Introduction, basic but important concepts

Felix Kubler

1DBF, University of Zurich and Swiss Finance Institute

October 7, 2017
Numerical analysis and scientific computing light-years ahead of us
Try to focus on a set of economic problems and search for best methods available for this problem
Which language?

- Matlab
- Python
- Julia
- Fortran/C++
Writing Code

- Tradeoff between HP-computing and HP-coding
- But remember that one can reuse parts of good code
- Also use other people’s codes (unless they are economists)
- Do not drink and code!
This week

- Standard stuff: Non-linear equations, optimization.
- Somewhat standard stuff: Numerical dynamic programming, projection methods
- Less standard stuff: Dynamic programming with forward looking constraints, sparse grid methods, applied projection methods
8 Chapters

1. Introduction
2. Non-linear equations
3. Non-linear programming
4. Dynamic programming
5. Integration and function approximation
6. Projection Methods
7. High-dimensional projection methods
8. Applications
Abstractly, we will try throughout the class to solve the following problem:
Find an \( x \) such that
\[
F(x, d) = 0,
\]
where \( d \) is a set of data which the solution depends on and \( F \) is some functional relation between \( x \) and \( d \). In this formulation \( x \) can be some element of Euclidean space or a function, while \( F \) might be a complicated functional equation or simply a non-linear system of equations.

An algorithm can now generally be thought of as a sequence of problems \( F_n(., d) \) and a sequence of \( x_n \) such that \( F_n(x_n, d) \approx 0 \).
Errors

In general there are 2 sources of errors in numerical computations: rounding and truncation errors.

- Rounding errors are a consequence of using finite precision arithmetic.
- Truncation errors: Even if one has *convergent* algorithm, for which
  \[\lim_{n \to \infty} F_n(x_n, d) \to F(x, d),\]
  one only has finite time and has to stop at some \( N < \infty \). Any algorithm we use will therefore stop at some approximation \( \hat{x} \) to a \( x^* \) with \( F(x^*, d) = 0 \). In general, we would be happy with a tiny relative error, i.e. with
  \[\frac{|x^* - \hat{x}|}{x^*} \approx u.\]
  This is called a small *forward* error. However, in some circumstances all we can hope for is a small *backward error*, i.e. that \( F(\hat{x}, \hat{d}) = 0 \) for some \( \hat{d} \) with \( \hat{d} - d \equiv u \).
Consider the function

\[ f(x) = \frac{1 - \cos(x)}{x^2} \]

with \( x = 1.2 \times 10^{-5} \) the value of \( \cos \), rounded to 10 significant digits is \( c = 0.9999999999 \) so that

\[ 1 - \cos(x) = 0.000000001 \]

Then \( (1 - c)/x^2 = 10^{-10} / 1.44 \times 10^{-10} = 0.69 \ldots \) Unfortunately, the correct result is 0.5

Suppose we are trying to solve the quadratic equation

\[ 10^{20} x^2 - 3 \cdot 10^{20} x + 2 \cdot 10^{20} = 0 \]

Using the simple formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

the result will almost certainly be \( NaN \).
Floating point arithmetics

A floating point number system \( F \subset \mathbb{R} \) is a subset of the reals which is characterized by 4 integer parameters

- the base \( \beta \) (typically 2)
- the precision \( t \)
- the exponent range \( e_{\text{min}} \leq e \leq e_{\text{max}} \)

The elements of \( F \) have the form

\[
y = \pm \beta^e \times 0.d_1d_2\ldots d_t.
\]

In general the numbers in \( F \) are not equally spaced. E.g. for \( \beta = 2, t = 3, e_{\text{min}} = -1 \) and \( e_{\text{max}} = 3 \) the non-negative numbers are

0, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.750, 0.875, 1.0, 1.25, 1.5, 1.75, 2.0, 2.5, 3, 3.5, 4.0, 5.0, 6.0, 7.0.
Floating point arithmetics (cont.)

