Numerical Integration and Differentiation

Quantitative Macroeconomics

Raül Santaeulàlia-Llopis

MOVE-UAB and Barcelona GSE

Fall 2017

1. Numerical Differentiation
   - One-Sided and Two-Sided Differentiation
   - Computational Issues on Very Small Numbers

2. Numerical Integration
   - Newton-Cotes Methods
   - Gaussian Quadrature
   - Monte Carlo Integration
   - Quasi-Monte Carlo Integration

Numerical Differentiation

The definition of the derivative at x_* is

$$f'(x_*) = \lim_{h \to 0} \frac{f(x_* + h) - f(x_*)}{h}$$

- Hence, a natural way to numerically obtain the derivative is to use:

$$f'(x_*) \approx \frac{f(x_* + h) - f(x_*)}{h} \tag{1}$$

  with a small h. We call (1) the **one-sided derivative**.

- Another way to numerically obtain the derivative is to use:

$$f'(x_*) \approx \frac{f(x_* + h) - f(x_* - h)}{2h} \tag{2}$$

  with a small h. We call (2) the **two-sided derivative**.

We can show that the two-sided numerical derivative has a smaller error than the one-sided numerical derivative. We can see this in 3 steps.

- Step 1, use a Taylor expansion of order 3 around x_* to obtain

$$f(x) = f(x_*) + f'(x_*)(x - x_*) + \frac{1}{2}f''(x_*)(x - x_*)^2 + \frac{1}{6}f'''(x_*)(x - x_*)^3 + \mathcal{O}_3(x) \quad (3)$$

- Step 2, evaluate the expansion (3) at $x = x_* + h$

$$f(x_* + h) = f(x_*) + f'(x_*)h + \frac{1}{2}f''(x_*)(h)^2 + \frac{1}{6}f'''(x_*)(h)^3 + \mathcal{O}_3(x_* + h) \quad (4)$$

and rearrange to obtain the **one-sided derivative** formula:

$$\frac{f(x_* + h) - f(x_*)}{h} = f'(x_*) + \frac{1}{2}f''(x_*)(h) + \frac{1}{6}f'''(x_*)(h)^2 + \frac{\mathcal{O}_3(x_* + h)}{h} \quad (5)$$

- Step 3, evaluate the expansion (3) at $x = x_* - h$

$$f(x_* - h) = f(x_*) + f'(x_*)(-h) + \frac{1}{2}f''(x_*)(-h)^2 + \frac{1}{6}f'''(x_*)(-h)^3 + \mathcal{O}_3(x_* - h) \quad (6)$$

and combine (4) and (6) to obtain the **two-sided derivative** formula:

$$\frac{f(x_* + h) - f(x_* - h)}{2h} = f'(x_*) + \frac{1}{6}f'''(x_*)(h)^2 + \frac{\mathcal{O}_3(x_* + h) - \mathcal{O}_3(x_* - h)}{2h} \quad (7)$$

Comparing (5) and (7) shows the error associated with the two-sided formula is smaller.

Computational Issues on Very Small Numbers

- Exact arithmetic and computer arithmetic do not always give the same answers. This is not an issue of programming skills but a matter of computer precision.

- For example, compute

$$y = (1.0e - 20 + 1.0) - 1.0$$

and

$$y = 1.0e - 20 + (1.0 - 1.0)$$

where $1.0e - 20$ is the computer's shorthand for 10^{-20}.

Exact arithmetic says the two statements above are identical because addition and substraction are associative. A computer, however, would evaluate the statements differently. The first statement would, incorrectly, likely result in $x = 0$ whereas the second one would, correctly, result in $x = 10^{-20}$.

- Usually, if one is using double precision in Fortran, numbers can be not precise if we reach 10^{-16}.

- For this reason, practice suggets for the two-sided formula an alternative

$$f'(x_*) \approx \frac{f(x_* + h) - f(x_* - h)}{(x_* + h) - (x_* - h)} \tag{8}$$

  with a small h. Note that the denominator might not be precisely $2h$. That is why this formula helps to avoid trouble associated with a small h.

- According to Miranda-Fackler, as rule of thumb for the two-sided derivative, h should be set to:

$$h = \max(|x_*|, 1)\sqrt[3]{\varepsilon} \simeq \max(|x_*|, 1) \times 6 \times 10^{-6}$$

  where ε is the machine epsilon.

