance as a weight) or the fastest path (using the time it takes to traverse an edge as a weight) through the city. We select St. Peter's Square (node 13 in the Northwest of the city) as our starting point and the Coliseum (node 52 in the Southeast of the city) as our destination. Most streets (edges) can be traversed in both directions, although not necessarily at the same speed; some streets, however, are one-way streets and can be traversed only in one direction (e.g. the two main roads along the river are one-way streets with $3-5-8-\ldots-53$ on the Western bank going mostly North-South and $54-49-47-\ldots-2$ on the Eastern bank going mostly South-North).
Dijkstra's algorithm can be readily applied to the network, once the adjacency matrix has been established from the coordinate and speed-limit information. An additional function, producing the weight matrix from this information, is necessary (see below).

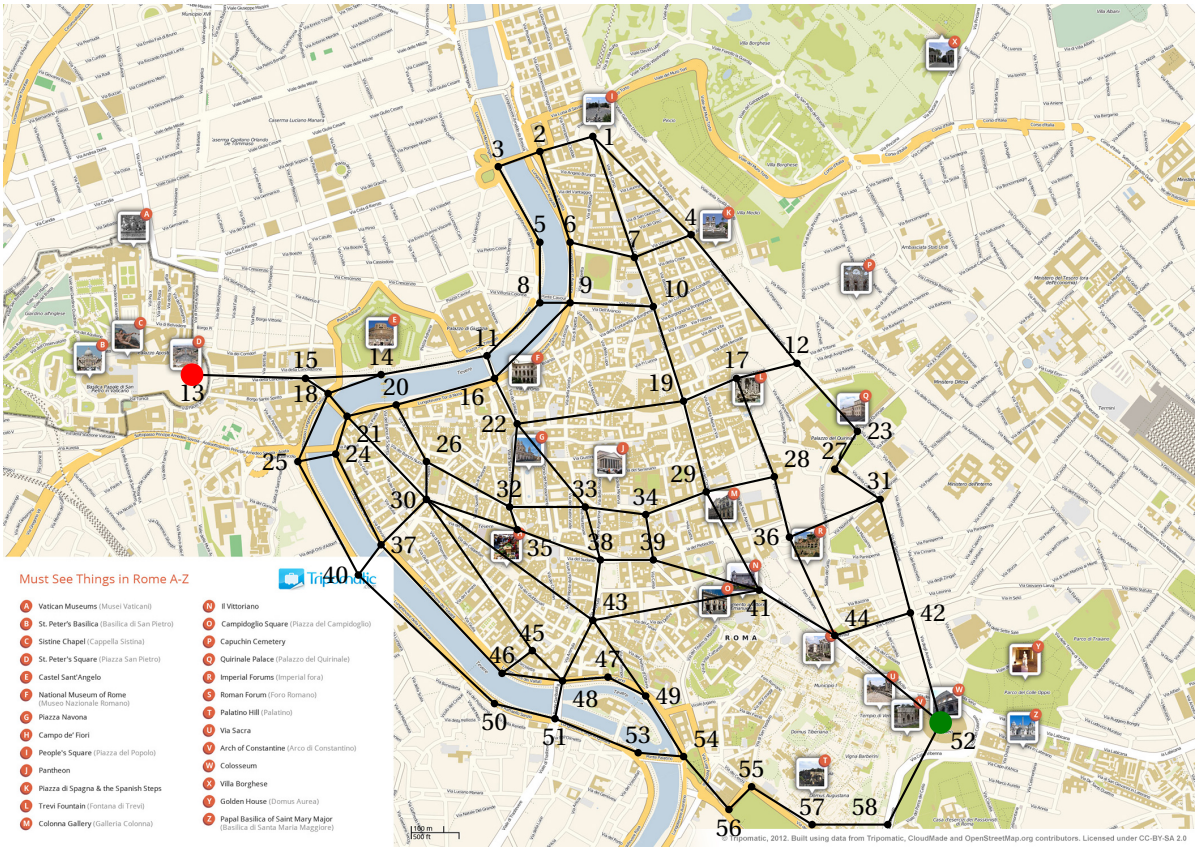

Figure 1.4: Map of Rome with a simplified traffic network consisting of 58 vertices and 156 edges (counting bi-directional edges double). What is the shortest or fastest path from node 13 (red symbol: St. Peter's Square) to node 52 (green symbol: Coliseum)?

```
def calcWei(RX,RY,RA,RB,RV):
    # calculate the weight matrix between the points

    n    = len(RX)
    wei = np.zeros((n,n),dtype=float)
    m    = len(RA)
    for i in range(m):
        xa = RX[RA[i]-1]
        ya = RY[RA[i]-1]
        xb = RX[RB[i]-1]
        yb = RY[RB[i]-1]
        dd = ma.sqrt((xb-xa)**2 + (yb-ya)**2)
        tt = dd/RV[i]
        wei[RA[i]-1,RB[i]-1] = tt
    return wei
```

The main code then reads in the geometric information (coordinate of vertices, `RomeX` and `RomeY`, and connectivity, `RomeA` and `RomeB`) from file; this information would be stored in form of maps on your GPS device. Additional information about the speed on each edge (`RomeV`) would come from established speed limits or interactively from current traffic information (reporting accidents, traffic jams, temporary detours or closures).

```
if __name__ == '__main__':

    # EXAMPLE 2 (path through Rome)
    RomeX = np.empty(0,dtype=float)
    RomeY = np.empty(0,dtype=float)
    with open('RomeVertices','r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeX = np.concatenate((RomeX,[float(row[1])]))
            RomeY = np.concatenate((RomeY,[float(row[2])]))
    file.close()

    RomeA = np.empty(0,dtype=int)
    RomeB = np.empty(0,dtype=int)
    RomeV = np.empty(0,dtype=float)
    with open('RomeEdges','r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeA = np.concatenate((RomeA,[int(row[0])]))
            RomeB = np.concatenate((RomeB,[int(row[1])]))
            RomeV = np.concatenate((RomeV,[float(row[2])]))
```

```
22      file.close()
23
24      wei = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV)
25
26      ist = 12 # St. Peter's Square
27      isp = 51 # Coliseum
28
29      # use the Dijkstra algorithm
30      t0 = time.time()
31      shpath = Dijkst(12,51,wei)
32      t1 = time.time()
33      print 'Dijkstra:     ',ist+1,' -> ',isp+1,' is ',np.array(shpath)+1,t
```

This exercise also demonstrates the efficiency of Dijkstra's algorithm: the above task of finding the optimal path through a simplified traffic network can be solved on a standard laptop computer in about a millisecond.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 31, 2017

1 SHORTEST PATH IN GRAPHS (BELLMAN-FORD ALGORITHM)

Dijkstra's algorithm is commonly used in finding the shortest path in weighted graphs. However, the weight assigned to edges, connecting nodes, have to be positive for the algorithm to converge. In many applications, the positivity of the edge values cannot be guaranteed. For example, when modeling the workflow through graphs from manufactoring, resource management or supply chains, certain parts of a path can imply a net gain, rather than a net cost. Dijkstra's algorithm fails in this case. Instead, an alternative algorithm – the Bellman-Ford algorithm – can be used. In contrast to Dijkstra's algorithm, which can be categorized as a greedy algorithm by search for the locally optimal way forward, the Bellman-Ford algorithm is a relaxation algorithm. It updates the travel costs through the graph by repeatedly updating the edges, until the optimal solution can be extracted.
The algorithm was invented in 1958 by Richard Bellman and Lester Ford. Due to its applicability to graphs with negative weights, it is more versatile than Dijkstra's algorithm, but this versatility comes at a cost of increased run times.

1.1 ALGORITHM

Identical to the Dijkstra algorithm, the Bellman-Ford method starts by initializing the distance to the source node to zero and the distance to all other nodes to infinity. In a loop over all edges, we check whether the distance to the end-node of the edge is greater than the distance to the start-node plus the edge value. Mathematically, we have that if

$$d[\texttt{start}] + edge[\texttt{start}-\texttt{to}-\texttt{end}] < d[\texttt{end}] \tag{1.1}$$

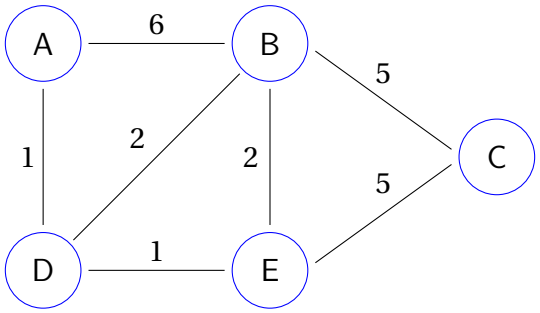

Figure 1.1: graph

with $d[\cdot]$ denoting the stored current distance to a node, we update the distance to the end-node by the lower value $d[\texttt{start}] + edge[\texttt{start}-\texttt{to}-\texttt{end}]$. We sweep through all edges of the graph. This constitutes one iteration of the Bellman-Ford algorithm. Since the longest possible path through the graph can have at most $|V|-1$ segments, we have to perform $|V|-1$ iterations to ensure convergence towards the shortest path from a starting to an ending node in the graph. As a consequence, the Bellman–Ford algorithm runs in $\mathcal{O}(|V|\cdot|E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges, respectively. This run time is longer than the one for Dijkstra's algorithm.

	A	B	C	D	E	A	B	C	D	E
$d[\cdot]$	0	∞	∞	∞	∞	0	6	∞	1	∞
DE = 1	0	∞	∞	∞	∞	0	6_A	∞	1_A	2_D
DB = 2	0	∞	∞	∞	∞	0	3_D	∞	1_A	2_D
BE = 2	0	∞	∞	∞	∞	0	3_D	∞	1_A	2_D
BC = 5	0	∞	∞	∞	∞	0	3_D	8_B	1_A	2_D
AD = 1	0	∞	∞	1_A	∞	0	3_D	8_B	1_A	2_D
EC = 5	0	∞	∞	1_A	∞	0	3_D	7_E	1_A	2_D
AB = 6	0	6_A	∞	1_A	∞	0	3_D	7_E	1_A	2_D
	1st iteration					2nd iteration				

Figure 1.2: Bellman-Ford tableau.

2 IMPLEMENTATION

The python code implementing the Bellman-Ford algorithm is given below.

```
import numpy as np
import scipy as sp

def BellmanFord(ist,isp,wei):
    #---------------------------------
    #  ist:    index of starting node
    #  isp:    index of stopping node
    #  wei:    adjacency matrix (V x V)
    #
    #  shpath: shortest path
    #---------------------------------

    V = wei.shape[1]

    # step 1: initialization
    Inf    = 1e300
    d      = np.ones((V),float)*Inf
    p      = np.zeros((V),int)
    d[ist] = 0

    # step 2: iterative relaxation
    for i in range(0,V-1):
        for u in range(0,V):
            for v in range(0,V):
                w = wei[u,v]
                if (w != 0):
                    if (d[u]+w < d[v]):
                        d[v] = d[u] + w
                        p[v] = u

    # step 3: check for negative-weight cycles
    for u in range(0,V):
        for v in range(0,V):
            w = wei[u,v]
            if (w != 0):
                if (d[u]+w < d[v]):
                    print('graph has a negative-weight cycle')

```

```
39      # step 4: determine the shortest path
40      shpath = [isp]
41      while p[isp] != ist:
42          shpath.append(p[isp])
43          isp = p[isp]
44      shpath.append(ist)
45
46      return shpath[::-1]
```

The associated main code is listed below.

```
1   if __name__ == '__main__':
2
3       # indices of starting and stopping vertices
4       ist = 4
5       isp = 3
6
7       # randomly generated adjacency matrix
8       #N   = 10
9       #ma  = np.around(np.random.uniform(0,1.2,(N,N)))
10      #wei = ma*np.random.uniform(0,30,(N,N))
11      #wei = np.tril(wei,-1) + np.triu(wei,1)
12
13      # adjacency matrix
14      wei = np.array([[ 0, 20,  0, 80, 0,  0, 90,  0],
15                      [ 0,  0,  0,  0, 0, 10,  0,  0],
16                      [ 0,  0,  0, 10, 0, 50,  0, 20],
17                      [ 0,  0, 10,  0, 0,  0, 20,  0],
18                      [ 0, 50,  0,  0, 0,  0, 30,  0],
19                      [ 0,  0, 10, 40, 0,  0,  0,  0],
20                      [20,  0,  0,  0, 0,  0,  0,  0],
21                      [ 0,  0,  0,  0, 0,  0,  0,  0]])
22
23      shpath = BellmanFord(ist,isp,wei)
24      print ist,' -> ',isp,' is ',shpath
```

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 6, 2017

1 LATTICE-BASED OPTION PRICING (COX-ROSS-RUBINSTEIN ALGORITHM)

Options are financial instruments that are based on a contract for a buyer to possibly exercise his/her right to buy or sell an underlying asset for a specified price (the strike price) at or before a specified future time (expiration date). This contract has to be bought, and the question in this section is how to fairly price the contract, given the specified time, specified price, and the current price and volatility of the underlying asset. When buying a contract to buy shares at a later time, we refer to the option as a **call** option; a contract to sell shares at a later time is referred to as a **put** option.
As an example, let's say you wish to buy a car in a month and you want to buy insurance against a possible price drop. You are interested in a contract that promises you a locked-in price on the car in a month from now; this contract should saveguard against potential financial loss. The insurance contract becomes invalid after the expiration date. The price of the insurance contract depends on many variables: (i) the price at which you wish to buy the car (the strike price; the lower the price, the more expensive the insurance), (ii) the duration of the contract (the longer the contract, the more expensive the contract), (iii) the volatility of the market (the more uncertain the market, the more expensive the contract), and (iv) the current interest rate on a bank account (since, alternatively, we could save the money, earn interest and thus compensate for price fluctuations in the price of the car). The question is then: how to fairly determine the price of the insurance contract? "Fair" will be taken as a condition such that alternative financial arrangements won't present any advantage over the insurance contract. An additional condition has to be taken into account: do you have to wait until the end of the contract before buying the car, or can you

buy the car at any time before the expiration date? This is the question about the exercise right of your contract.
The techniques to determine this fair price can be used in many other applications where we have to specify the value of an instrument based on (i) an estimate of risk inherent in the development of the associated asset and (ii) the ambient financial environment.

1.1 Background

The pricing of an option is based on an arbitrage argument. It states that a proper balance of shares of the underlying asset and a risk-free asset should present neither favorable nor unfavorable odds for profit when compared to the option contract. There are various methods of computing the value of option contracts based on this arbitrage argument. Among them is the famous Black-Scholes equation which implements a continuous model involving a partial differential equation to solve for the option price and its temporal evolution from the current time to the time of expiration. An alternative technique is based on the binomial model, proposed in an algorithmic way by Cox, Ross & Rubinstein.

1.2 The binomial model

The binomial model is a discrete-in-time, tree-based model that computes the value of an option contract in two steps. The first step propagates the price of the underlying asset forward in time, applying the probability for a rise or drop of the asset value for each time interval. This step gives the likely evolution of the price under a simple statistical model. Once at expiration date, the value of the option contract can be easily determined. From this distribution at the expiration date, the value of the contract is then traced back (using again an arbitrage argument) to determine the current value of the option contract.

1.3 The forward problem: establishing the binomial tree

The first step establishes a binomial tree for the underlying asset according to the following rule. The price rises by a relative amount u with a probability of p and falls by a relative amount d with probability $1-p$. The relative amount u is given by the volatility σ (or the standard deviation) of the underlying asset; using a random walk argument we have

$$u = \exp(\sigma\sqrt{\Delta t}) \tag{1.1}$$

over a time interval Δt. The drop d is simply $d = 1/u$. The volatility σ can be estimated from historical records, but the assumption of a constant volatility over the duration of the option contract puts a degree of non-realism on the outcome. We still have to determine the probability p of a rise or fall of the underlying asset. To this end, we invoke an arbitrage argument and suppose that the expected relative return (i.e., the return multiplied by its associated probability) is equivalent to the return of a risk-free asset with rate r over the same time interval. We have

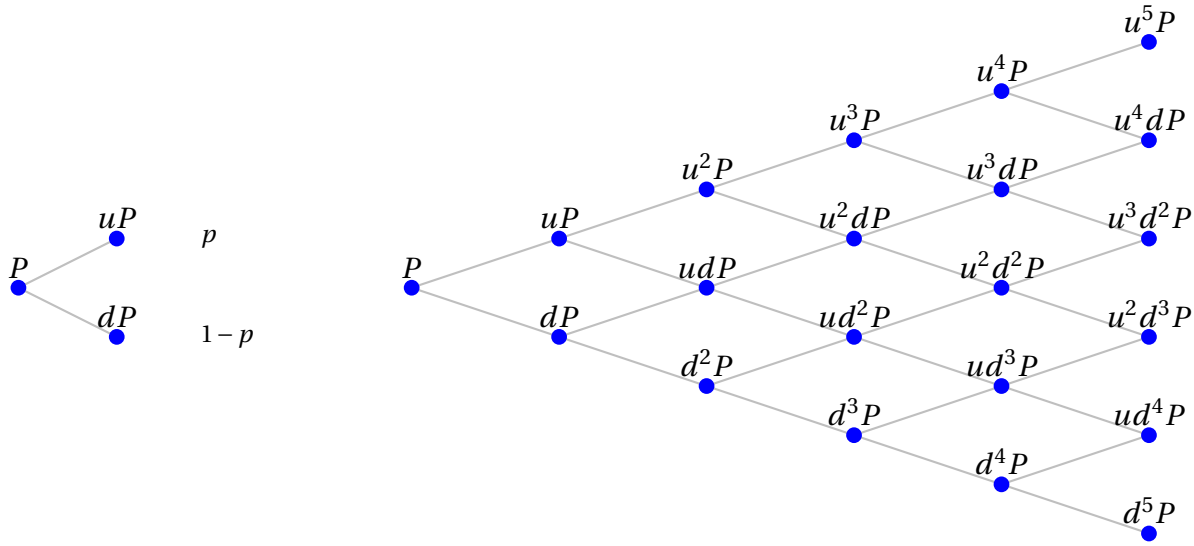

Figure 1.1: Binomial tree structure for computing the development of price P under a binomial statistical model: (left) over one day, (right) over five days, resulting in a tree structure.

$$pu + (1-p)d = \exp(r\Delta t) \tag{1.2a}$$

$$p = \frac{\exp(r\Delta t) - d}{u - d} \tag{1.2b}$$

We can continue this rise-drop procedure for each of the successive days to arrive at a binomial tree structure displayed in figure 1.1.
In the first step of the Cox-Ross-Rubinstein algorithm we build a binomal tree starting from the current date up to the expiration day of the option contract. The final level of the tree then represents the price distribution starting from the current price under a binomial probability model based on the asset's volatility. Figure 1.1(b) gives an example for a 5-level tree, spanning a time interval of five days until expiration.
Example: Given the current price of an underlying asset as $P = 100$, together with the (annualized) volatility of $\sigma = 0.3$ and the (annualized) rate of return of a risk-free asset (e.g. a Treasury note) of $r = 0.05$ we compute the forward binomial tree over five days. From the formula above, we have $u = 1.0191$ and $d = 0.9813$. The results for this example are displayed in Figure 1.2.