Most computers and programming languages (e.g. all Intel based computers) use the so called \textit{IEEE} floating point standard. In this standard there are three precisions with

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{type} & \beta & t & \varepsilon_{\text{min}} & \varepsilon_{\text{max}} & u \\
\hline
\text{IEEE single} & 2 & 24 & -125 & 128 & 6 \times 10^{-8} \\
\text{IEEE double} & 2 & 53 & -1021 & 1024 & 1 \times 10^{-16} \\
\text{IEEE extended} & 2 & 64 & -16381 & 16384 & 5 \times 10^{-20} \\
\hline
\end{array}
\]

Where the maximal \textbf{relative} roundoff error (machine precision) is

\[
u = \frac{1}{2} \beta^{1-t}.
\]
Precision in matlab

- Typically, matlab uses double precision (i.e. 16 digits)
- The command "digits" can be used to increase precision (and decrease speed)
- Fun to play around with
Numerical linear algebra is an extremely well developed field. Excellent code in Fortran, C, and Matlab were developed in the 1970’s and 1980’s. One prominent problem is to solve a linear system: Solve $Ax = b$ for a vector $x \in \mathbb{R}^n$, assuming that the $n \times n$ matrix $A$ is invertible.
Direct methods to solve linear system

NEVER use $x = A^{-1}b$. Instead:

- $LL'$ (Cholesky) factorization (sparse, symmetric, positive definite)
- $LDL'$ factorization (sparse, symmetric, indefinite)
- $LU$ factorization (sparse, generic, full rank)
- $QR$ factorization (dense, generic)
- $USV'$ singular value decomposition (dense, generic)
Iterative methods

One guesses a starting $x^0$ then iterates on

$$x^{k+1} = Bx^k + f$$

until $\|x^{k+1} - x^k\|$ becomes small. Hopefully $B$ and $f$ can be chosen to ensure that $\|x^k - x^*\| \to 0$ for $x^*$ with $Ax^* = b$.

Two important methods: Gauss-Seidel and Gauss-Jacobi
Gauss-Jacobi

For each $i = 1, \ldots, n$, set

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k \right)$$

Note that, in this scheme $B = I - D^{-1} A$ where $D$ is a diagonal matrix with the diagonal elements of $A$ on its diagonal, $f = D^{-1} b$. Note that, fortunately, $x = Bx + f$ and $Ax = b$ are equivalent. Therefore, if $x \approx Bx + f$ that then $Ax \approx b$, so if the iterations converge they converge to the right solution.
Gauss-Seidel

Sometime it is useful to use information from iteration $k + 1$ already in that iteration. One example is Gauss-Seidel

$$x_i^{k+1} = \frac{1}{a_{ii}} \left( b_i \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{k} \right)$$

Lower storage requirements. Perhaps faster.
Iterative methods (concluded)

- Methods are easy to code and can solve huge systems.
- Convergence only guaranteed under unrealistic assumptions.
- Can be very slow, but we can use acceleration methods. For example Chebychev second-order iterative schemes. Here we set

$$\alpha_{i+1} = \omega G(\alpha_i) + (\tau - \omega) \alpha_i + (1 - \tau) \alpha_{i-1}.$$
A very important concept is that of a condition number. Badly conditioned system will be difficult or impossible to solve. For linear systems we can make this concrete:

Define a matrix norm as follows

$$
\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}.
$$

The condition-number of a matrix $A$ is defined by

$$
\kappa(A) = \|A\| \cdot \|A^{-1}\|.
$$
Conditioning and loss in precision (cont)

For small perturbations in linear systems, we have that if

\[ A(x + \delta x) = b + \delta b \]

then

\[ \frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta b\|}{\|b\|} \]
Conditioning and loss in precision (concl.)

\[ \frac{\|\delta_x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta_b\|}{\|b\|} \]

Why?

\[ \frac{\|\delta_x\|}{\|x\|} = \frac{\|\delta_x\|}{\|b\|} \frac{\|b\|}{\|x\|} = \frac{\|A^{-1}\delta_b\|}{\|b\|} \frac{\|Ax\|}{\|x\|} \leq \|A^{-1}\| \cdot \|A\| \frac{\|\delta_b\|}{\|b\|} \]

If \( b \) has \( t \) digits of accuracy then \( x \) has \( t - \log(\kappa(A)) \) digits of accuracy. In this sense \( \log(\kappa) \) measures the loss of precision!

Check out Matlab command
Derivatives

In many numerical methods, we need to evaluate not only a non-linear function but also its first (and perhaps even second) derivatives. In practice we often use a numerical approximation to the derivative, the so-called finite difference derivative:

\[
\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h},
\]

where \( e_i \) is the i’th column of the identity matrix and \( h \) is an appropriate step-length.