- If h is too large we will have approximation error in computing derivative. If h is too small we will have rounding error.

Approximation of a gradient

- Straight forward extension of the univariate case
- Bivariate case:

$$\bigtriangledown f(x_{1,*}, x_{2,*})|_{x_{1,*},x_{2,*}} = \left[f_1(x_{1,*}, x_{2,*}))|_{x_{1,*},x_{2,*}}, f_2(x_{1,*}, x_{2,*}))|_{x_{1,*},x_{2,*}}\right]$$

$$\simeq \left[\frac{f(x_{1,*}+h, x_{2,*}) - f(x_{1,*}-h, x_{2,*})}{(x_{1,*}+h)-(x_{1,*}-h)}, \frac{f(x_{1,*}, x_{2,*}+h) - f(x_{1,*}, x_{2,*}-h)}{(x_{2,*}+h)-(x_{2,*}-h)}\right]$$

- Summing up: Write your own finite difference routine, experiment with h, and be aware you may need to adapt it to each problem you use. MF's suggestion tends to work well.

Numerical Integration

- Goal: Compute the definite integral of a real-valued function f w.r.t. a weight function w over an interval I of $\Re^n$,

$$\int_I f(x)\, w(x)\, dx$$

- The weight function can be, for example
  - $w(x) \equiv 1 \rightarrow$ the integral is the area under f
  - $w(x) \equiv$ p.d.f. of a r.v. $\widetilde{x}$ with support $I \rightarrow$ the integral is $E[f(\widetilde{x})]$.

- We study methods that approximate a definite integral with a weighted sum of function values,

$$\int_I f(x)\, w(x)\, dx \approx \sum_{i=0}^{n} w_i\, f(x_i)$$

- We will see three classes of numerical integration (*numerical quadrature*) methods that differ on how the *quadrature weights* w_i and the *quadrature nodes* x_i are chosen.
    - **Newton-Cotes** methods approximate the integrand f between nodes using low-order polynomials and sum the integrals.
    - **Gaussian Quadrature** methods choose the nodes and weights that satisfy some moment-matching conditions.
    - **Monte Carlo** (and **quasi-Monte Carlo**) methods use equally weighted random or equidistributed nodes.

Newton-Cotes Methods

- Univariate quadrature methods are designed to approximate the integral of a real-valued function f defined on a bounded interval $[a, b]$ of the real line.
- Two Newton-Cotes methods are widely used,
  - **Trapezoid Rule**
  - **Simpson's Rule**
- Both rules are easy to implement and are typically adequate for computing the area under a continuous function.

Trapezoid Rule

- First, partition the interval $[a, b]$ into subintervals, (say, though not necessarily, equal length)
    - Define the nodes $x_i = a + (i-1)h$ for $i = 1, ..., n$ with $h = \frac{b-a}{n-1}$.
- Second, approximate f over each subinterval $[x_i, x_{i+1}]$ using piecewise linear spline passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$

- Third, the area under each line segment defines a trapezoid approximates the area under f over the subinterval,

$$\int_{x_i}^{x_{i+1}} f(x)\, dx \approx \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right]$$

Summing up the areas of the trapezoides across the subintervals yields the Trapezoide rule:

$$\int_a^b f(x)\, dx \approx \sum_i^n w_i\, f(x_i)$$

with $w_1 = w_n = \frac{h}{2}$ and $w_i = h$ otherwise.

- Remarks:
  - It is simple and robust.
  - First-order exact: if not for rounding error, it will exactly compute the integral of any first-order polynomial (a line)
  - If the integrand is smooth, the trapezoid rule yields an approximation error that shrinks quadratically with the width of the subintervals, $\mathcal{O}(h^2)$.