1.4 The backward problem: pricing the option contract

Once the price tree for the underlying asset is established, we determine, in a second step, the price of the option contract. To this end, we build a second tree (of identical structure) for the value of the option contract. From the final level of the tree in figure 1.1, representing the price distribution of the underlying asset at expiration, we can easily determine the

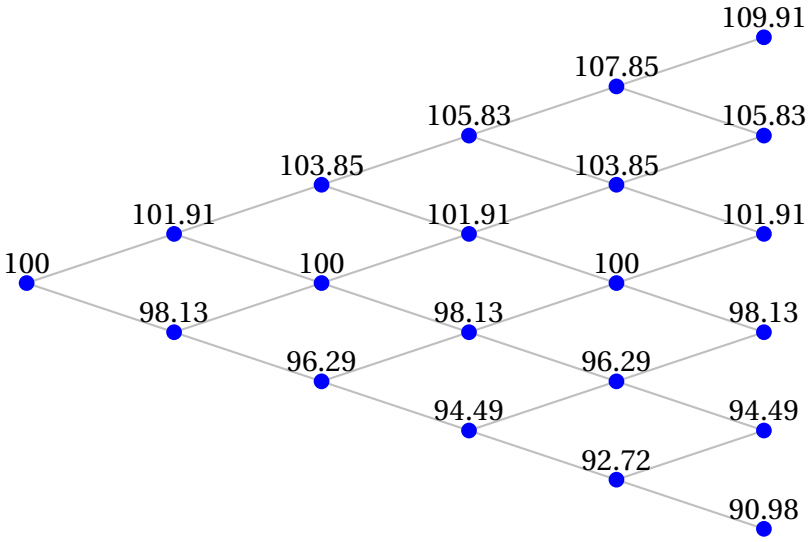

Figure 1.2: Binomial tree structure, for the example given in the text.

value of the option contract. We will consider a call option. In this case, the price C of the call option is given as $C = \max\{0, P - S\}$ where P is the price of the underlying asset and S represents the strike price of the option. This pricing dictates that the option contract at expiration is simply the difference of the current price P minus the strike price (at which we can buy the underlying asset). If the price P is lower than the strike price S, our contract is worthless and there is no point in exercising it, since we can buy the asset cheaper on the open market than by using our option contract.
From this known final price distribution at expiration, we follow the tree back to its root (the current date) by using an arbitrage argument at every node.
We invoke an arbitrage argument as follows. We price the call option such that over one period Δt the gain from the call option is matched by the gain of a mixed portfolio containing n shares of the risky underlying asset and an amount b of the risk-free asset; the variables n and b, i.e. the exact mixture of risky and risk-free assets, are yet unknown. The call price C at the start is thus expressed as $nS + b$. We then have over one period (see Figure 1.4)

$$C \equiv nS + b \quad \longrightarrow \quad unS + \exp(r\Delta t)b \equiv C_u \quad \text{with probability} \quad p \tag{1.3a}$$
$$C \equiv nS + b \quad \longrightarrow \quad dnS + \exp(r\Delta t)b \equiv C_d \quad \text{with probability} \quad 1 - p \tag{1.3b}$$

which, given the call option price from expiration (red box), allows us to determine the call-option prices C at previous time levels. The above equations, in matrix form, read

$$\begin{pmatrix} uS & \exp(r\Delta t) \\ dS & \exp(r\Delta t) \end{pmatrix} \begin{pmatrix} n \\ b \end{pmatrix} = \begin{pmatrix} C_u \\ C_d \end{pmatrix} \tag{1.4}$$

which yields the solution

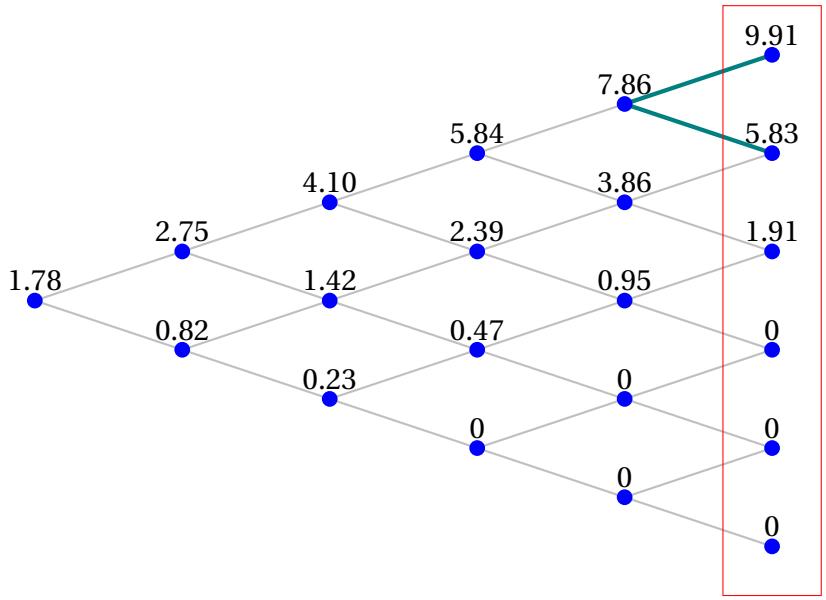

Figure 1.3: Binomial tree structure for the option price, starting at the right edge (red box) and back-tracking to its root by an arbitrage argument (see text).

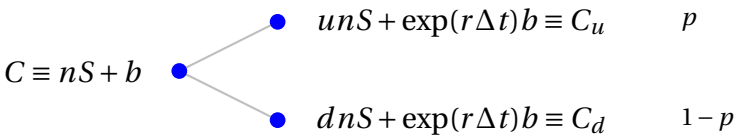

Figure 1.4: Recovery of the call-option value over one time period.

$$n = \frac{C_u - C_d}{(u-d)S}, \tag{1.5a}$$

$$b = \frac{uC_d - dC_u}{(u-d)\exp(r\Delta t)}, \tag{1.5b}$$

and from there the call value C at the earlier time is given as

$$C = \alpha C_u + \beta C_d \tag{1.6}$$

with

$$\alpha = \frac{1}{u-d}\left(1 - \frac{d}{\exp(r\Delta t)}\right) \qquad \beta = \frac{1}{u-d}\left(\frac{u}{\exp(r\Delta t)} - 1\right). \tag{1.7}$$

We then have the full Cox-Ross-Rubinstein algorithm. We first establish a price tree for the underlying asset, using the volatility of the market to determine the amount of relative gain or loss. Once we have the price distribution at expiration, we determine its matching option contract value (since only at expiration do we have information about option pricing). From this final option-price distribution, we recover the option price for the rest of the tree

in a progressive way (from right to left) using an arbitrage argument, until we arrive at the root of the tree (single left node), where we recover today's fair price of the option contract.

1.5 Adjustments for early exercise

The above algorithm assumes that the option contract can only be exercised at the date when it expires. This assumption has established our option value distribution at the right edge of the tree structure (see Figure 1.2 (red box)), where the value of the option at expiration is given by the value of the underlying asset at expiration minus the strike price of the option contract. From this distribution, we back-tracked the value of the option contract to the current date (the root of the tree). An option contract that can only be exercised at the end of its lifetime (at expiration) is known as a **European** option.
In contrast, option contracts that can be exercised *any time* before expiration are referred to as **American** options. Pricing American options follows the same principle with one important difference: when computing the option values during the backward-sweep through the binomial tree we have to compare the computed option value (i.e., the value computed above) to the amount that the current asset value exceeds (in the case of a call option) the strike price. If the former value is larger than the latter, it makes more sense to keep the option contract, and we accept the computed option value. In the reverse case (the difference between the asset and the strike price is larger than the computed value of the option contract), it would make sense to exercise the option right away and realize a risk-free gain. In this case, the fair value of the option contract is the excess of the asset above the strike price; we take this value in the tree, instead of the computed option value.

2 Lattice-based option pricing (Cox-Ross-Rubinstein algorithm)

The `python`-code below implements the Cox-Ross-Rubinstein algorithm, outline above, for European and American call and put options. The tree structure is represented as an array.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def CoxRossRubinstein(volat,riskfree,days,Current, \
5                        Strike,optType):
6      # Cox-Ross-Rubinstein (CRR) algorithm for option
7      # pricing using a binomial tree structure and
8      # arbitrage arguments
9
10     # time step (approx. 250 trading days in a year)
11     dt = 1./250.
```

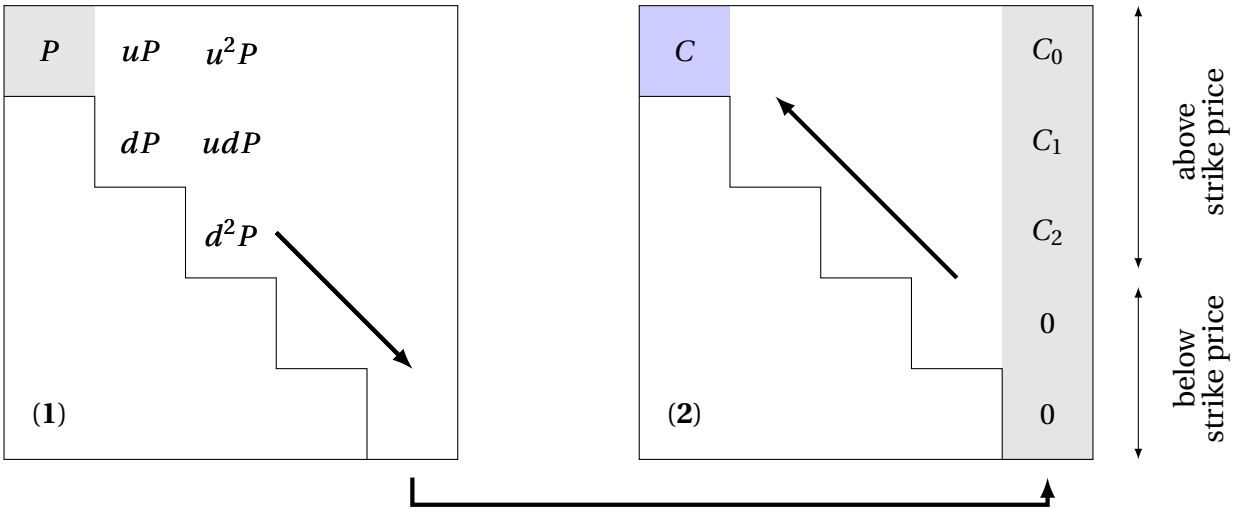

Figure 1.5: Cox-Ross-Rubinstein algorithm (demonstrated for pricing a call option). In the first step, the upper triangular part of the matrix, representing the binomial tree, is filled from the upper-left element (the current price) to the left-most column. From this last column the option price is determined and the results are transferred to the last column of a second matrix. The formula (1.6) is used to fill the columns from right to left, until we arrive at C (light blue box), the value of the option contract today.

```
12      # up/down factor (based on volatility)
13      u = np.exp(volat*np.sqrt(dt))
14      d = 1/u
15      # risk-free factor
16      Rf = np.exp(riskfree*dt)
17      alpha = (1. - d/Rf)/(u-d)
18      beta  = (u/Rf - 1.)/(u-d)
19
20      # mapping matrix for underlying asset
21      PP      = np.zeros((days+1,days+1))
22      PP[0,0] = u
23      PP     += np.diag(d*np.ones(days),-1)
24      # price vector at root of binomial tree
25      P       = np.zeros((days+1,1))
26      P[0]    = Current
27
28      # build underlying-asset binomial tree
29      SS = P.copy()
30      for i in range(0,days):
31          P  = np.matmul(PP,P)
32          SS = np.hstack((SS,P))
```

```
33
34      # evaluation for call options on expiration
35      tmp = SS[:,-1]-Strike
36      CCC = np.clip(tmp,0,max(tmp))
37      # evaluation for put options on expiration
38      tmp = Strike-SS[:,-1]
39      CCP = np.clip(tmp,0,max(tmp))
40
41      # reverse binomial tree for call option price
42      Cnew       = np.zeros((days+1,days+1))
43      Cnew[:,-1] = CCC
44      for it in range(days,-1,-1):
45          for i in range(0,it):
46              Cu = Cnew[i,it]
47              Cd = Cnew[i+1,it]
48              tmp = alpha*Cu + beta*Cd
49              if (optType == 'American'):
50                  tmp2 = SS[i,it]-Strike
51                  tmp  = max(tmp,tmp2)
52              Cnew[i,it-1] = tmp
53      CPrice = Cnew[0,0]
54
55      # reverse binomial tree for put option price
56      Cnew       = np.zeros((days+1,days+1))
57      Cnew[:,-1] = CCP
58      for it in range(days,-1,-1):
59        for i in range(0,it):
60            Cu = Cnew[i,it]
61            Cd = Cnew[i+1,it]
62            tmp = alpha*Cu + beta*Cd
63            if (optType == 'American'):
64                tmp2 = Strike-SS[i,it]
65                tmp  = max(tmp,tmp2)
66            Cnew[i,it-1] = tmp
67      PPrice = Cnew[0,0]
68
69      return PPrice,CPrice
```

```
1   if __name__ == '__main__':
2
3       import pandas as pd
4
```

```
    # read S&P 500 prices from file
    df    = pd.read_csv('SP500Prices.csv')
    AdjC  = df['AdjClose']
    Price = np.array(AdjC[::-1])

    # calculate logarithmic returns
    returns = []
    for i in range(0,len(Price)):
        r = np.log(Price[i]/Price[i-1])
        returns.append(r)

    # compute daily volatility
    volat_d = np.std(returns)
    # adjust to annualized volatility
    volat   = volat_d*np.sqrt(250)

    # input to CRR-function
    Current  = Price[-1]
    days     = 100
    riskfree = 0.05
    Strike   = 2170

    optType  = 'American'   # or 'European'

    # compute put and call option values
    P,C = CoxRossRubinstein(volat,riskfree,days,Current,\
                            Strike,optType)

    # output
    print('Historical volatility = %0.5f' % volat)
    print('Current price         = %.2f' % Current)
    print('Strike price          = %4i' % Strike)
    print('Option type           = %s' % optType)
    print('days till expiration  = %3i\n' % days)
    print('SPX call option       = %.2f' % C)
    print('SPX  put option       = %.2f' % P)
```

3 Example 1

We compute the price distribution of a call option (versus the value of the underlying price) as it develops from today until expiration in 30 days.

```
if __name__ == '__main__':
```

```
2
3       # parameters
4       riskfree = 0.01
5       volat    = 0.3
6       optType  = 'European'
7       Current  = np.arange(45,65,1) # scan over prices
8
9       # loop over days until expiration
10      for i in range(30):
11          days   = 5*i
12          Strike = 55
13          CPrice = np.zeros(len(Current))
14          j = 0
15          for CC in Current:
16              P,C = CoxRossRubinstein(volat,riskfree,days,CC,\
17                                      Strike,optType)
18              CPrice[j] = C
19              j += 1
20
21          plt.plot(Current,CPrice)
22      plt.savefig('Call.png')
23      plt.draw()
24      plt.show()
```

4 EXAMPLE 2

We compute the price distribution (versus the value of the underlying price) of a portfolio consisting of one call and one put option for the same strike price, as it develops from today until expiration in 30 days. This option configuration guards against a quick rise in the market (when the call-option produces profit) and against a sharp drop in the market (when the put-option produces profit). Only when the market fails to move do we realize a loss.

```
1   if __name__ == '__main__':
2
3       # parameters
4       riskfree = 0.01
5       volat    = 0.3
6       optType  = 'European'
7       Current  = np.arange(45,65,1) # scan over prices
8
9       # loop over days until expiration
```

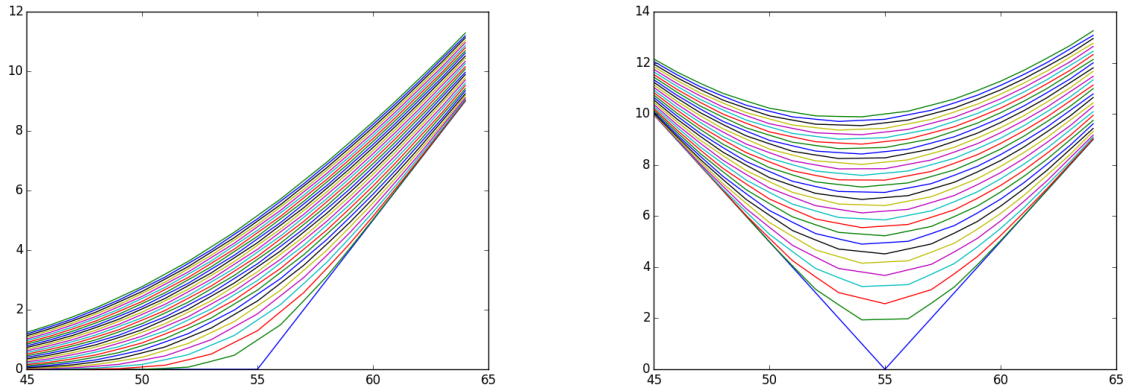

Figure 4.1: (a) Price of call option versus price of underlying asset for different expiration times. (b) Price of call-put combination (same strike price) versus price of underlying asset for different expiration times.