This is also sometimes refereed to as ’one-sided’ finite differences and there might be situations where it is better to use two-sided finite-differences defined as

\[
\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}.
\]
Automatic Differentiation

- Any high-school student can take derivatives, so we should be able to teach it to a computer.
- There are programs for Matlab, Fortan, C that do automatic forward differentiation (just google "automatic differentiation matlab")
- It is more efficient to have Mathematica or other algebraic packages to the differentiation and have it simplify the expression
Computational Noise

- We will often be faced with the problem that we want to minimize (or find the root of) a function that is the result of an elaborate calculation. This calculation could involve approximation of other functions and/or integration.
- Clearly if we use Monte-Carlo’s method for integration this function will be noisy in the sense that we actually want to minimize some other true function but can only compute only a noisy approximation.
- Even if the computations are deterministic, this will turn out to be a problem because of truncation and rounding errors.
- In order for many numerical methods to work, this noise should not be too large. It is also easy to see that finite-difference derivatives might be meaningless when there is a lot of noise and, as it turns out, the step-length, $h$, should depend on the noise in the function.
Moré and Wild (2011) introduce a very simple workable definition and an algorithm to estimate noise.

They assume that the true function $f_s(x)$ can only be computed with some random additive element, i.e.

$$f(x) = f_s(x) + \epsilon(x), \quad x \in \mathcal{N}(x_0),$$

where $\epsilon(x)$ is a random variable whose distribution is independent of $x$.

They define the noise as the standard deviation of $\epsilon$.

Their algorithm *ECnoise* (that is available in matlab on Stefan Wild’s website at Argonne National Lab) takes as given the function sampled at $k$ equi-spaced points and determines the noise.
Computational Noise (cont.)

The basic idea for the univariate case is as follows.

- Define $\Delta^0 f(t) = f(t)$ and recursively

  $$\Delta^{k+1} f(t) = \Delta^k f(t + h) - \Delta^k f(t)$$

- If the true function $f_s(.)$ is sufficiently smooth, we obtain that for large $k$

  $$\Delta^k f(t) = f^{(s)}(\xi) h^k + \Delta^k \epsilon(t),$$

  for some intermediate value $\xi$,

- for sufficiently small $h$, $h^k \simeq 0$ which implies that

  $$\Delta^k f(t) \simeq \Delta^k \epsilon(t).$$

As it turns out, we have that

$$\text{var}(\epsilon) = \frac{(k!)^2}{2k!} E((\Delta^k \epsilon(t))^2)$$
Linear programming

We consider the following linear minimization problem under linear constraints.

\[
\min_{x \in \mathbb{R}^n} \quad c^T x \\
\text{s.t.} \quad Ax + b \leq 0 \\
\quad x \geq 0,
\]

where \( c \in \mathbb{R}^n \) non-zero, \( A \) is an \( m \times n \) matrix and \( b \) is a \( m \)-vector.

The dual problem to the above is

\[
\max_{\mu \in \mathbb{R}^m} \quad b^T \mu \\
\text{s.t.} \quad A^T \mu + c \leq 0 \\
\quad \mu \geq 0.
\]
Linear programming (cont.)

- Simplex method
- Interior point method(s)
Convergence rates

Given a method (e.g. to solve non-linear equations or to optimize) that produces a sequence of iterates \((x^k)_{k=0,1,...}\) converging to the solution of the problem \(x^*\), we will sometimes be interested how fast does the method converge. The sequence is said to converge:

- linearly, if there exist 0 < \(q < 1\) and \(k_{max} \geq 0\) such that for all \(k \geq k_{max}\)
  \[
  \|x^{k+1} - x^*\| \leq q\|x^k - x^*\|
  \]

- superlinearly, if there exists a sequence \(q^n \to 0, 0 < q^n\) for all \(n\), and \(k_{max} \geq 0\) such that for all \(k \geq k_{max}\)
  \[
  \|x^{k+1} - x^*\| \leq q_k\|x^k - x^*\|
  \]

- quadratically, if there exists a \(c > 0\) and \(k_{max} \geq 0\) such that for all \(k \geq k_{max}\)
  \[
  \|x^{k+1} - x^*\| \leq c\|x^k - x^*\|^2
  \]
Parallel computations

- The trends in hardware design are such that in the near future even ordinary desktops will host dozens of general purpose processors (possibly combined with co-processors and/or GPUs) making the use of parallelization techniques important even for standard users.

- HPC systems can probably be considered as the most powerful and flexible research instruments available today and allow us to ask questions that would otherwise be impossible to address.

- Their computational power nowadays often reaches multiple petaflops, with an eleven-year cycle of achieving a three-orders-of-magnitude increase in performance.

- Such systems are difficult to deal with. To use them efficiently, one has to design the software carefully, taking into account the individual advantages that the various (off-the-shelf) hardware components of such machines offer. One needs to move away from Matlab.