Simpson's Rule

- First, partition the interval $[a, b]$ into an even number of subintervals, (say, equal length)
    - Define the nodes $x_i = a + (i-1)h$ for $i = 1, ..., n$ with $h = \frac{b-a}{n-1}$ and n is odd.
- Second, approximate f over the jth pair of subintervals $[x_{j-1}, x_j]$ and $[x_j, x_{j+1}]$ using a piecewise quadratic function that passes through $(x_{j-1}, f(x_{j-1}))$, $(x_j, f(x_j))$ and $(x_{j+1}, f(x_{j+1}))$

- Third, the area under this quadratic function approximates the area under f over the subinterval,

$$\int_{x_{j-1}}^{x_{j+1}} f(x)\, dx \approx \frac{h}{3}\left[f(x_{j-1}) + 4f(x_j) + f(x_{j+1}))\right]$$

Summing up the areas of the quadratic approximants across subintervals yields the Simpson's Rule:

$$\int_a^b f(x)\, dx \approx \sum_i^n w_i\, f(x_i)$$

with $w_1 = w_n = \frac{h}{3}$ and otherwise, $w_i = 4\frac{h}{3}$ if i is even, and $w_i = 2\frac{h}{3}$ if i is odd.

- Remarks:

  - Easy to implement, as the Trapezoide rule.
  - Even thought it is based on locally quadratic approximations of the integrand, it is third-order exact: if not for rounding error, it will exactly compute the integral of any cubic polynomial.
  - If the integrand is smooth, the Simpson's rule yields an approximation error that shrinks at twice the geometric rate of the error associated with the trapezoid rule, $\mathcal{O}(h^4)$.
  - Simpson's rule is preferred to the Trapezoid rule when f is smooth because it offers twice the degree of approximation.
  - If f exhibits discontinuities, the trapezoid rule will often be more accurate.
  - Newton-Cotes rules based on 4th and higher order piecewise polynomial approximations exist, but rarely used.

- **Higher dimensional integration:** generalizations of the univariate Newton-Cotes quadrature schemes through tensor product principles.

- Suppose one wishes to integrate a real-valued function defined on a rectangle $\{(x_1, x_2) | a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2\}$ in $\Re^2$.

- One way to proceed is to compute the Newton-Cotes nodes and weights:
  - $\{(x_{1i}, w_{1i}) | i = 1, ..., n_1\}$ for the real interval (a_1, b_1), and
  - $\{(x_{2j}, w_{2j}) | j = 1, ..., n_2\}$ for the real interval (a_2, b_2)
- The tensor product Newton-Cotes rule for the rectangle would comprise of the $n = n_1 n_2$ grid points of the form
  - $\{(x_{1i}, w_{2j}) |\ i = 1, ..., n_1,\ j = 1, ..., n_2\}$ with associated weights,
  - $\{w_{ij} = w_{1i} w_{2j} |\ i = 1, ..., n_1,\ j = 1, ..., n_2\}$
- This construction principle can be applied to higher dimensions using repeated tensor product operations.

Gaussian Quadrature

- Gaussian quadratrue rules are construced w.r.t. specific weight fucntions w.
- For a weight function w defined on an interval $I \subset \Re$ of the real line and for a given order of approximation n, the quadrature nodes $x_1, ..., x_n$ and quadrature weights $w_1, ..., w_n$ are chose so as to satisfy $2n$ "moment-matching" conditions.
- $$\int_I x^k \, w(x) \, dx = \sum_{i=1}^{n} w_i \, x_i^k, \quad \text{for} \quad k = 0, ..., 2n-1 \tag{9}$$
- The integral approximation is computed by the weighted sum of function values at the prescribed nodes:

$$\int_I w(x) \, f(x) \, dx \approx \sum_{i=1}^{n} w_i \, f(x_i) \tag{10}$$

- Gaussian quadrature over a bounded interval w.r.t. the identity weight function, $w(x) \equiv 1$, is called Gauss-Legendre quadrature. This is appropriate for computing the area under a curve because of its consistency with Riemann-integrable functions. If f is Riemann integrable, then the approximation afforded by Gauss-Legendre quadrature can be made arbitrarily precise by increasing the number of nodes.

  Gauss-Legendre quadrature should be applied with caution if the function has discontinuous derivatives, as in $f(x) = \frac{x+|x|}{2}$. If the function f possesses known known king points, it is often possible to break theintegral into the sum of 2 or more integlras of smooth functions. When it is not possible to produce smooth integrands this way, then Newton-Cotes quadrature methods are more efficient.