```
10      for i in range(30):
11          days   = 5*i
12          Strike = 55
13          CPrice = np.zeros(len(Current))
14          j = 0
15          for CC in Current:
16              P,C = CoxRossRubinstein(volat,riskfree,days,CC,\
17                                      Strike,optType)
18              CPrice[j] = C + P
19              j += 1
20
21          plt.plot(Current,CPrice)
22      plt.savefig('Straddle.png')
23      plt.draw()
24      plt.show()
```

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 14, 2017

1 GLOBAL OPTIMIZATION (PARTICLE-SWARM ALGORITHM)

Particle swarm optimization (PSO) is a technique that uses swarm intelligence to find a global optimum of a user-defined multi-variate function. Swarm intelligence systems consist of a number of agents that communicate with other agents in the swarm, exchanging information about the local and neighboring environment. By this exchange of information, the swarm moves through a multi-variate function space converging towards a global optimum.
To cite the inventors of PSO, the "particle swarm algorithm imitates human (or insects) social behavior. Individuals interact with one another while learning from their own experience, and gradually the population members move into better regions of the problem space".
Underlying a particle swarm optimization is a **fitness function** that for each particle can be easily evaluated and measures the "degree of success" of our optimization. PSO is based on particles that move according to a particle velocity associated with each particle. The computation of the particle velocity has three component (see Figure 1.1): (i) an **inertial** component, (ii) a **cognitive** component, and (iii) a **social** component. The inertial component maintains a fraction of the current velocity vector, by continuing along the same direction as during the previous step. The cognitive component is based on the difference between the current position and the particle's best position (in terms of the fitness function) during its evolution. This component accounts for memory effects of the optimization process as witnessed by each single agent. Finally, the social component of the new velocity vector is proportional to the difference of the current agent position and the current best position of the entire swarm. This final component accounts for the communication of the best ob-

tained results (up to this point) to all agents, inducing a kind of flocking towards the global optimum.

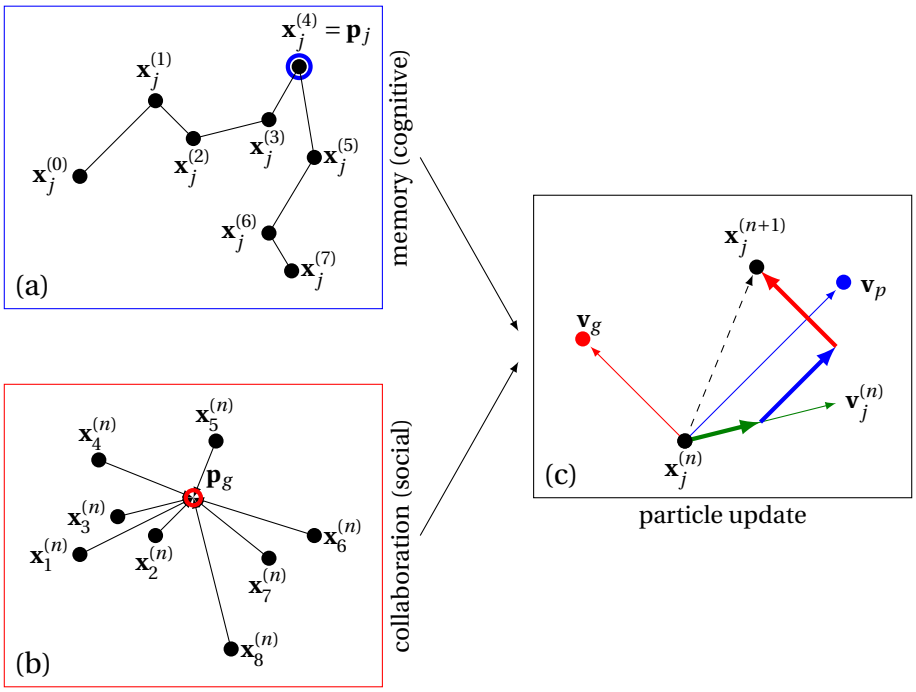

Figure 1.1: Principles of particle swarm optimization, combining a cognitive (a) and social (b) component. For the cognitive component, the best position for each particle j throughout the iterations is determined and stored as $\mathbf{p}_j$. For the social component, the best position throughout the iterations and over the entire swarm is determined and stored as $\mathbf{p}_g$. The velocity vector for each particle's update (c) involves a fraction of the previous velocity vector (in green), a random portion of the velocity $\mathbf{v}_p$ towards the particle's best position (in blue) and a random portion of the velocity $\mathbf{v}_g$ towards the globally best position (in red). The combined velocity vector determines the new position $\mathbf{x}_j^{(n+1)}$ of the j-th particle.

The final expression for the velocity update of each particle is then given as

$$\mathbf{v}_j^{(n+1)} = \omega \mathbf{v}_j^{(n)} + \mathbf{R}_1 \otimes \left(\mathbf{p}_j - \mathbf{x}_j^{(n)}\right) + \mathbf{R}_2 \otimes \left(\mathbf{p}_g - \mathbf{x}_j^{(n)}\right) \tag{1.1a}$$

$$\mathbf{x}_j^{(n+1)} = \mathbf{x}_j^{(n)} + \mathbf{v}_j^{(n+1)} \tag{1.1b}$$

with $\mathbf{x}_j$ denoting the position of the j-th particle and $\mathbf{v}_j$ its velocity. The superscript $^{(n)}$ stands for the iteration counter. The vector $\mathbf{p}_j$ stores the personal best of the j-th particle; the vector $\mathbf{p}_g$ stands for the global optimum across the entire swarm. The symbol $\otimes$ represents elementwise multiplication. The coefficient ω measures the inertial effects and regulates how much the new velocity follows the old one. The two coefficient vectors $\mathbf{R}_1$

and $\mathbf{R}_2$ consist of a deterministic scalar component and a random vector component. The deterministic part allows the user to weight the various components of the velocity update formula, e.g., how strongly we should follow a current global optimum. The random part (chosen as a uniformly distributed random number between zero and one) introduces a probabilistic component to the optimization algorithm that allows the exploration of other parts of the problem space and helps avoid the convergence to local rather than global optima.
The computation of the velocity vector for the new, updated particle position balances the concept of solution-space exploration and solution-space exploitation. The exploration part is induced by the random coefficients (as well as the coefficients) and allows the probing of solutions away from the current optima. The exploitation part is accomplished by the communication of the global optimum to all particles, which yields a general tendency of the particles to follow a "leading" particle (with the currently best solution). The design of an effective weighing strategy for exploration and/or exploitation is a challenging task and common to many optimization algorithms.
The above algorithm in its most primitive form can be used to find optima of complex, high-dimensional functions that are unconstrained by additional conditions. However, in most practical cases, we have to deal with side constraints either on the solution space itself or by enforcing supplementary conditions. Constraints can be in the form of equalities (equaitons) or inequalities (bounds).

1.1 Accounting for constraints

The incorporation of side constraints for our optimization problem requires additional modifications to the general particle swarm algorithm.
We formulate a general optimization problem as

$$f(x_1, x_2, \ldots, x_n) \longrightarrow \text{opt} \tag{1.2a}$$
$$g_j(x_1, x_2, \ldots, x_n) = 0 \qquad j = 1, \ldots, J \tag{1.2b}$$
$$h_k(x_1, x_2, \ldots, x_n) \leq 0 \qquad k = 1, \ldots, K \tag{1.2c}$$
$$l_i \leq x_i \leq h_i \qquad i = 1, \ldots, n \tag{1.2d}$$

where $x_1, \ldots, x_n$ denotes our n-dimensional solution vector, and f is the cost function we wish to optimize. The functions g_j impose J equality constraints that the solution has to satisfy to be feasible. Similarly, the functions h_k impose K inequality constraints. Finally, the last condition restricts the solution vector to given intervals in its various components.

The simplest constraints are on the solution itself, i.e., equation (1.2d). They are enforced by monitoring the solution after each update and **clipping** the components of the solution vector back to the permissible interval. Due to inertial part of the update formula, it is reasonable to expect that the same clipped particle will leave the interval again in the next iteration. For this reason, the velocity component corresponding to the clipped solution component is also changed in sign. The technique is referred to as **velocity mirroring**.

We also enforce limits on the velocities, after each iteration, to avoid excessive speeds across the solution space, but instead encourage a more careful exploitation of the solution space.

Besides the ranges for the solution, additional constraints come into two forms: equality and inequality constraints, and there are several techniques how to handle them in the algorithm. For inequality constraints, the most common ones are (i) feasibility checks (which we will use), and (2) adding the constraints to the cost function via penalty terms. In the first technique, we simply check whether a potential new solution $\mathbf{x}_j^{(n+1)}$ satisfies all inequality constraints; only then will it be accepted as a legitimate solution. Otherwise it will be rejected. For simple (and liberal) constraints and for rather low-dimensional problems, this technique works quite well; for higher-dimensional problems or very stringent constraints, additional modifications to the algorithm are necessary. The second strategy adds the constraints to the cost function via penalty terms: in this manner, not satisfying the constraints causes additional cost and is thus discouraged. Again, difficulties arise for high-dimensional and highly restricted optimization problems and special care has to be exercised.
Equality constraints are more difficult to deal with in the particle swarm algorithm. Two general lines are followed in the PSO-community. A nonlinear Newton-Raphson solver for the equality constraint is formulated, and an approximate Jacobian is formulated based on the current particle position, which is then used to advance the particles while observing the equality constraints. A second technique approaches the problem by formulating a repair function that projects the non-feasible solution back onto a feasible one. This projection is applied after each iteration.

1.2 Implementation

```
1  import numpy as np
2  import scipy as sp
3  import math as ma
4
5  def sphere(x):
6      # spherical fitness functional
7      c = ((x-1)**2).sum(axis=0)
8      return c
9
10 def rosenbrock2(x):
11     # 2D Rosenbrock fitness functional
12     xx = x[0,]
13     yy = x[1,]
14     a  = 2.
15     c  = (a-xx)**2 + 100.*(yy-xx**2)**2
```

```
    return c

def rosenbrock3(x):
    # 3D Rosenbrock fitness functional
    xx = x[0,]
    yy = x[1,]
    zz = x[2,]
    c  = (1.-xx)**2 + (1.-yy)**2 + \
         100.*(yy-xx**2)**2 + 100.*(zz-yy**2)**2
    return c

def rastrigin2(x):
    # 2D Rastrigin fitness functional
    xx = x[0,]
    yy = x[1,]
    a  = 10.
    c  = 2*a + xx**2 - a*np.cos(2*ma.pi*xx) + \
         yy**2 - a*np.cos(2*ma.pi*yy)
    return c

def rastrigin3(x):
    # 3D Rastrigin fitness functional
    xx = x[0,]
    yy = x[1,]
    zz = x[2,]
    a = 10.
    c  = 3*a + xx**2 + yy**2 + zz**2 - \
         a*np.cos(2*ma.pi*xx) - a*np.cos(2*ma.pi*yy) - \
         a*np.cos(2*ma.pi*zz)
    return c
```

```
if __name__ == '__main__':

    #------ particle swarm optimization (PSO)

    # number of variables and swarm size
    nvar   = 2
    nswarm = 30

    # bounds on the variable x
    xMin = -5
    xMax =  5
```

```
12
13      # bounds on the velocity
14      vMax = 0.1*(xMax-xMin)
15      vMin = -vMax
16
17      # maximum number of iterations
18      itMax = 5000
19      # inertia and inertia damping
20      w = 1
21      wdamp = 0.99
22      # acceleration coefficients
23      c1 = 2
24      c2 = 2
25
26      # initialization
27      x     = np.random.uniform(xMin,xMax,(nvar,nswarm))
28      v     = np.zeros((nvar,nswarm),float)
29      c     = rosenbrock2(x)
30      p     = x.copy()
31      g, ig = c.min(0),c.argmin(0)
32      gx    = x[:,ig]
33
34      # PSO main loop
35      for i in range(1,itMax+1):
36
37          # inertia damping
38          w = w*wdamp
39
40          for j in range(0,nswarm):
41
42              # random weights for cognitive and social behavior
43              Rcog = np.random.uniform(0,1,(1,nvar))
44              Rsoc = np.random.uniform(0,1,(1,nvar))
45
46              # update velocity (inertia + cognitive + social)
47              v[:,j] = w*v[:,j] + c1*Rcog*(p[:,j] - x[:,j]) + \
48                       c2*Rsoc*(gx - x[:,j])
49
50              # apply velocity limits
51              v[:,j] = np.clip(v[:,j],vMin,vMax)
52
53              # update position
54              x[:,j] += v[:,j]
55
```

```
56              # velocity mirroring
57              imirr      = ( (x[:,j] < xMin).nonzero() or \
58                             (x[:,j] > xMax).nonzero() )
59              v[imirr,j] = -v[imirr,j]
60
61              # apply variable limits
62              x[:,j] = np.clip(x[:,j],xMin,xMax)
63
64              # evaluation of best cost
65              cc  = rosenbrock2(x[:,j])
66              ccp = rosenbrock2(p[:,j])
67
68              # update particle-best and globally best solution
69              if cc < ccp:
70                  # update particle-best solution
71                  p[:,j] = (x[:,j]).copy()
72                  if cc < g:
73                      # update globally best solution
74                      g  = cc.copy()
75                      gx = (x[:,j]).copy()
76                      print(g)
```

1.3 Extension to multi-objective particle swarm optimization (MO-PSO)

In many applications of optimization, we have to deal with multiple objectives that have to be satisfied simultaneously. In most cases, these objective are conflicting, and an improvement along one objective can deteriorate the optimal solution when measure in another objective. The task then is to compute an optimal solution that satisfies all constraints and optimally balances the multiple objectives. Problems of this type are referred to as multi-objective optimization problems. A classical example is the optimization of a portfolio of financial instruments: we commonly cannot simultaneously maximize the return of the portfolio and minimize its volatility. A superior portfolio return usually induces a certain amount of volatility.

For multi-objective optimization, we have to introduce the idea of **domination**: we say that a solution vector **a** dominates over a solution vector **b**, if **a** is no worse than **b** in all given objectives (fitness functions), and if **a** is strictly superior than **b** in at least one of the multiple objectives (fitness functions).

We trace each solution in a higher-dimensional objective space, where each coordinate corresponds to the fitness of the solution regarding the respective coordinate. For the above example of portfolio optimization, we plot the return and the volatility of each solution in a two-dimensional plane. As the particle swarm moves through problem space and as the above dominant concept is applied, we converge towards a front in this multi-variate

fitness/objective space: this front is known as the **Pareto front**.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 20, 2017

1 COMBINATORIAL OPTIMIZATION (ANT-COLONY ALGORITHM)

Combinatorial optimization is concerned with finding an optimal solution from a large but finite set of possibilities. These possibilities represent combinations of subsets drawn from the larger set. Some common problems of combinatorial optimization are the traveling salesman problem (TSP), where a finite set of cities have to be visited only once on a circular path of minimal length, or the minimum spanning tree problem, where all points of a finite set of points are connected by an edge-weighted, undirected graph of minimum total edge weight (and without any cycles). In most cases, an exhaustive search through all combinations is not a viable option. For example, for the traveling salesman problem with 45 cities (as it is solved below) an exhaustive search of all possible roundtrips would require finding the minimum of 45! possibilities, where $45! \approx 1.2\ 10^{56}$. A similar problem with 22 cities (also treated below) still yields $22! \approx 1.1\ 10^{21}$ combinations. It is these staggering numbers why, from an algorithmic point of view, combinatorial optimization problems rank among the most challenging problems in optimization.

In this section we introduce and implement a meta-heuristic algorithm, inspired by nature: the **ant colony algorithm**. It has been introduced in 1992 by Marco Dorigo and develop further by numerous scientists since then. Since its introduction, it has found a great many applications in operations research, artificial intelligence, machine learning and many other fields.

It is designed by observing a group of ants foraging for food sources from their nests, in essence solving a combinatorial path optimization problem. While each individual ant has a limited capability of "solving the food problem", the group, in contrast, is capable of impressive solutions to complex problems. By proper communication between individual

ants, passing on their findings, the path optimization by ants represents a grand example of **swarm intelligence** or intelligent group collaboration. In ant colonies, communication is accomplished by leaving chemicals (**pheromones**) on the chosen trails. These pheromones influence the successive ant's behavior in as much as paths laced with more pheromones are preferred over paths with less pheromones. This preference, however, is *stochastic* in nature: ants will choose a higher-pheromone trail with higher probability than selecting a lower-pheromone path; still, it will choose a lower-pheromones trail from time to time. Spreading and following pheromones thus promotes an **autocatalytic** behavior: as ants more often follow the higher-pheromone trail, this trail becomes increasingly attractive. This mechanism creates a positive feedback loop.

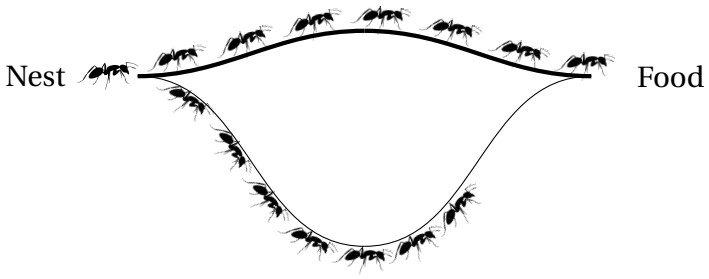

Figure 1.1: Principle of ant colony optimization, explained on a two-branch path.

We explain the ant-colony algorithm by considering the double-path experiment with asymmetric branches. Furthermore, we assume that all ants travel at the same speed and leave the same amount of pheromones on the trail. Ants are foraging for food from the nest. They initially choose paths at random; in our example, ants are as likely to choose the upper as they are to select the lower path. While they move towards the food source, they spread pheromones along the respective trails. The ants on the short, upper path return back to the nest in far shorter time than the ants that chose the longer, lower path. By the time the next ants are exploring the two paths, the upper path will be laced with more pheromones and thus be more likely to be chosen by subsequent ants. Over time, the upper path will dominate over the lower path as the shortest way to the food source.

Casting this behavior into a metaheuristic algorithm, we need to decide on a methodology how to distribute pheromones along trails and determine quantitative measures on how to use the pheromone information when making choices about possible paths.

The probability of choosing a certain path at a crossing point depends on the amount of deposited pheromones. In view of the double-path problem above, we have the following **stochastic model** for the probability of choosing path s from a reference point i over an alternative path l (also from the same reference point i)

$$P_{is}(t) = \frac{(t_s + \phi_{is}(t))^\alpha}{(t_s + \phi_{is}(t))^\alpha + (t_s + \phi_{il}(t))^\alpha} \tag{1.1}$$

with $t_s = l_s / v$ as the travel time along path s, with l_s as the length of path s, and v as the (uniform) velocity at which all ants travel. The quantity ϕ_{is} is the total amount of pheromones

on the branch between reference point i and the end of the path s. The same quantity ϕ_{il} then denotes the amount of pheromones on the alternative route l. The parameter α determines the probability distribution for the decision and is selected experimentally as $\alpha = 2$. Once P_{is} is determined, we then draw a random number from a uniform distribution and decide on the path to take by comparing it to P_{is}; this is akin to the rejection method for making decisions based on general probability distributions. For combinatorial optimization problems that involve a connection between a source (ant heap) and a target (food), it is important to have the ants travel towards the target and back to the source, while depositing pheromones on the forward and backward paths. For our application to the traveling salesman problem (see below), this is not necessary.

While the above outline already constitutes the rudimentary version of the ant-colony optimization algorithm, there are various improvements that can be considered. Loop correction is one of the most common: it proposes to conduct the forward path towards the target without spreading pheromones, after which we check the chosen path for loops; once the loops are eliminated, pheromones are spread *a-posteriori* on the loop-corrected path. Another improvement (which we will also implement below) is concerned with laying down more pheromones on more optimal paths, or considering the current optimality of the path in the probability of choosing a path. For example, we can deposit additional pheromones, indirectly proportional to the length of the trip, on better path combinations. With this trick we need less ants to converge to the optimal solution.

One aspect is also important in ant-colony optimization, which was not included in the original version of the algorithm: **pheromone evaporation**. We assume a constant proportion of pheromones to evaporate and vanish from the path over time. This feature allows the ants to explore less optimal and less-travelled routes; from a mathematical point of view, it helps avoid being trapped in local minima.

The full ant-colony optimization (ACO) algorithm then reads

1. Place a small and uniform pheromone level on all branches of the network.
2. Randomly place ants on the nodes of the network.
3. Move the ants forward through the network, choosing a path by probabilistically selecting the next node based on the relative pheromone levels of surrounding nodes.
4. Eliminate loops in the traced path.
5. Retrace the steps and deposit pheromones.
6. Evaporate the pheromones (globally).
7. Add additional pheromones to the retraced arcs, inversely proportional to the path length.
8. Go back to step 2, until converged.

The main characteristics of the ant-colony optimization algorithm is that it is (i) bio-inspired, (ii) based on stochastic searching of optimal solutions and (iii) relying on a positive feedback mechanism via communication and learning by pheromone levels.

1.1 Application to the traveling salesman problem (TSP)

The traveling salesman problem is the quintessential combinatorial optimization problem and often used to benchmark and showcase combinatorial optimization algorithms. It tries to solve the problem of visiting all points in a plane once on a circular path of minimal length. It is akin to a salesman who has to visit a set of cities once, while minimizing the distance he travels.

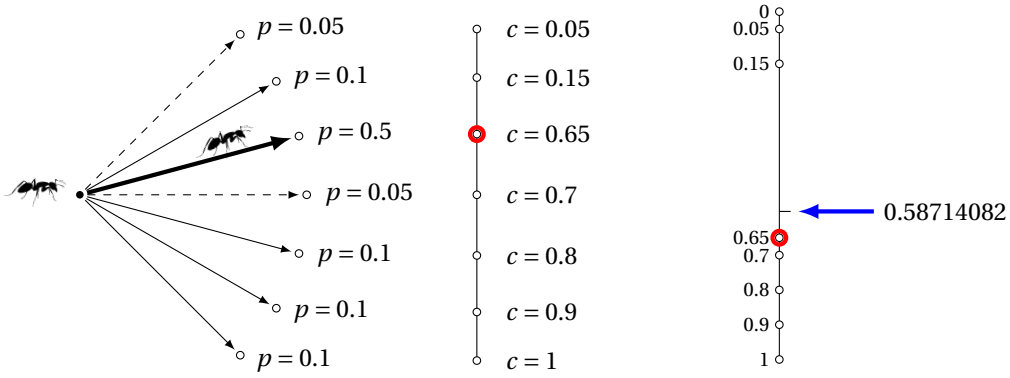

Figure 1.2: Choosing a path. The probabilities p are determined by the amount of pheromones on the trails and the length of the trails. A random number is drawn (see blue arrow) and compared to the cumulative probabilities c, from which a path is selected (indicated by red circle).

Following the ant-colony algorithm outlines above, we initialize all $N(N-1)$ edges with a uniform amount of pheromones. We then choose a starting point for k ants randomly from the N nodes; each ant follows a path, choosing the next node stochastically based on the pheromone level of the various choices. To accomplish a circular path, we introduce a variables `Unv` which records the previously visited nodes. When choosing the next node, this variable guarantees that previously visited sites are excluded. After a circular path is established, its path-length is determined and pheromones are deposited on the path according to the path-length: short paths are "rewarded" more than longer paths. A running record of the best solution will be kept. As a final step in each iteration, the pheromone levels on all paths will be evaporated, which is accomplished by a simple multiplication of the pheromone matrix by a scalar factor less than one. The algorithm is stopped after a fixed number of iterations; other stopping criteria are feasible as well.
The code below is implementing the ant-colony algorithm to solve the traveling salesman problem.

```
import numpy as np
import scipy as sp
import math as ma
import matplotlib.pyplot as plt

def getCities(nc):
    # city data (latitude - longitude)
    yy = np.zeros(45)
    xx = np.zeros(45)
    yy[0] = 48. + 12./60; xx[0] = 16. + 22./60 # Vienna
    .
    .
    .
    yy[44] = 50. + 27./60; xx[44] = 30. + 31.5/60 # Kiev
    .
    .
    .
    # initialize the coordinates of the cities
    x  = np.zeros(nc)
    y  = np.zeros(nc)
    # copy city data and calculate distance matrix
    x  = xx[0:nc]
    y  = yy[0:nc]
    sc = cc[0:nc]
    D  = calcDist(y,x)

    return x,y,D,sc
```

```
def calcLatLonDist(Lat1,Lon1,Lat2,Lon2):
    # calculate the distance between two
    # points given in (Lat,Lon)
    R    = 6373  # average radius of Earth (in km)
    # conversion from degrees
    La1  = Lat1/180*ma.pi
    La2  = Lat2/180*ma.pi
    Lo1  = Lon1/180*ma.pi
    Lo2  = Lon2/180*ma.pi
    # formula from the U.S. Census Bureau website
    dLat = La2-La1
    dLon = Lo2-Lo1
    a    = (ma.sin(dLat/2))**2 + \
           ma.cos(La1)*ma.cos(La2)*(ma.sin(dLon/2))**2
    c    = 2*ma.atan2(ma.sqrt(a),ma.sqrt(1-a))
```

```
        dis  = R*c
        return dis
```

```
def calcDist(La,Lo):
    # calculate the distance matrix between the cities
    n    = len(La)
    dist = np.zeros((n,n))
    for i in range(n):
        for j in range(i+1,n):
            L1 = La[i]
            L2 = La[j]
            B1 = Lo[i]
            B2 = Lo[j]
            dist[i,j] = calcLatLonDist(L1,B1,L2,B2)
            dist[j,i] = dist[i,j]
    return dist
```

```
def drawPath(path,X,Y):
    # graphics output
    nc = len(path)
    XX = list()
    YY = list()
    px = path + [path[0]]
    for i in px:
        XX.append(X[i])
        YY.append(Y[i])

    plt.plot(XX,YY,'o',XX,YY)
    # plt.savefig('PATH.png')
    plt.draw()
    plt.show()
```

```
def pathLength(path,D):
    # step 1: get city-index-pairs
    pairs = zip(path, path[1:] + [path[0]])
    # step 2: sum the city-pair distances
    pL    = sum([D[r,c] for (r,c) in pairs])
    return pL
```

```
def compNextBranch(p):
    r = np.random.random()
    c = np.cumsum(p)
```

```
        j = np.argwhere(c >= r).flatten().min()
        return j
```

```
def costFunction(path,D):
    # could be changed to something else
    C = pathLength(path,D)
    return C
```

```
def genPath(eta,tau):
    # generate a path connecting all cities
    alpha = 2.  # pheromone exponential weight
    beta  = 2.  # heuristic exponential weight
    nc    = len(eta)
    Unv   = np.ones(nc) # keep track of visited cities
    # step 1: randomly choose a city
    curr      = np.random.randint(0,nc-1)
    Unv[curr] = 0
    # step 2: start the path
    path    = [curr]
    # visit all cities from the starting city
    for i in range(nc-1):
        p_tau = np.power(tau[curr,:],alpha)
        p_eta = np.power(eta[curr,:],beta)
        p_et  = (p_tau*p_eta)*Unv
        p     = p_et/sum(p_et) # prob. distrib. to move from current
        curr  = compNextBranch(p)
        Unv[curr] = 0
        path.append(curr)
    return path
```

```
def optAnts(n,x,y,D,maxit,nAnts):
    # optimization parameters
    Q        = 1.
    tau0     = 10.*Q/(n*np.mean(D)) # initial pheromones
    rho      = 0.15;     # evaporation rate
    # initialization
    E        = np.eye(n)
    eta      = 1./(D + E) - E      # heuristic information matrix
    tau      = tau0*np.ones((n,n)) # pheromone matrix
    BestCost = np.zeros(maxit)     # array to hold best cost values
    BestSol  = 1.e30
    AntPath  = np.zeros((nAnts,n), dtype=int)
```

```
13      AntCost  = np.zeros(nAnts)
14
15      for it in range(maxit):
16          # move ants
17          for k in range(nAnts):
18              path         = genPath(eta,tau)
19              AntPath[k,:] = path
20              AntCost[k]   = costFunction(path,D)
21              if (AntCost[k] < BestSol):
22                  BestSol  = AntCost[k]
23                  bpath    = AntPath[k].tolist()
24          pL = pathLength(bpath,D)
25          print it,'pathLength (aco) = ',pL
26
27          #...update pheromones
28          for k in range(nAnts):
29              path  = AntPath[k,:].tolist()
30              pairs = zip(path, path[1:] + [path[0]])
31              # step 2: update the city-pair pheromones
32              for (r,c) in pairs:
33                  tau[r,c] = tau[r,c] + Q/AntCost[k]
34          # evaporation
35          tau = (1.-rho)*tau
36
37          #...store best cost
38          BestCost[it] = BestSol
39
40          #...show iteration information
41          #print 'iteration = ',it,' best cost = ',BestCost[it]
42      return bpath,BestCost
```

```
1   if __name__ == '__main__':
2
3       nc             = 45 # number of cities considered (max: 45)
4       x,y,D,sc       = getCities(nc) # construct model
5       maxit          = 250  # maximum number of iterations
6       nAnts          = 100  # number of ants (population size)
7       bpath,BestCost = optAnts(nc,x,y,D,maxit,nAnts)
8       for i in bpath:
9           print sc[i],' > ',
10      drawPath(bpath,x,y)
11
12      plt.plot(BestCost)
```

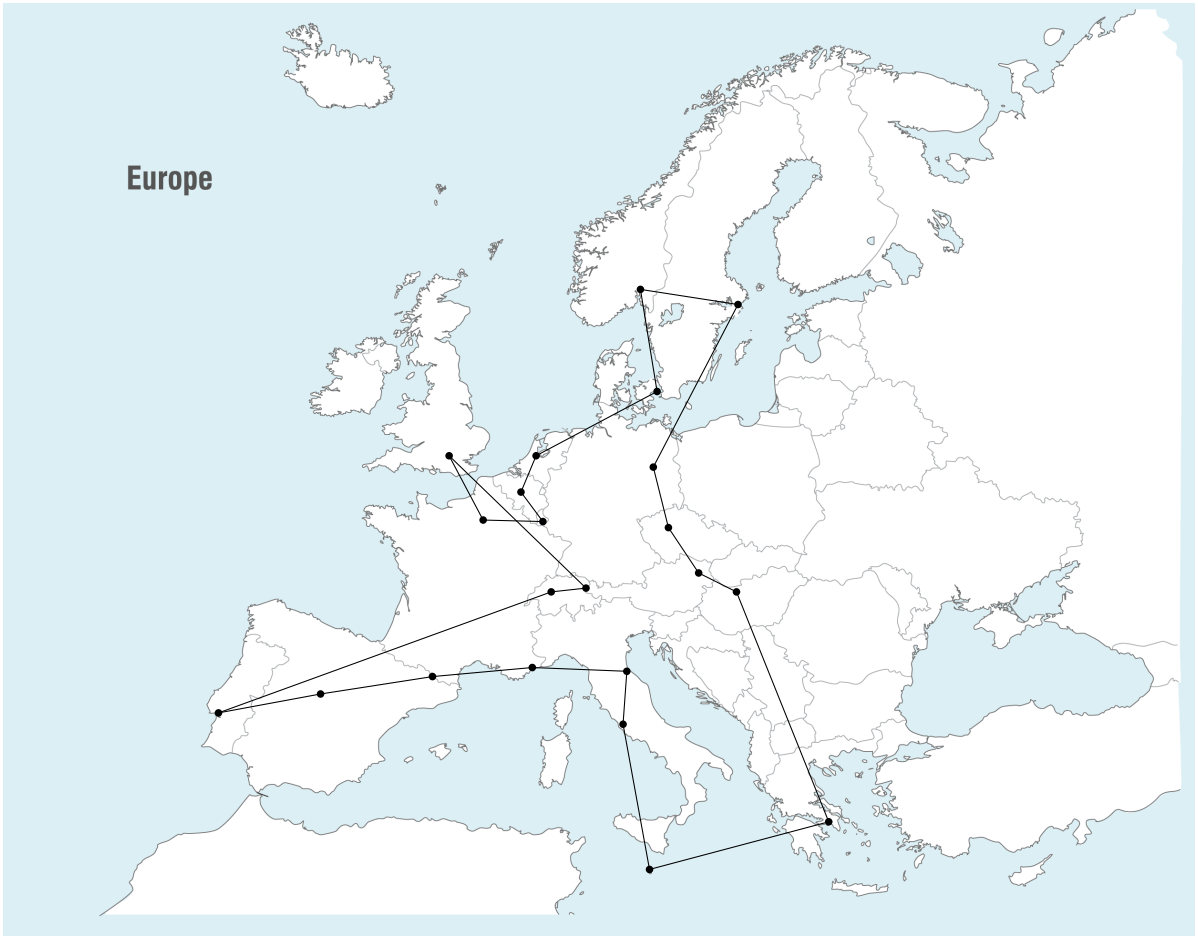

Figure 1.3: Solution to the traveling salesman problem across 22 selected European capital cities. The path length is 11503.74 kilometers. This is the result after 250 iterations, using 100 ants. After these iterations, the path shows crossings, which are clearly suboptimal.