- When the weight function w is the probability density function of some continuous random variable $\widetilde{X}$, Gaussian quadrature basically "discretizes" the continuous random variable $\widetilde{X}$ by replacing it with a discrete random variable with mass points x_i and probabilities w_i that approximate $\widetilde{X}$ in the sense tha tboth random variables have the same moents of order less than $2n$:

$$\sum_{i=1}^{n} w_i x_i^k = E\widetilde{X}^k \quad \text{for} \quad k = 0, ..., 2n-1 \tag{11}$$

Given the mass points and proabilities of the discrete approximant, the expectation of any function of the continuous random variable may be approximated using the expectation of the function of the discrete approximant:

$$Ef(\widetilde{X}) = \int_I f(x)w(x)dx = \sum_{i=1}^{n} w_i f(x_i) \tag{12}$$

Monte Carlo Integration

- Motivated by the Strong Law of Large Numbers: if $x_1, x_2, ...$ are independent realizations of a random variable $\widetilde{X}$ and f is a continuous function, then

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) = Ef(\widetilde{X}) \tag{13}$$

  with probability one.

- The Monte Carlo integration scheme: compute an approximation of the expectation $f(\widetilde{X})$, one draws a random sample $x_1, x_2, ..., x_n$ from the distribution of $\widetilde{X}$ and sets

$$Ef(\widetilde{X}) \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

- Issues with the pseudorandomness:

  - Most packages produce pseudorandom variables that are uniformly distributed on the interval $[0, 1]$.

    A uniform random number generator is very useful for generating random samples from other distributions: suppose $\widetilde{X}$ is has a cumulative distribution function

    $$F(x) = Pr(\widetilde{X} \leq x) \tag{14}$$

    whose inverse has a well-defined closed form.

    If $\widetilde{U}$ is uniformly distributed on $(0, 1)$, then $F^{-1}(\widetilde{U})$ has the same distribution as $\widetilde{X}$.

    Thus, to generate a random sample $x_1, x_2, ..., x_m$ from the $\widetilde{X}$ distribution, one generates a random sample $u_1, u_2, ..., u_n$ from the uniform distribution and sets $x_i = F^{-1}(u_i)$.

  - Most packages also provide intrinsic routines that generate pesudorandom standard normal variables.

- Problem: it is almost impossible to generate a truly random sample of variates of any distribution. The employed subroutines generate purely deterministic, not random, sequences of numbers that depend on the seed (initiazion point). Good subroutines generate sequences that appear to be random, in that they pass certain statistical tests for randomness. This is why we call them "pseudorandom" numbers. [1]

[1] In general, when simulating a model we deal with it by doing a large enough number of simulations and computing statistics that average all the simulations—this applies for both estimated and calibrated economies.

Some other Pros and Cons

- Preferred over Gaussian quadratures because of its simplicity if the routine for computing Gaussian mass points and probabilities is not efficient (or if the integration is over many dimensions).

- Monte Carlo integration is subject to sampling error that cannot be bounded with certainty.

- The approximation can be made more accurate by increasing the size of the random sample, but doing so can be expensive if evaluating f or generating the pseudorandom variate is costly.

- Approximations generated by Monte Carlo integration will vary from one integration to the next unless intiated at the same point. This implies that making use of Monte Carlo integration in conjuction within other iterative schemes (dynamic programming) is then very problematic.

- Quasi-Monte Carlo methods can circumvent some of this problems.

Quasi-Monte Carlo Integration

- Quasi-Monte Carlo methods started from insights from probability theory.
- These methods rely on sequences $\{x_i\}$ with the property that

$$\lim_{n\to\infty} \frac{b-a}{n} \sum_{i=1}^{\infty} f(x_i) = \int_a^b f(x)dx \tag{15}$$

without regard to whether the sequence passes standard tests of randomness.

- Any sequence that satisfies theis condition for arbitrary (Riemann) integrable functions can be used to approximate an integral on $[a, b]$.

- Although the Law of Large Numbers assures us that his statement is true when the x_i are i.i.d., othe sequences also satisfy this property. Actually, sequences that are explicitly nonrandom but instead attempt to fill in space in a regular manner, can often provide more accurate approximations to definite integrals: there are numerous schemes for generating equidistributed sequences including the Neiderreiter, Weyl, and Haber sequences.