```
13        plt.show()
```

The application of the above code to the traveling salesman problem for a set of European capital sities is given below. Using a colony of 100 ants and running 250 iterations on a 22-city configuration, we observe a rather short, but not optimal circular path. In particular, the path-crossing involving London, Paris, Brussels and Luxembourg adds unnecessary length to the path; by untangling this loop, a shorter and more optimal path seems possible. The algorithm clearly converged to a local minimum. Running the code for more iterations could solve this problem, since due to the pheromone evaporation other path-configurations may be explored. A more direct method for directing the algorithm towards a global minimum (using local corrections) is dealt with in the exercises.
The situation for more cities (now 45 selected European capitals) shows a worsening of the path-crossing problem. Again, we use 100 ants and run the code for 250 iterations (admit-

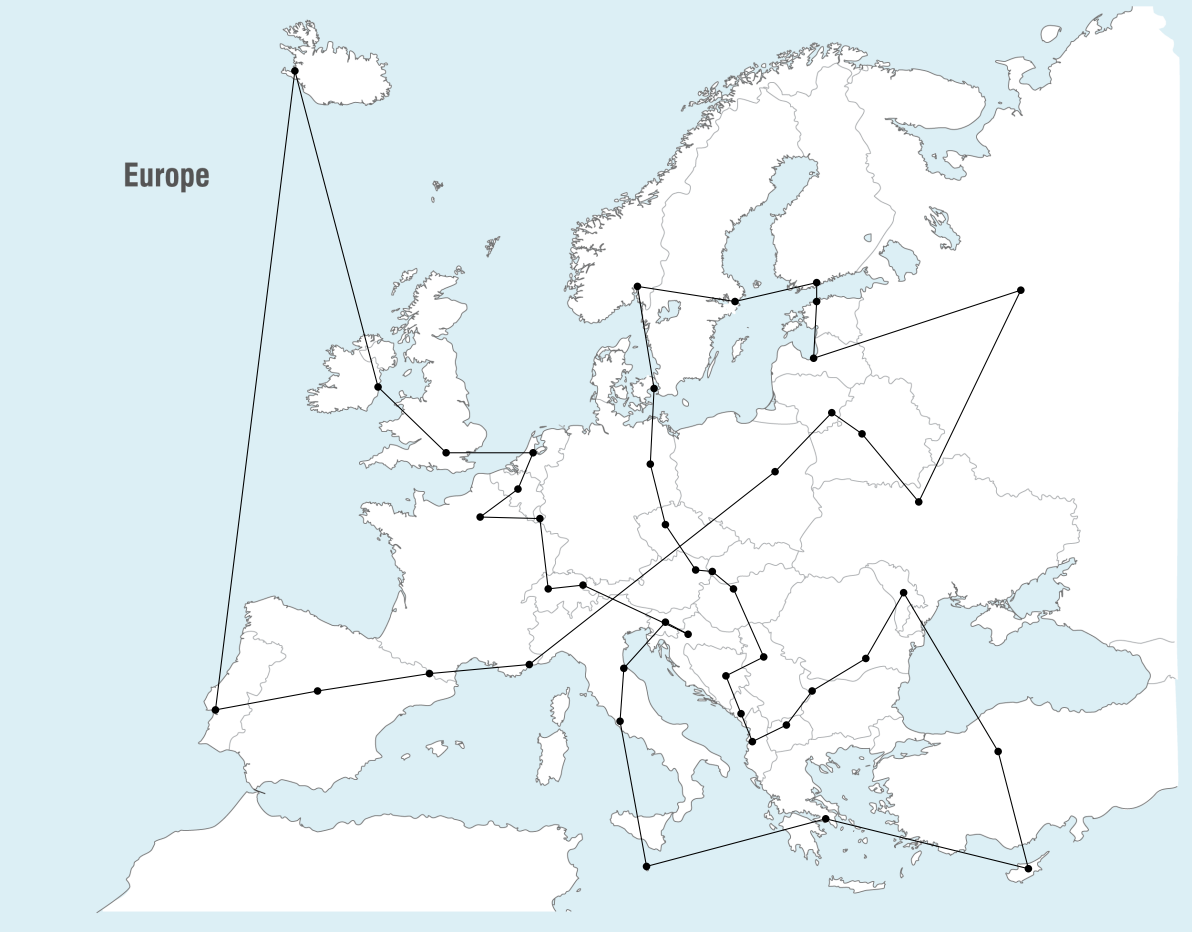

Figure 1.4: Solution to the traveling salesman problem across 45 selected European capital cities. The path length is 21476.80 kilometers. This is the result after 250 iterations, using 100 ants. After these iterations, the path shows crossings, which are clearly suboptimal.

tedly, too low a number of iterations). Even though the path-length of 21476 kilometers is within about a 1000 kilometers of the optimum, the proposed path is clearly not optimal. The trip from Monaco (Southern France) to Warsaw (Poland) will surely not be part of the final, optimal solution. Running the code for more iterations may mitigate the problem, but a simple hybrid modification, combining the global view of the ants with a local path-untangling strategy, will solve the problem a lot more efficiently.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 28, 2017

1 RECOMMENDER SYSTEMS (MATRIX COMPLETION ALGORITHM)

In many of today's consumer businesses, it is common practice to provide suggestions for possible purchases based on previous orders. This type of problem is referred to as a **recommender problem**, or more popularly, as the Netflix problem, after the online movie-rental company. The Netflix problem is concerned with computing recommendations for customers, when only a few previous movie rentals and their ratings are known. From a mathematical point of view, we represent the Netflix data-base as a rectangular matrix (see Figure 1.1 for a sketch), where each row represents one customer and each column accounts for a particular movie in the Netflix collection. For each customer (row) we record an entry in the matrix that describes the rating of the rented movie (in the respective column), from 0 for the lowest rating to 5 for the highest. The entire Netflix data-base matrix is then a sparsely filled rectangular matrix of positive values. The question is then on how to fill the remaining entries of the matrix, so that Netflix can make intelligent suggestions for new rentals based on a customer's preferences in movies.

The underlying principle is to complete a matrix based on only a few entries. In the mathematical literature, this is known as the **matrix completion problem**. In general, it is an ill-determined problem, unless additional conditions are imposed on the full matrix to be constructed. For example, if we are asked to fill two missing elements (indicated by $\times$) in the matrix D

$$\mathsf{D} = \begin{bmatrix} 1 & 2 \\ \times & 6 \\ 2 & \times \end{bmatrix} \tag{1.1}$$

we would conclude that this is an ill-posed problem, i.e., we are not given sufficient information to complete the task. However, if we are asked to come up with a rank-one solution, the problem becomes trivial. The matrix D can have a maximal rank of two (two linearly independent columns). By asking for a rank-one matrix, we ask for a matrix with only one independent column; the second column is simply a multiple of the first. Looking at the first row, it becomes clear that the second column is just the double of the first column; we therefore can complete the matrix as follows.

$$\mathsf{D} = \begin{bmatrix} 1 & 2 \\ \times & 6 \\ 2 & \times \end{bmatrix} \quad \longrightarrow \quad \begin{bmatrix} 1 & 2 \\ 3 & 6 \\ 2 & 4 \end{bmatrix} \tag{1.2}$$

The additional constraint of looking for a low-rank (or rank-deficient) matrix made the problem well-posed.

We will apply a similar regularization approach to the Netflix problem. Again, we are using the rank of the recovered matrix, i.e., the number of linearly independent columns of the data matrix, as an additional condition that needs to be satisfied. In different terms, we postulate that the taste in movies can be expressed in simple terms with only a few column vectors (far less than the number of movies in the Netflix collection).
We introduce the notation Ω which denotes the set of (i, j) indices where we have data (all the ratings in Figure 1.1). The ratings themselves are given by N_{ij}.
We can then formulate the recommender problem as follows: we wish to recover a matrix A by solving the following optimization problem:

$$\min \operatorname{rank}(\mathsf{A}), \tag{1.3a}$$
$$\text{subject to} \quad \mathsf{A}_{ij} = \mathsf{D}_{ij} \qquad \{i, j\} \in \Omega, \tag{1.3b}$$

i.e., we minimize the rank of the recovered matrix A while respecting the set Ω of known entries D_{ij}.
The above optimization problem is difficult due to the fact that the rank of a matrix (the number of linearly independent columns or rows) takes on integer values, making the objective in the optimization problem discontinuous, and the optimization problem non-convex. It is advantageous to replace the rank condition by a continuous analog that still measures the rank; this condition is known as the nuclear norm of A, denoted by $\|\mathsf{A}\|_*$. The nuclear norm of a matrix A is defined as the *sum* of its singular values, while the rank is the *number* of non-zero singular values.
The new (convexified) optimization problem is then

$$\min \|\mathsf{A}\|_*, \tag{1.4a}$$
$$\text{subject to} \quad \mathsf{A}_{ij} = \mathsf{D}_{ij} \qquad \{i, j\} \in \Omega, \tag{1.4b}$$

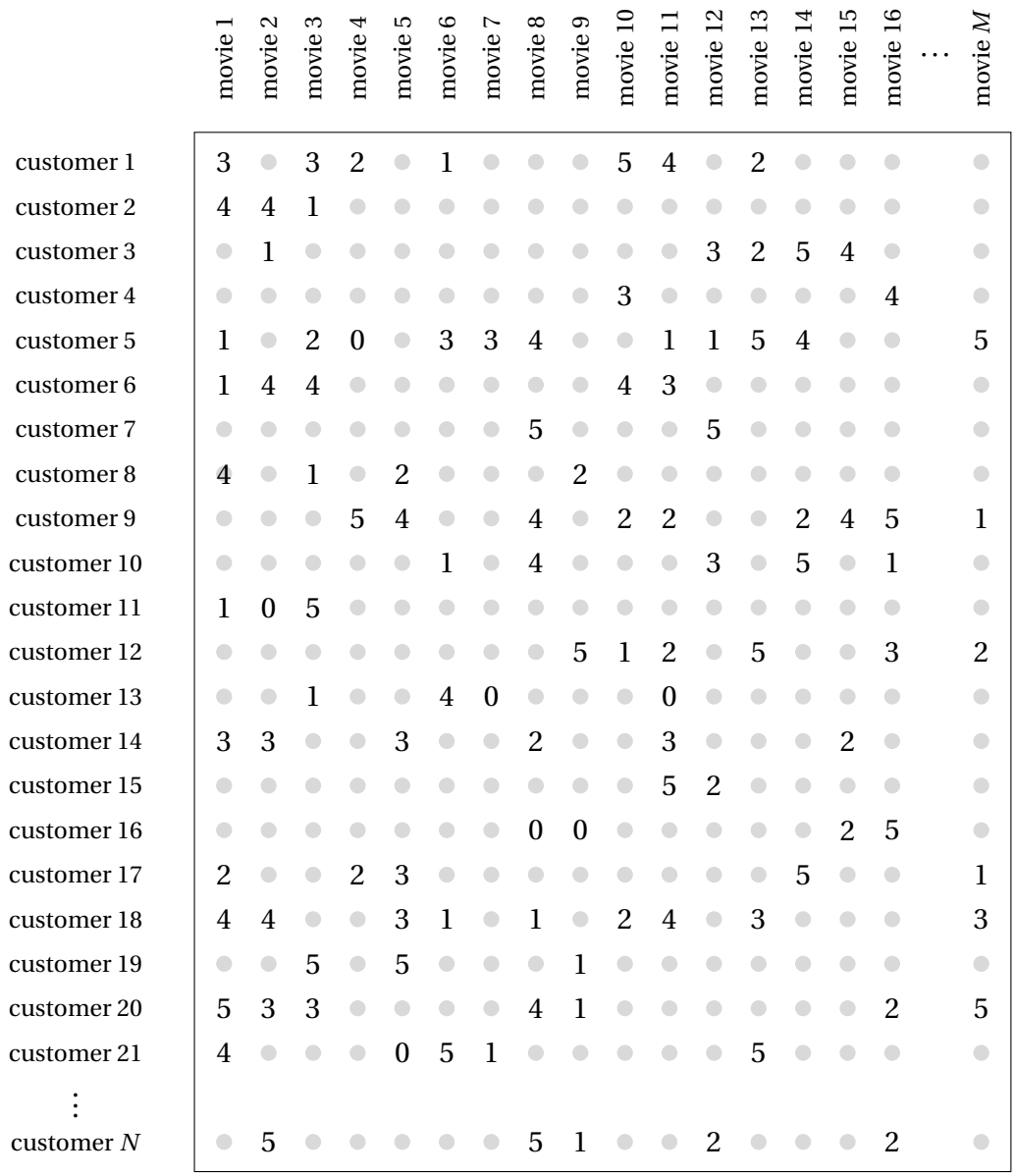

Figure 1.1: Example of a data matrix D for a recommender system.

i.e., we minimize the nuclear norm of the recovered matrix A while, as before, observing the set Ω of known entries D_{ij}.

It seems obvious that rank, singular values, nuclear norm, etc. will feature prominently in the algorithm. For this reason, we will present a brief review of the singular value decomposition, which underlies these concepts.

1.1 INTERLUDE: THE SINGULAR VALUE DECOMPOSITION

The singular value decomposition, or SVD for short, is an essential decomposition of a general (rectangular) matrix. Another, maybe more familiar decompositions, is the eigenvalue decomposition of a square matrix. In general, decompositions are very useful tools to categorize, quantify or analyze matrices. They are based on the following question: how can a matrix be described by its action on a set of vectors? In this sense, a decomposition constitutes an input-output analysis for a matrix. We know that acting with a matrix on a vector will, in general, rotate and stretch the vector. Decompositions are a way of isolating the rotation from the stretching part. Stretching a vector is simply a multiplication by a scalar; and stretching a collection of column vectors can be expressed as a multiplication by a diagonal matrix (with the individual stretching factors along the diagonal).

The singular value decomposition proposes the following split: we transform the vectors of an orthogonal input basis to an orthogonal output basis. This transformation contains additional stretching of the individual vectors that we express as a multiplication by a diagonal matrix. The rotational part of the transformation is contained in the fact that the input basis is not the same as the output basis, but rather the output basis is rotated with respect to the input basis. A sketch of the decomposition is given in Figure 1.2.

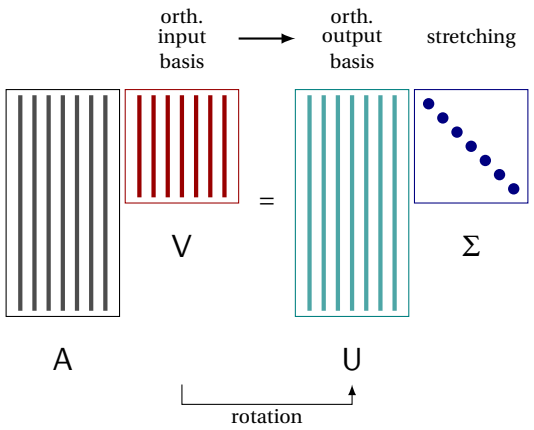

Figure 1.2: The singular value decomposition of a matrix A is expressed as the matrix' action on an orthogonal input matrix V. This actions (right-hand side) is broken into a rotated, orthogonal output basis U and a diagonal matrix Σ that accounts for the stretching of the individual basis vectors.

In two dimensions and for a 2×2-matrix A, the sketch in Figure 1.3 illustrates the action of the matrix A : we transform the orthogonal (and normalized) input basis (the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$) into the orthogonal basis (the vectors $\mathbf{u}_1$ and $\mathbf{u}_2$) with additional stretching factors σ_1 and σ_2. Under this mapping and representation, circles in the $\mathbf{v}_1$-$\mathbf{v}_2$-coordinate system turn into ellipses in the $\mathbf{u}_1$-$\mathbf{u}_2$-coordinate system. The stretching factors (singular values $\sigma_{1,2}$) determine the aspect ratio (eccentricity) of the ellipse. In the special case of $\sigma_2 = 0$, the ellipse degenerates to a line.

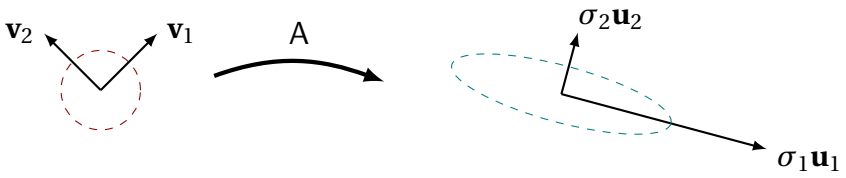

Figure 1.3: Mapping of a 2-dimensional orthogonal basis under A onto a rotated, orthogonal, stretched basis. Circles in the input basis map to ellipses in the output basis.

The stretching factors are referred to as **singular values**, are real, non-negative numbers and – by convention – are presented in descending order. Singular values can be zero; in this case, certain vectors from the input space get mapped to zero in the output space, i.e., they vanish. In other words, zero singular values are associated with a loss of "dimensionality" of the matrix A. The number of nonzero singular values is referred to as the **rank** of a matrix.

A simple reordering of the singular value decomposition allows us to decompose *any* matrix into its input-output formulation. For this, we transfer the input-matrix V to the right-hand side. More mathematically, we have

$$\mathsf{AV} = \mathsf{U}\Sigma \qquad \rightarrow \qquad \mathsf{A} = \mathsf{U}\Sigma\mathsf{V}^H \tag{1.5}$$

where U and V are orthogonal (or unitary) matrices containing the output and input basis vectors as columns, and Σ is a diagonal matrix with the stretching factors (ordered singular values) along the diagonal. The second expression above uses the fact that the inverse of an orthogonal (unitary) matrix is simply given by its conjugate transpose (indicated by ^{H}). Breaking the resulting product of three matrices $\mathsf{U}\Sigma\mathsf{V}^H$ into its individual vector components, we arrive at an interesting representation: it says that any matrix A can be written as the **sum of rank-one matrices**. See Figure 1.4.

We have

$$\mathsf{A} = \mathbf{u}_1\sigma_1\mathbf{v}_1^H + \mathbf{u}_2\sigma_2\mathbf{v}_2^H + \cdots + \mathbf{u}_m\sigma_m\mathbf{v}_m^H \tag{1.6}$$

with m as the number of columns in A (which we assume to be smaller than the number of rows, i.e., A is a tall-and-skinny matrix). Each one of these components on the right-hand side is a rank-one matrix, since the same column vector $\mathbf{u}_i$ is multiplied by the elements in $\mathbf{v}_i$ and σ_i to form a matrix; in the end, each resulting matrix $\mathbf{u}_i\sigma_i\mathbf{v}_i^H$ contains just stretched copies of the first column.

With the singular value decomposition we can thus determine the rank of the matrix A by looking at the singular values σ_i. The rank is given by the number of non-zero singular values. For example, a matrix with $\Sigma = \mathrm{diag}\{\sigma_1, \sigma_2, \sigma_3, 0, 0, \ldots, 0\}$ with $\sigma_{1,2,3} \neq 0$ has rank three. In this case, the above rank-one expansion (1.6) terminates after three terms.

Even if the singular values do not exactly drop to zero, we can still give an optimal rank-three approximation to a matrix A using the singular value decomposition. We get

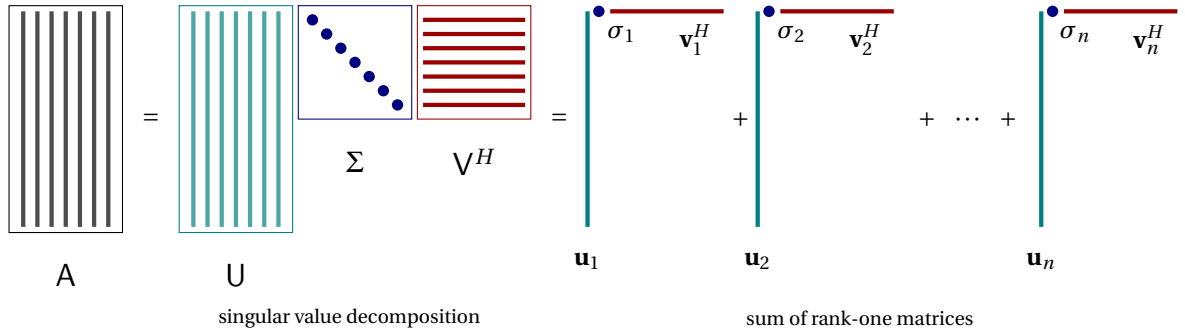

Figure 1.4: Singular value decomposition of a rectangular matrix A resulting into a product of three matrices: the output basis U, the stretching factors Σ and the output basis V. On the right: a representation of the decomposition as a sum of rank-one matrices $\mathbf{u}_i \sigma_i \mathbf{v}_i^H$.

$$\mathsf{A} \approx \mathbf{u}_1\sigma_1\mathbf{v}_1^H + \mathbf{u}_2\sigma_2\mathbf{v}_2^H + \mathbf{u}_3\sigma_3\mathbf{v}_3^H = \mathsf{U}_{(1:3)}\Sigma_{(1:3)}(\mathsf{V}_{(1:3)})^H. \tag{1.7}$$

We will take advantage of this low-rank truncation, based on the SVD, in the Netflix or matrix-completion algorithm below. From the expressions above, we conclude that the rank (given by the singular values) is a proxy for the "information content" of a matrix. Even though a low-rank matrix can consist of many entries (degrees of freedom), its action can be described rather low-dimensionally.

1.2 The mathematics of the optimization problem

Continuing with the matrix completion problem, we introduce an operator $\pi_\Omega(\cdot)$ which, when operating on a matrix, extracts the known entries $\{i, j\} \in \Omega$ from the matrix. We then have

$$\min \|\mathsf{A}\|_*, \tag{1.8a}$$
$$\text{subject to} \quad \pi_\Omega(\mathsf{A}) = \pi_\Omega(\mathsf{D}) \tag{1.8b}$$

or, introducing the matrix E which recovers the missing entries while leaving the true entries untouched, we have

$$\min \|\mathsf{A}\|_*, \tag{1.9a}$$
$$\text{subject to} \quad \mathsf{A} + \mathsf{E} = \mathsf{D} \quad \text{and} \quad \pi_\Omega(\mathsf{E}) = 0. \tag{1.9b}$$

Since the matrix E will recover the missing entries of D, these entries of D are simply set to zeros.

The above problem is a constrained optimization problem: we try to minimize the nuclear norm (a continuous proxy for rank) subject to matching the known entries in D. We transform the constrained problem to an unconstrained problem by adding the constraints to the cost functional using Lagrange multipliers. We then seek to minimize the augmented expression

$$\mathscr{L}(\mathsf{A},\mathsf{E},\mathsf{Y},\mu) = \|\mathsf{A}\|_* \; + \; \langle \mathsf{Y}, \mathsf{D}-\mathsf{A}-\mathsf{E}\rangle \; + \; \mu\|\mathsf{D}-\mathsf{A}-\mathsf{E}\|_F^2. \tag{1.10}$$

We see that we have added the constraint $\mathsf{D}-\mathsf{A}-\mathsf{E}=0$ via the Lagrange multiplier Y and the scalar product $\langle\cdot,\cdot\rangle$ to the objective $\|\mathsf{A}\|_*$. The last expression is introduced to turn the optimization problem into a convex optimization problem; this will have advantageous for the algorithm. This procedure is also referred to as a relaxation of the optimization problem. The parameter μ is very small and adaptive and is introduced as an unknown variable (to be optimized). The subscript $_F$ on the last expression denotes the Frobenius norm. Ideally, $\mathsf{D}-\mathsf{A}-\mathsf{E}$ should be very small; so the last expression is negligible near the optimum and does not significantly influence the final solution. The last constraint $\pi_\Omega(\mathsf{E})=0$ is not explicitly enforced, but rather needs to be imposed at every step of the algorithm.

The above unconstrained optimization problem needs to be solved by finding a set of independent variables $\mathsf{A},\mathsf{E},\mathsf{Y}$ and μ that minimize the augmented expression. Gradients with respect to all four variables, together with a standard gradient-based algorithm, would direct the optimization process. However, we choose a different strategy which employs the idea of alternating directions. To this end, we solve a sequence of simpler optimization problems by freezing three of the four variables and optimizing with respect to the last one (see Figure 1.5). By cyclically rotating through all four variables (always freezing three and optimizing with respect to the other), we iteratively approach the final solution that minimizes the augmented (composite) expression.

With four independent variables, each iteration of the algorithm involves four local optimizations. First we optimize A, while keeping E, Y and μ constant.

$$\mathsf{A}_{k+1} = \arg\min_{\mathsf{A}} \; \mathscr{L}\left(\mathsf{A},\mathsf{E}_k,\mathsf{Y}_k,\mu_k\right) \qquad \text{[opt 1]} \tag{1.11}$$

where we introduced the subscript $_k$ to indicate the iteration count. Next, we optimize with respect to the variable E. We get

$$\mathsf{E}_{k+1} = \arg\min_{\mathsf{E}} \; \mathscr{L}\left(\mathsf{A}_{k+1},\mathsf{E},\mathsf{Y}_k,\mu_k\right) \qquad \text{[opt 2]} \tag{1.12}$$

where we took the updated iterate A_{k+1} from the previous step. The third step is a simple update: we determine the mismatch between the new $\mathsf{A}_{k+1}+\mathsf{E}_{k+1}$ and D for the elements in the complementary set of indices $\overline{\Omega}$ to update Y_k to Y_{k+1}. We notice that only indices of the complementary set are non-zero in Y, since only these indices have to be adjusted. We have

$$\mathsf{Y}_{k+1} = \mathsf{Y}_k + \mu_k\left(\mathsf{D}-\mathsf{A}_{k+1}-\mathsf{E}_{k+1}\right). \qquad \text{[opt 3]} \tag{1.13}$$

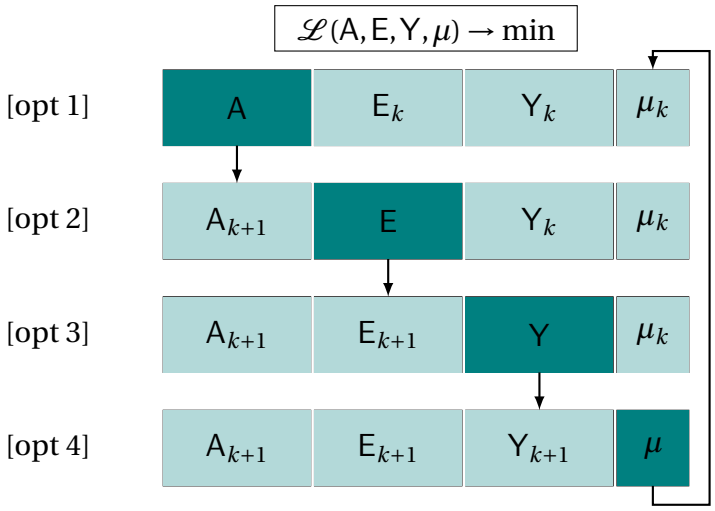

Figure 1.5: Sketch of iterative scheme of the incomplete alternating Lagrangian method (IALM).

The fourth and final step will update the parameter μ_k to μ_{k+1}. A heuristic dependence on the iteration counter will be used to approach a steady (and small) value of the regularization parameter μ. We then return to the first step.

The first optimization problem, where we determine a new A matrix, is given in generic form

$$\mathsf{X} = \arg\min_{\mathsf{X}} \ \epsilon\|\mathsf{X}\|_* + \frac{1}{2}\|\mathsf{X}-\mathsf{W}\|_F^2 \tag{1.14}$$

which is a mixed-norm optimization problem with one term covering the nuclear norm (the low-rank objective) and another term implementing the Frobenius-norm regularization. The solution to this optimization problem can be obtained using the singular value decomposition (SVD). Without proof, we give the solution of the above optimization as

$$\mathsf{X} = \mathsf{U}\ \mathrm{shrink}_\epsilon(\Sigma)\ \mathsf{V}^H \tag{1.15}$$

where $\mathsf{U}, \Sigma, \mathsf{V}$ are the three components of an SVD of W. The shrink-function (see Figure 1.6) is given as

$$\mathrm{shrink}_\epsilon(x) = \begin{cases} x-\epsilon, & \text{if } x > \epsilon \\ x+\epsilon, & \text{if } x < -\epsilon \\ 0, & \text{otherwise} \end{cases} \tag{1.16}$$

The solution to the (regularized) optimization problem, i.e. matching a matrix X of minimal rank to a given matrix W, is thus given by a thresholded singular value decomposition. Applied to our recommender system problem, we have to determine the A_{k+1} in the first subproblem as follows,

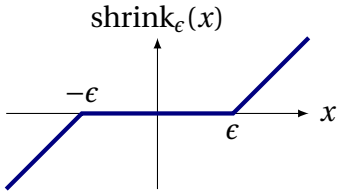

Figure 1.6: Sketch of the shrink-function for soft thresholding of the singular values.

$$\begin{aligned} \operatorname{svd}\left(\mathsf{D}-\mathsf{E}_k+\mu_k^{-1}\mathsf{Y}_k\right) &= \mathsf{U}\Sigma\mathsf{V}^H && (1.17a)\\ \mathsf{A}_{k+1} &= \mathsf{U}\,\operatorname{shrink}_{\mu_k^{-1}}(\Sigma)\,\mathsf{V}^H \qquad [\text{opt 1}] && (1.17b)\end{aligned}$$

i.e., we take a singular value decomposition, apply the shrink-function to the singular values with a cut-off threshold of μ_k^{-1}, and then reconstitute the three-part decomposition with a replaced diagonal matrix $\operatorname{shrink}_{\mu_k^{-1}}(\Sigma)$. This part of the algorithm will aim for a low-rank matrix A.

With a new matrix A_{k+1} we solve the second optimization problem (1.12). This is simply accomplished by a projection using $\pi_{\overline{\Omega}}$ onto the

$$\mathsf{E}_{k+1}=\pi_{\overline{\Omega}}\left(\mathsf{D}-\mathsf{A}_{k+1}+\mu_k^{-1}\mathsf{Y}_k\right). \qquad [\text{opt 2}] \qquad (1.18)$$

For this reason, the new matrix E_{k+1} has only non-zero entries at indices where we need to recover missing data (and zeros at indices where we do have data).

The third optimization step, which yields a new value for the Lagrange multiplier Y_{k+1}, is already given by (1.13). There is no need to project again, since E_{k+1} has non-zero entries for indices that need to be recovered.

Finally, the threshold and regularization value μ_k has to be updated. This last step concludes one iteration. We return to the first subproblem or terminate if a user-specified criterion is satisfied.

Two important remarks are in order:

1. Sparsity. For each $k \geq 0$, Y_k vanishes outside of Ω and is, therefore, sparse, a fact which can be used to evaluate the shrink-function rapidly.

2. Low-rank property. The matrix A_k turns out to have low rank, and hence the algorithm has minimum storage requirement, since we only need to keep principal factors in memory.

1.3 Algorithm

We formulate an augmented Lagrange multiplier algorithm which we will solve by alternating between four optimization sub-problems until convergence is reached.

For our matrix completion problem, we try to recover missing entries in the matrix by using the assumption of a low-rank property of the underlying (complete) matrix. Problems of this type not only appear in recommender systems, but also in machine learning, control applications and computer vision.
The full algorithm loops through the following steps:

1. initialize all matrices and parameters.
2. compute the singular value decomposition of $D - E_k + \mu_k^{-1} Y_k$ and apply the shrink-function to the singular values with a threshold of μ_k^{-1}. The solution of the first optimization subproblem is the reconstituted singular value decomposition with the thresholded singular values.
3. compute the update for E_{k+1} according to $\pi_{\overline{\Omega}}(D - A_{k+1} - Y_k/\mu_k)$, i.e. a projection onto the complementary index set.
4. update the Lagrange multiplier matrix Y_{k+1} according to $Y_k + \mu_k(D - A_{k+1} - E_{k+1})$.
5. adjust the parameter μ_k.
6. go back to step 1 or terminate if proper convergence conditions are met.

1.4 Implementation

The implementation of the above algorithmic steps is given below.

```
1  import numpy as np
2  import math as ma
3
4  def IALM(D,mu,rho):
5      # incomplete alternating Lagrangian
6      # method (IALM) for solving the
7      # matrix completion problem
8
9      # thresholds
10     ep1 = 1.e-7
11     ep2 = 1.e-6
12     Dn  = np.linalg.norm(D,'fro')
13     # projector matrix (complementary)
14     PP = (D == 0)
15     P  = PP.astype(np.float)
16     # initialization
17     m,n  = np.shape(D)
18     Y    = np.zeros((m,n))
19     Eold = np.zeros((m,n))
20     # iteration
```

```
    for i in range(1,1000):
        # compute SVD
        tmp   = D - Eold + Y/mu
        U,S,V = np.linalg.svd(tmp,full_matrices=False)
        # threshold and patch matrix back together
        ss = S-(1/mu)
        s2 = np.clip(ss,0,max(ss))
        A  = np.dot(U,np.dot(np.diag(s2),V))
        # project
        Enew = P*(D - A + Y/mu)
        DAE  = D - A - Enew
        Y   += mu*DAE
        # check residual and (maybe) exit
        r1   = np.linalg.norm(DAE,'fro')
        resi = r1/Dn
        print i,' residual ',resi
        if (resi < ep1):
            break

        # adjust mu-factor (heuristic)
        muf  = np.linalg.norm((Enew-Eold),'fro')
        fac  = min(mu,ma.sqrt(mu))*(muf/Dn)
        if (fac < ep2):
            mu *= rho

        # update E and go back
        Eold = np.copy(Enew)

    E = np.copy(Enew)
    return A,E
```

The main code is given as

```
if __name__ == '__main__':

    # matrix completion problem (Netflix problem)

    # size of problem
    m = 500                     # rows
    n = 150                     # columns
    r = int(round(min(m,n)/3))  # rank
    # construct low-rank matrix
    U = np.random.random((m,r))
    V = np.random.random((r,n))
    A = np.dot(U,V)
```

```
13      # sampling matrix
14      PP = (np.random.random((m,n)) > 0.433)
15      P  = PP.astype(np.float)
16      # number of non-zero elements
17      Omega = np.count_nonzero(P)
18      # data matrix
19      D = P*A
20      fratio = float(Omega)/(m*n)
21      print 'fill ratio ', fratio
22      # initialize parameters
23      mu  = 1./np.linalg.norm(D,2)
24      rho = 1.2172 + 1.8588*fratio
25      # call IALM-algorithm
26      AA,EE = IALM(D,mu,rho)
27      # compare
28      print('\n')
29      print('Data matrix')
30      print D[0:5,0:5]
31      print('\n')
32      print('Recovered matrix')
33      print AA[0:5,0:5]
34      print('\n')
35      print('Original matrix')
36      print A[0:5,0:5]
```

1.5 Example

The recovery of the missing data entries of a low-rank matrix depends on the number of known entries. Theory tells us that the number of known entries p must follow the inequality $p \geq Cr\,n^{6/5}\ln n$ where n denotes the number of columns, r stands for the rank and C represents a positive constant.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

March 7, 2017

1 EIKONAL EQUATION (FAST MARCHING ALGORITHM)

1.1 THE EIKONAL EQUATION

Interface problems are very common in science and engineering, ranging from multiphase fluid problems to medical imaging. Interfaces, fronts and edges are often represented by an indicator function that measures the minimal distance from any point in space to the interface. The interface is then defined as the zero-isosurface of the higher-dimensional distance function.

If we define a propagating interface implicitly as $u(\mathbf{x}(t),t) = 0$, the chain rule produces a propagation equation for u according to

$$\frac{\partial u}{\partial t} + \nabla u \cdot \frac{d\mathbf{x}}{dt} = 0. \tag{1.1}$$

We assume that the front propagates in the direction of the local normal $\mathbf{n}$ with a speed F which yields

$$\frac{d\mathbf{x}}{dt} \cdot \mathbf{n} = F \qquad\qquad \mathbf{n} = \frac{\nabla u}{|\nabla u|}. \tag{1.2}$$

Substituting this expression into the evolution equation we arrive at

$$\frac{\partial u}{\partial t} + F|\nabla u| = 0. \tag{1.3}$$

The steady-state of the above equation then produces the eikonal equation

$$F|\nabla u| = 1 \tag{1.4}$$

which describes an equation for the travel time of information through a "speed" field F. The speed field represents the local traveling-wave speed and influences the time of travel from any point to any other point. The travel time information, can be used in a multitude of application, among them seismic exploration, optimal path planning and robotics. Constant values of u represent surfaces of constant phase information or the wave fronts; the normals to the same surfaces represent wave rays.
For a special case of $F = 1$, the eikonal equation reduces to a governing equation for a distance function

$$|\nabla u| = 1 \tag{1.5}$$

which states that the absolute value of the gradient is constant. Note that the absolute value is necessary to express that distance from the interface is independent of the direction we march.

The eikonal equation constitutes a nonlinear first-order partial differential equation, which can be solved by the method of characteristics or a wave-propagation approach. Special emphasis has to be drawn to crossing characteristics, in which case, we are interested in a regularized solution or a viscosity solution.

Typically, the eikonal equation is solved starting from an initial condition $u(x) = 0$ for $x \in \mathscr{I}$ which stated that along a given set $\mathscr{I}$ the travel time is zero. From there, we propagate local wave solutions omnidirectionally and thus advance a front of constant travel time. This propagation is contingent on the presence of obstacles, other geometric constraints and the speed field (which sets the rate of front propagation). The final solution then presents the minimal travel time from any point on the initial set $\mathscr{I}$ to any point in the rest of the domain.

This travel time information can be used in multiple ways: (i) in seismic exploration it is used to reconstruct the material layering in the earth's crust from localized measurements; (ii) in medical applications, it is used to segment medical images to isolate abnormal from normal tissue; (iii) in robotics, it is used to solve problems of optimal path and motion planning. In the latter case, the travel time field is used to determine geodesic lines, i.e., lines of minimal distance between two points on a general manifold and/or under a varying propagation speed. The geodesic lines can simply be extracted from the computed travel-time field u according to

$$\frac{d\mathbf{x}}{dt} = -\frac{\nabla u}{|\nabla u|}. \tag{1.6}$$

This states that, starting with the end point of the geodesic line, we integrate backwards along a path of minimal travel time to the starting time, i.e., a point in the set $\mathscr{I}$. In robotics, the vector $\mathbf{x}$ can contain not only Cartesian coordinates, but also rotational angles. For the motion of a multi-degree-of-freedom system (for example, a complex robot arm), the eikonal equation to be solved and subsequently integrated can be rather high-dimensional, and special numerical techniques have to be applied.

In this section, we will concentrate on the two-dimensional Cartesian case, but will structure our `python`-program to allow a straightforward extension to the higher-dimensional case.

1.2 Discretization

We use a highly efficient numerical method to solve the eikonal equation, the Fast Marching Method (FMM). In its core, it is closely related to Dijkstra's method, albeit applied to a Cartesian mesh rather than a graph. We exploit the nature of the eikonal equation and solve it by propagating a front (from one or multiple starting positions) while observing the governing equation. The method calculates the time u it takes for a wave to reach every point in our computational domain. We assume that the front propagates normal to the front.

The start with a Cartesian grid, with i, j denoting the i-th grid point in the x-direction and j-th grid point in the y-direction. The notation u_{ij} then stands for the value of u at the grid point (i, j). We next introduce the first-order gradient approximations

$$D_{ij}^{-x}u = \frac{u_{ij} - u_{i-1,j}}{\Delta x}, \tag{1.7a}$$

$$D_{ij}^{+x}u = \frac{u_{i+1,j} - u_{ij}}{\Delta x}, \tag{1.7b}$$

$$D_{ij}^{-y}u = \frac{u_{ij} - u_{i,j-1}}{\Delta y}, \tag{1.7c}$$

$$D_{ij}^{+y}u = \frac{u_{i,j+1} - u_{ij}}{\Delta y}, \tag{1.7d}$$

where Δx and Δy stand for the grid spacing in the x- and y-direction, respectively. Due to the absolute value in the governing equation and the fact that we can have discontinuous gradients in the solution, it is advantageous and necessary to slope-limit the gradients based on the options above. In effect, we have to take the larger of two gradients in each direction, while avoiding negative gradients altogether. The avoidance of negative gradients stems from the fact that travel time away from the initial set $\mathscr{I}$ can only increase. We choose as our implemented discretization the scheme

$$\max\left(D_{ij}^{-x}u, -D_{ij}^{+x}u, 0\right)^2 + \max\left(D_{ij}^{-y}u, -D_{ij}^{+y}u, 0\right)^2 = 1/F_{ij}^2. \tag{1.8}$$

After a bit of algebra, we arrive at

$$\max\left(\frac{u-u_1}{\Delta x}, 0\right)^2 + \max\left(\frac{u-u_2}{\Delta y}, 0\right)^2 = 1/F_{ij}^2 \tag{1.9}$$

with the definitions

$$u = u_{ij} \tag{1.10a}$$

$$u_1 = \min(u_{i-1,j}, u_{i+1,j}) \tag{1.10b}$$

$$u_2 = \min(u_{i,j-1}, u_{i,j+1}) \tag{1.10c}$$

The above equation (1.9) leads to a quadratic equation for u, the solution at the new point. We have

$$\left(\frac{u-u_1}{\Delta x}\right)^2 + \left(\frac{u-u_2}{\Delta y}\right)^2 = \frac{1}{F_{ij}^2} \tag{1.11}$$

with the solution

$$u = -\frac{\alpha}{2} + \frac{1}{2}\sqrt{D} \tag{1.12}$$

and

$$\alpha = -2\frac{\Delta y^2 u_1 + \Delta x^2 u_2}{\Delta x^2 + \Delta y^2}, \tag{1.13a}$$

$$\beta = \frac{\Delta y^2 u_1^2 + \Delta x^2 u_2^2 - \Delta x^2 \Delta y^2 / F_{ij}^2}{\Delta x^2 + \Delta y^2}, \tag{1.13b}$$

$$D = \alpha^2 - 4\beta. \tag{1.13c}$$

It depends on the distribution of u-values in the immediate neighborhood of the grid point (i, j). Special treatment includes the case where one of the terms in (1.9) vanishes, in which case we have a simple linear equation (and a corresponding planar wave propagation):

$$u = u_1 + \frac{\Delta x}{F_{ij}} \qquad \text{or} \qquad u = u_2 + \frac{\Delta y}{F_{ij}}. \tag{1.14}$$

For the algorithm, we propagate a wave front, starting from an inital location, by solving (1.9). For this we divide the grid points into `Accepted`, `NarrowBand` and `UnVisited`. The `Accepted` points are grid points where the travel time u_{ij} has converged to the final value. Initially, only the starting point is in the `Accepted` group, with a value of $u_{ij} = 0$. The `UnVisited` group are grid points that still need to be computed. In the beginning, all but the initial grid point are `UnVisited`. The `NarrowBand` category is among the `UnVisited` group, but is in the neighborhood of the `Accepted` points. The `NarrowBand` points form a thin layer around the `Accepted` points.

With these definitions, the algorithm to solve the above equation proceeds in the steps listed below (following sketch outlined in figure 1.1).

`step0` Initially, see figure 1.1(a), the only `Accepted` point is the initial point, with $u_{ij} = 0$. The u-value for all other grid points is set to a very large value.

`step1` We determine the neighborhood of the `Accepted` points, see the blue points in figure 1.1(a), and add them to the `NarrowBand` list. For each point in the `NarrowBand` list we solve the eikonal equation (i.e., the quadratic or degenerate linear equation). We choose the grid point in the `NarrowBand`-list with the smallest travel time u (indicated by the circles symbol in figure 1.1(a)) and transfer the corresponding grid point from the `NarrowBand`- to the `Accepted`-list. The associated value of the travel time is the final value.

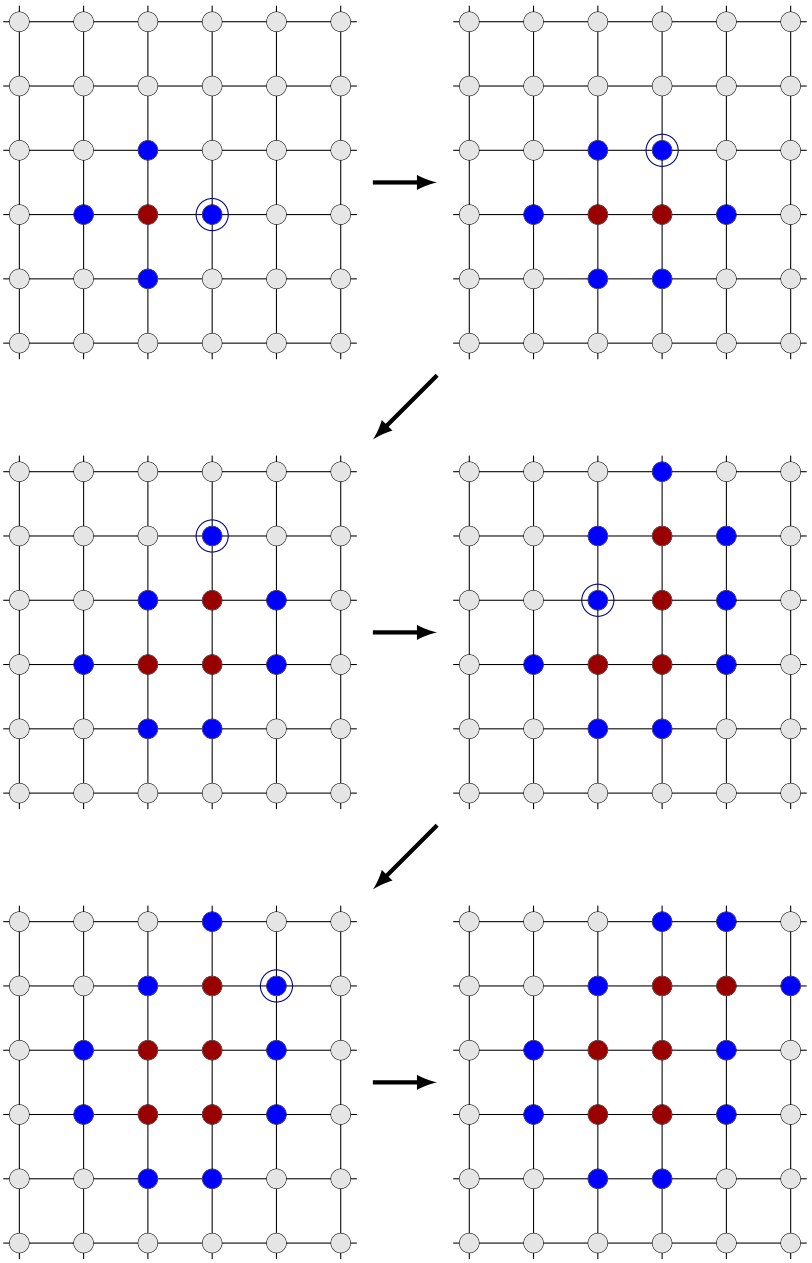

Figure 1.1: Sketch of marching a distance-function front from an initial condition based on the eikonal equation. Color code: (gray) `UnVisited` grid points, (red) visited and `Accepted` grid points, (blue) grid points in the `NarrowBand` area around `Accepted` grid points, (blue circled) `NarrowBand`-point with the minimum distance-function value (will be `Accepted` point in the next step).

`step2` We augment the `NarrowBand`-list by the neighboring point of the newly added grid point, compute the eikonal value u for the new neighbors and continue with the algorithm above (see figure 1.1(b-f)).

step3 The algorithm terminates, when no neighboring points of the Accepted grid points can be found.

The code listing for the python-implementation is listed below. It consists of three functions: the first develops an index list between every point of the computational domain and its Northern, Southern, Western and Eastern neighbors; the second solves the local eikonal equation for a five-point stencil consisting of the central and the Northern, Southern, Western and Eastern points; the third function advances the NarrowBand-front from an initial condition to its conclusion.

```
import numpy as np
import scipy as sp
import time

import matplotlib
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

def makeIndexList(N,M):
    # running index of grid points
    C   = np.arange(1,N*M+1).reshape(N,M)
    # zero-padding
    CN  = np.zeros((N+2,M+2),dtype=np.int)
    CN[1:1+N,1:1+M] = C
    # shift to get N,S,W,E neighbors
    C_N = CN[1:1+N,2:M+2].flatten()
    C_S = CN[1:1+N,0:M].flatten()
    C_E = CN[2:N+2,1:M+1].flatten()
    C_W = CN[0:N,1:M+1].flatten()
    # rearrange into neighbor list
    CN  = np.array(zip(C_N,C_S,C_E,C_W))
    C0  = C.flatten()
    # initialize
    iA  = np.ones(N*M, dtype=int)  # accepted points = 0, not accepted =
    iC  = np.zeros(N*M, dtype=int) # close points = 1, not close = 0
    return C0,CN,iA,iC
```

This function establishes the four neighbors of a given point in the grid. It is compiled into a zip-list containing the grid indices of the four neighbors. This way, it simplifies the code when looking for the neighbors of a given point.

```
def solveEik2 (uN,uS,uE,uW,dx,dy,f):
    # initialize
```

```
    u1  = 1.e38
    u2  = 1.e38
    u3  = 1.e38
    # grid parameters
    dx2 = dx*dx
    dy2 = dy*dy
    # minimum in each direction
    Tx  = min(uE,uW)
    Ty  = min(uN,uS)
    # case 1: no x-neighbor
    if (Tx > 1.e37):
        u1 = Ty + dy/f
    # case 2: no y-neighbor
    if (Ty > 1.e37):
        u2 = Tx + dx/f
    # case 3: both x-neighbor and y-neighbor are Alive
    if ( (Tx < 1.e38) and (Ty < 1.e38) ):
        alpha = -2*(dy2*Tx + dx2*Ty)/(dx2 + dy2)
        beta  = (dy2*Tx*Tx + dx2*Ty*Ty - dx2*dy2/(f*f))/(dx2 + dy2)
        discri = alpha*alpha - 4*beta
        if (discri > 0):
            u3 = (-alpha + np.sqrt(discri))/2
    return min(u1,u2,u3)
```

This function solves the local eikonal equation, based on the quadratic equation above. For the special cases of no x-neighbor or no y-neighbor, a linear equation is substituted.

```
def eikonal(f,N,M,dx,dy,xS,yS):
    # make index list C0 and neighbor list CN
    C0,CN,iA,iC = makeIndexList(N,M)
    # initialize narrow-band arrays
    iNB       = np.empty(0, dtype=int)
    NB        = np.empty(0, dtype=float)
    # domain size
    Lx        = (N-1)*dx
    Ly        = (M-1)*dy
    # starting point
    nS        = int(np.floor(xS/Lx*N))
    mS        = int(np.floor(yS/Ly*M))
    kk        = (nS-1)*M + mS
    # initialization
    uBig      = 1.e38
    u         = np.ones(N*M)*uBig
    u[kk-1]   = 0
```

```
18      iA[kk-1] = 0
19      # declare all obstacles (with f=0) "accepted"
20      # so they won't be updated
21      iA       = iA*f
22      iA       = np.array((iA!=0), dtype=int)
23
24      # main loop for eikonal equation
25      ic = 0
26      while ((len(iNB)>0) or (ic==0)):
27          # compute the non-accepted neighbors
28          nei  = CN[kk-1]          # (1) get neighbors
29          nei  = nei*iA[nei-1]     # (2) eliminate accepted points
30          nei  = nei[nei!=0]       # (3) eliminate zeros
31          neiC = nei*iC[nei-1]     # (4) check for already close neighbors
32          neiC = neiC[neiC!=0]
33          for i in range(len(neiC)): # (5) eliminate already close neighbor
34              ii  = neiC[i]
35              jj  = np.where(iNB==ii)[0][0] # search for already close neig
36              iNB = np.delete(iNB,jj)       # eliminate them in iNB and NB
37              NB  = np.delete(NB,jj)
38          iC[nei-1] = 1            # flag new neighbors in close-array
39          # compute the u for the indices in nei
40          for i in range(len(nei)):
41              ii  = nei[i]
42              iNB = np.append(iNB,ii)
43              [No,So,Ea,We] = CN[ii-1]
44              uNo = (1.e37 if (No==0) else u[No-1])
45              uSo = (1.e37 if (So==0) else u[So-1])
46              uEa = (1.e37 if (Ea==0) else u[Ea-1])
47              uWe = (1.e37 if (We==0) else u[We-1])
48              ff  = f[ii-1]
49              uu  = solveEik2(uNo,uSo,uEa,uWe,dx,dy,ff)
50              NB  = np.append(NB,uu)
51          # determine the minimum u-value in narrow band
52          umin,imin = NB.min(),NB.argmin()
53          # next index
54          kk       = iNB[imin]
55          # delete from narrow band
56          NB       = np.delete(NB,imin)
57          iNB      = np.delete(iNB,imin)
58          # update u, iA and iC
59          u[kk-1]  = umin
60          iA[kk-1] = 0
61          iC[kk-1] = 0
```

```
62          ic      += 1
63      return u.reshape(N,M)
```

This routine is the core routine of the eikonal solver. It tracks multiple lists: the Accepted-point list `iA`, which is zero for accepted points and one for unaccepted, the Close-point list `iC`, which is one for points in the NarrowBand and zero otherwise, `iNB` for a list of indices in the NarrowBand, and `NB` for the corresponding u-value (a preliminary value that is not yet accepted). When determining the neighbors of new points, we have to eliminate zero indices (since they indicate either obstacles or the boundaries of the domain), but also previously Close points (since they have to be recomputed). Once the proper neighbor list has been compiled, the u-values are determined, and the local eikonal equation is solved (either in full quadratic form, or in one of its degenerate linear form). The minimum u-value in the `NB` is then determined: this value is then eliminated from the neighbor lists `iNB` and `NB` and transferred from the Close-list `iC` to the Accepted-list `iA`. This concludes one iteration. The iterations continue until no more neighbors can be found. In this case, all grid points have been updated and Accepted.

The main code is listed below. It tests the eikonal code on a variety of speed fields. In addition, an integration routine for tracing geodesic lines is included that allows us to determine minimal paths in the speed fields (case 0 for a test problem, cases 1 and 2 for a labyrinth).

```
1  if __name__ == '__main__':
2
3      kstride = 20              # points between parallel walls
4      N  = 15*kstride+1         # number of rows
5      M  = N                    # number of columns
6
7      # domain parameters
8      Lx = 1.                   # domain length in x
9      Ly = 1.                   # domain length in y
10
11     # mesh initialization
12     dx  = Lx/(N-1)
13     dy  = Ly/(M-1)
14     x   = np.linspace(0,Lx,N)
15     y   = np.linspace(0,Ly,M)
16     Y,X = np.meshgrid(y,x)
17
18     # slowness field
19     f0  = np.ones((N,M), dtype=float)
20     f0[:int(N/2),:int(M/2)] = 4.
21     f0[130:170,130:170]     = 0.
22     f   = f0.flatten()
23
```

```
    # starting points and other parameters
    xs     = 0.75
    ys     = 0.75
    cclip  = 1
    x0     = np.array([0.55, 0.9, 0.25, 0.1, 0.3])
    y0     = np.array([0.1,  0.1, 0.2,  0.6, 0.9])
    ncont  = 80
    nsteps = 500

    # solve eikonal equation
    t0 = time.time()
    u  = eikonal(f,N,M,dx,dy,xs,ys)
    t1 = time.time()
    print 'eikonal:  total time = ',t1-t0

    # graphics
    plt.figure()
    u1 = 0
    u  = np.clip(u,0.,cclip)
    u2 = u.max()
    du = (u2-u1)/ncont
    plt.contour(X,Y,u,levels=np.arange(u1,u2,du))
    CS = plt.contour(X,Y,u,levels=np.arange(u1,u2,du))

    for i in range(len(x0)):
        x00   = x0[i]
        y00   = y0[i]
        t0    = time.time()
        xT,yT = compGeodesics(u,dx,dy,x,y,x00,y00,nsteps)
        t1    = time.time()
        print 'geodesic: total time = ',t1-t0
        plt.plot(xT,yT,'k',linewidth=3)

    plt.axis('image')
    # plt.savefig('eik.png')
    plt.show()
```

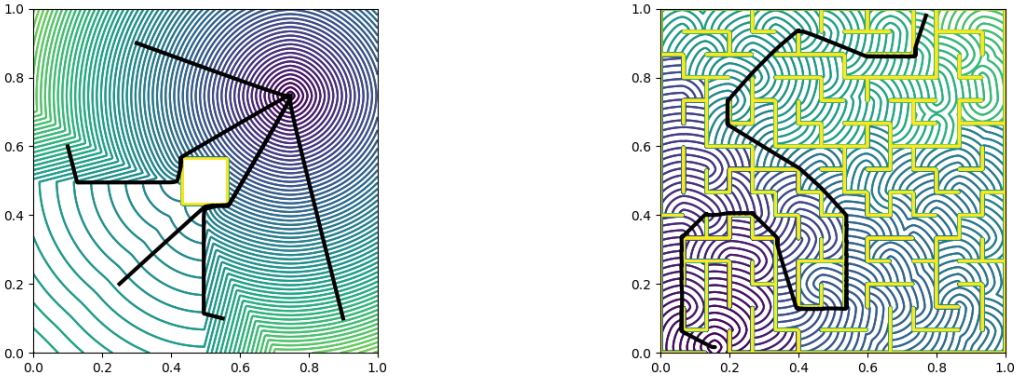

Figure 1.2: (left) Distance function: solution of the eikonal equation, using the Fast Marching Mathod, for a given speed field. Indicated are also geodesic lines in this speed fields, starting from different initial conditions and ending at the initial set $\mathscr{I}$. (right) Distance function and geodesic path for a maze.

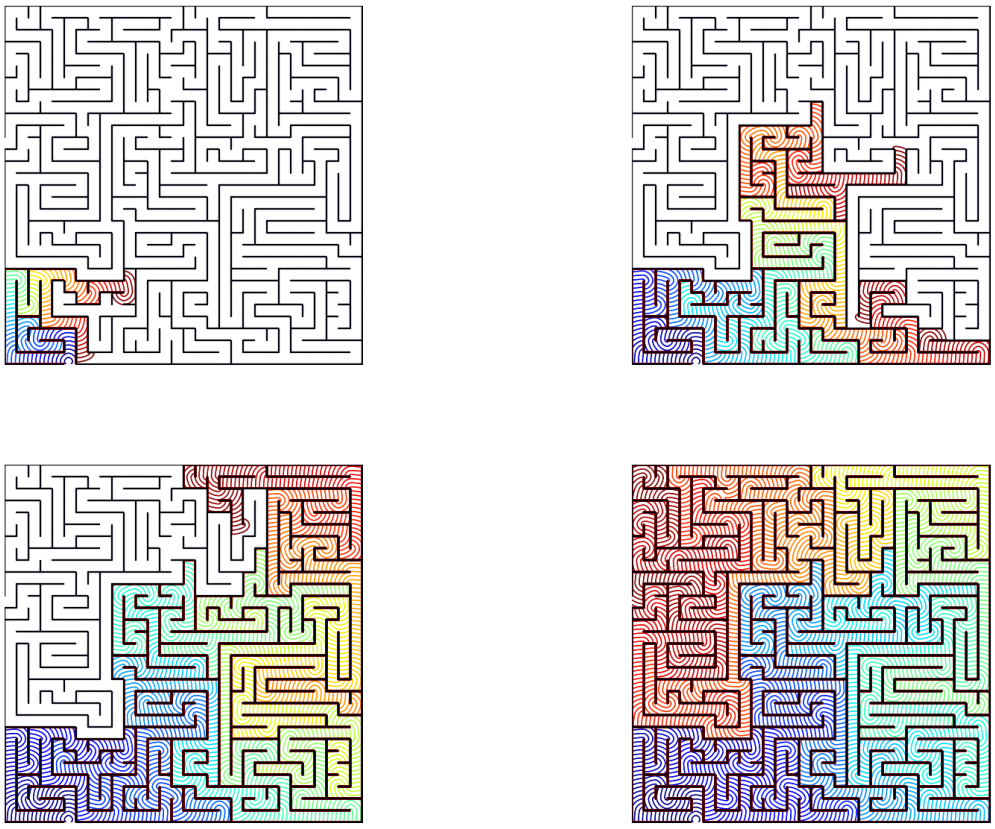

Figure 1.3: Propagation of distance function using the eikonal solver.

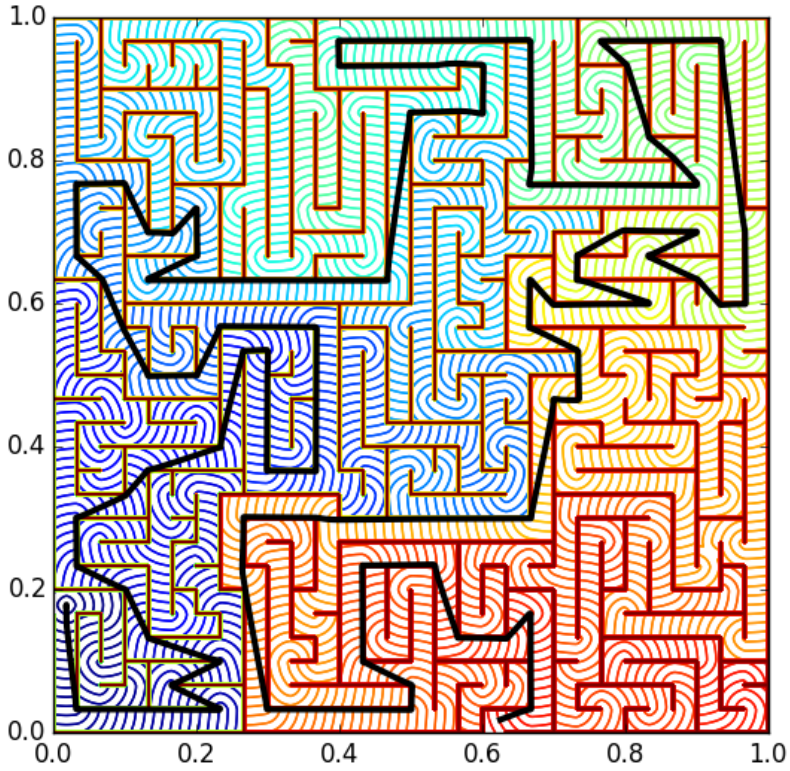

Figure 1.4: Distance function through a more complicated maze, together with a geodesic path.

IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

March 14, 2017

1 INPAINTING

Images are ubiquitous in the scientific and artistic area and are some of the oldest means of expressing information, recording historic or cultural events or displaying creative energy. The restoration of artwork is an ancient practice and has been used and perfected extensively over centuries. With the rise of digital photography and image processsing, ancient and modern techniques for graphics manipulation and repair have been encoded in particular algorithms. In this section, we will concentrate on image inpainting, a technique that attempts to restore and fill in damaged and otherwise missing parts of an image; removal of scratches, cracks, noise, stains are applications of inpainting, as are removal of stamped dates, watermarks, reflections, glare, red-eye effects or other artifacts.

Digital image inpainting requires the specification of an area of undesirable features which are to be removed and subsequently filled in based on the information available in the surrounding area of the image. This latter information is propagated (or diffused) into the removed part (referred to as the mask) until the gap is filled and the full image restored. Different algorithms and techniques are available depending on the type of information that is propagated or preserved; structure, texture, sharp gradients are among the image features that can be conserved or approximated during the process. The propagation of image information classifies the algorithms into PDE-based, convolution-based, texture-based, transform-based, etc. A broader classification rests on diffusion-based and non-diffusion-based methods. In the section below, we will explore two diffusion-based techniques: (i) isotropic diffusion and (ii) anisotropic diffusion. Both techniques will be implemented as convolution filters. The first technique applies an isotropic smoothing filter to fill in the

missing information, while the second technique, based on bilateral filters, takes into account sharp gradients and edges.
Digital image reconstruction started with the solution of partial differential equation, propagating image information into the damaged or missing area. Various types to image information are used, the most common one being the image Laplacian which represents a measure of image smoothness; the propagation direction is chosen along so-called isophotes which are perpendicular to contours of image gradients. Starting from the edge of the inpainting area, a front is then propagated into the missing domain, until the image is properly restored.

The earliest PDE-based approach is based on the heat equation

$$\frac{\partial I}{\partial t} = D\nabla^2 I, \qquad \mathbf{x} \in \Omega \tag{1.1}$$

which diffuses image information into the damaged area Ω. In this expression, $I(\mathbf{x})$ denotes the pixel value at location $\mathbf{x}$, and D stands for the diffusion coefficient (degree of smoothing). While simple to implement and sufficiently accurate for small cracks or smooth images, it has quickly been recognized that a more sophisticated approach is necessary. While clinging to the heat equation, an anisotropic diffusivity has been introduced which replaces the isotropic (direction-independent) propagation of information with a directed (anisotropic) propagation of information.

$$\frac{\partial I}{\partial t} = \nabla \cdot (a(I)\nabla I), \qquad \mathbf{x} \in \Omega \tag{1.2}$$

In this case, the image information does not diffuse isotropically or omnidirectionally into the damaged area; rather, the diffusivity coefficient $a(I)$ guides the diffusion in a manner that locally depends on the image information. Various functions for $a(I)$ are available, the most known is linked to the famous Perona-Malik equation with

$$a(I) = g\left(|\nabla I|^2\right) \qquad g(s) = \frac{1}{1+s}. \tag{1.3}$$

The local absolute image gradient thus controls the amount of diffusion into Ω. For numerical stability, the gradient has to be presmoothed, for example by a Gaussian filter. Additional constraints can be added to the propagation of image information, such as the preservation of total variation, the preservation of isophotes or identified edges.

Starting from the realization that a diffusive step can be modelled as a convolution with a (Gaussian) heat-kernel, the concept of inpainting by filtering has been born. In this approach, repeated application of a compactly supported filter window (for example, 3×3 pixels) over the missing area will diffuse information from the edge of the inpainting region to the missing pixels. These techniques are particularly attractive as they eliminate the need for the tedious (and slow) solution of complex partial differential equations; the gain in speed, however, has to be balanced by the fact that isophote directions are not preserved and high-contrast areas are excessively smoothed. Special conditions have to be implemented to account for edges, high-gradient areas, small details or particular image texture.

Independent of the applied algorithm, the image to be processed is taken as a two-dimensional numerical array of gray values, ranging from 0 (black) to 255 (white). When processing color images, the algorithm has to be applied to the R (red), G (green) and B (blue) subimages separately and then reassembled again, even though a more appropriate decomposition into luminance and two chromas is more advantageous and avoids color artifacts near edges. The damaged area to be filled is referred to as Ω and is a (small) subset of the full two-dimensional array. This subset Ω is defined by the user as a mask; the image outside this mask is left untouched by the algorithm, but will provide the starting values for filling in the damaged parts.

For large areas Ω and for images that contain a significant amount of complex texture, sophisticated algorithms have to brought to bear. For locally smaller (and thinner) regions Ω, however, simpler models can be used to restore image information from neighboring pixels.

2 Inpainting by isotropic filtering

For locally small areas Ω, we can fill the missing information, starting from the boundary $\partial\Omega$ of Ω, by an isotropic diffusion process that propagates pixel values from the edge into the area to be repaired. The diffusion process is equivalent to an iterative filtering process whereby the cleared pixels in Ω are updated using a nearest-neighbor stencil; this is equivalent to a localized convolution with a diffusion kernel. The repeated application of this convolution to all pixels in Ω results a steady state solution of an isotropic diffusion based on Dirichlet data on $\partial\Omega$.

For our application, we choose a 3×3-neighborhood and update the pixel at the center by isotropically averaging over its eight neighbors. This is equivalent to a convolution with a local-area filter; the corresponding stencil is sketched in figure 3.1(a). Repeated application of the filter, until a steady state is reached, will restore the missing area of the image.

A minor modification of the stencil, accounting for the varying distance of the neighboring pixels from the central pixel yields slightly different filter weights (see figure 3.1(b)). This modification removes artificial grid effects and more accurately represents a radially symmetric propagation/diffusion of information.

3 Inpainting by anisotropic bilateral filtering

The diffusion of image information by the above filters is highly dissipative, in the sense that sharp edges are increasingly smoothed out and eventually lost. This effect is the consequence of the filter's concentration on spatial information (while neglecting pixel information). In other words, only the distance to the central pixel sets the weight of the filter; the pixel value itself does not enter the final filter weight. It thus should not come as a surprise, that edges (i.e., large jumps in pixel values) are ignored and ultimately dissipated. To

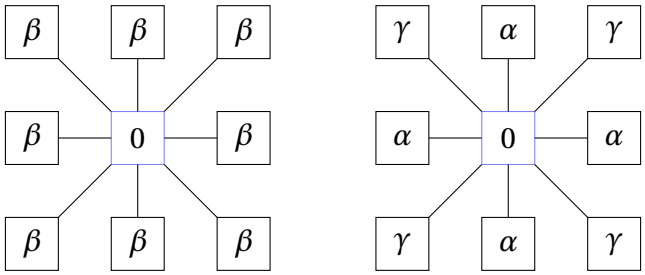

Figure 3.1: Filter weights for isotropic and anisotropic convolution. For the isotropic case (a), we have $\beta = 1/8$; for the anisotropic case (b), we have $\alpha = 0.176765$ and $\gamma = 0.073235$.

counterbalance this tendency, a bilateral filter is introduced: it consists of two parts, (i) a smoothing spatial filter (similar to the one above) and (ii) an edge-preserving filter.

In its simplest representation we take two Gaussian filters for each of the components and write

$$w_1(\mathbf{x},\mathbf{y}) = \exp\left(-\frac{1}{2}\frac{\|\mathbf{x}-\mathbf{y}\|^2}{\sigma_s^2}\right), \tag{3.1a}$$

$$w_2(\mathbf{x},\mathbf{y}) = \exp\left(-\frac{1}{2}\frac{|I(\mathbf{x})-I(\mathbf{y})|^2}{\sigma_r^2}\right). \tag{3.1b}$$

The first weight function w_1 depends on the spatial distance of the pixels and is responsible for pure diffusion. The second weight function w_2 takes into consideration the pixel values I between the two considered points $\mathbf{x}$ (the central position) and $\mathbf{y}$ (the neighborhood position). For a large jump in I between $\mathbf{x}$ and $\mathbf{y}$, the Gaussian in w_2 will be exceedingly small. The two standard deviations σ_s and σ_r will determine the filter width of each component.

The full convolutional filtering then reads

$$I(\mathbf{x}) \leftarrow \frac{1}{w(\mathbf{x},\mathbf{y})}\sum_{\mathbf{y}\in\mathcal{N}} w_1(\mathbf{x},\mathbf{y})\,w_2(\mathbf{x},\mathbf{y})\,I(\mathbf{y}), \tag{3.2a}$$

$$w(\mathbf{x},\mathbf{y}) = \sum_{\mathbf{y}\in\mathcal{N}} w_1(\mathbf{x},\mathbf{y})\,w_2(\mathbf{x},\mathbf{y}). \tag{3.2b}$$

In the above expression, $\mathcal{N}$ is the neighborhood considered in the filter windows. For thin cracks, the window size typically contains the crack and connects known information from both sides of the crack.

4 Implementation

The code below implements the inpainting algorithm above. We import an image that has been vandalized by graffiti. We wish to restore the vandalized image to its original glory. We

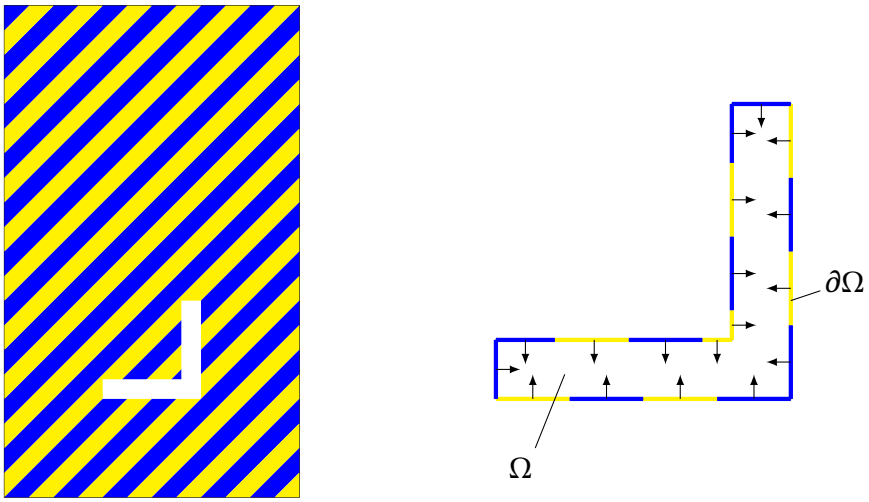

Figure 3.2: Concept of inpainting. A damaged or missing area is identified using a (user-defined) mask (a), after which the information on the edge is propagated into the interior of the domain (b). The propagation ranges from simple diffusion to complex information-propagation along preferred direction. Additional constraints for the propagation, to preserve image features and texture, can be imposed as well.

define a mask that identifies the part of the vandalized image that we want to remove. Once this part is flagged, the pixels in this region are set to 255 (white) and subsequently filled in from the known edges. When applying the convolutional, bilateral filter it is important to consider efficiency issues. While it may be tempting to loop over all pixels in the damaged region, it is far more efficient (you are encouraged to explore the difference) to first identify the indices of the missing pixels and then process them as vectors. This vectorized version of processing the damaged region is far superior in speed than a for-loop. This kind of consideration should be made not only in this example, but in general.

```
1  from PIL import Image
2  from scipy.misc import imsave
3  import numpy as np
4
5  def inpaint(imarr,mask,niter):
6      # eliminate the part of the image covered by the mask
7      irow,jcol = np.where(mask==1)
8      imarr[irow,jcol] = 255
9
10     # embed in a bigger frame
11     n,m = imarr.shape
12     imARR = np.zeros((n+2,m+2))
13     imARR[1:-1,1:-1] = imarr
14
```

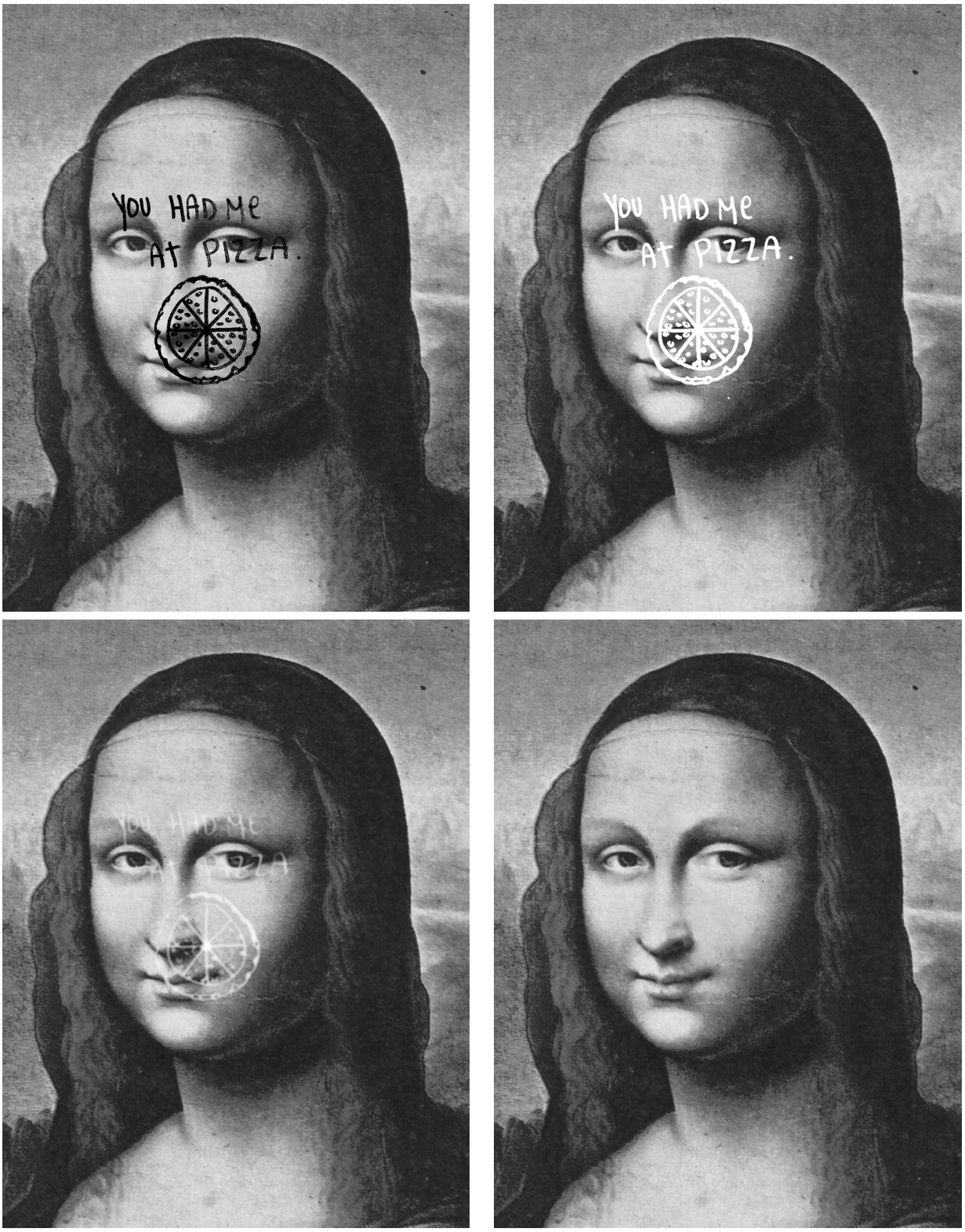

Figure 3.3: Restoration of a vandalized painting by image inpainting. (upper left) vandalized image, with black graffiti; (upper right) the part to be eliminated has been flagged (in white) and defined as a mask Ω; (lower left) work in progress, after 30 iterations of the inpainting algorithm; (lower right) restored painting, after 300 iterations of the inpainting algorithm.

```
    # Neumann extensions
    imARR[0,1:-1]  = imarr[0,:]
    imARR[-1,1:-1] = imarr[-1,:]
    imARR[1:-1,0]  = imarr[:,0]
    imARR[1:-1,-1] = imarr[:,-1]

    # adjust indices from mask (due to embedding)
    ir = irow+1
    jc = jcol+1

    # set the parameters
    iwid   = 1
    sigma2 = 1.
    alpha  = 250.

    # start the inpainting loop
    for i in range(niter):
        if (i%100 == 0):
            print 'iteration = ',i,' out of ',niter
        dARR = np.zeros_like(imARR)
        ww   = np.zeros(len(ir))
        for id in range(-iwid,iwid+1):
            for jd in range(-iwid,iwid+1):
                if ((id==0) and (jd==0)):
                    ee = np.zeros(len(ir))
                    EE = np.zeros(len(ir))
                else:
                    # bilateral filter (Gaussian)
                    dd = np.sqrt(id*id + jd*jd)
                    ee = np.exp(-dd*dd/sigma2)
                    # bilateral filter (gradients)
                    xk = (imARR[ir+id,jc+jd]-imARR[ir,jc])/dd
                    EE = np.exp(-xk*xk/alpha/alpha)
                # combine the two filters
                dARR[ir,jc] += ee*EE*imARR[ir+id,jc+jd]
                ww          += ee*EE
        imARR[ir,jc] = dARR[ir,jc]/ww

    # take off boundary rows
    imarr = imARR[1:-1,1:-1]

    # convert back to image and return
    im_inp = Image.fromarray(imarr.astype(np.uint8))
    return im_inp
```

```
59
60 def convert2BW(img_in):
61     # open the image file
62     im_file = Image.open(img_in)
63
64     # convert image to monochrome
65     im_BW = im_file.convert('L')
66
67     # convert to num-array
68     im_array = np.array(im_BW)
69
70     # convert back to image (for later use)
71     im_file2 = Image.fromarray(im_array)
72
73     # show image (for later use)
74     # im_file2.show()
75     return im_array
```

The main code is given below. It imports the image and the graffiti, overlays the two images, then defines a mask and calls the inpainting algorithm to restore the vandalized image.

```
1  if __name__ == '__main__':
2
3      # convert an image to an "black-and-white" array
4      aa = convert2BW('MonaLisa2.jpg')
5      a  = aa[0:1300,500:1500]
6
7      # convert overlay to an "black-and-white" array
8      bb    = convert2BW('Pizza.jpg')
9      nb,mb = bb.shape
10     for i in range(0,nb):
11         for j in range(0,mb):
12             if (bb[i,j]<100):
13                 bb[i,j] = 0
14     b       = bb.copy()
15     c       = 255*np.ones_like(a)
16     (nb,mb) = b.shape
17     dx      = 210
18     dy      = 180
19     c[dx:dx+nb,dy:dy+mb] = b
20     f     = np.minimum(a,c)
21     ma    = (c<=220)
22     mask  = ma.astype(np.int)
23
```

```
24      niter = 300
25      # convert back to image (for later use)
26      im_f  = Image.fromarray(f.astype(np.uint8))
27      im_f.show()
28      im_f.save('vandalized.png')
29
30      # inpainting
31      img_pp = inpaint(f,mask,niter)
32
33      # display image
34      img_pp.show()
35      img_pp.save('inpainted.png')
```

After 300 iterations of the convolutional, bilateral filter, the image is sufficiently restored (see figure 3.3). While this image has not seriously challenged our bilateral filter in terms of preservation of gradients and edges, it has shown efficiency and efficacy in recovering a moderately damaged painting by inpainting.