Nummerical Programming II

Personal summary by Gilberto Lem

Summer 2018

Contents

7 ODE's: Initial Value Problems

7.1 A User Manual (Part 1)

An Ordinary Differential Equation is presented in the following form:

$$y'(t) = f(t, y(t))$$

Where f is fiven, t is the independent (scalar) variable, and y is the dependent variable. Solution lines for ODE's never cross, they can only touch, in which case uniqueness is lost.

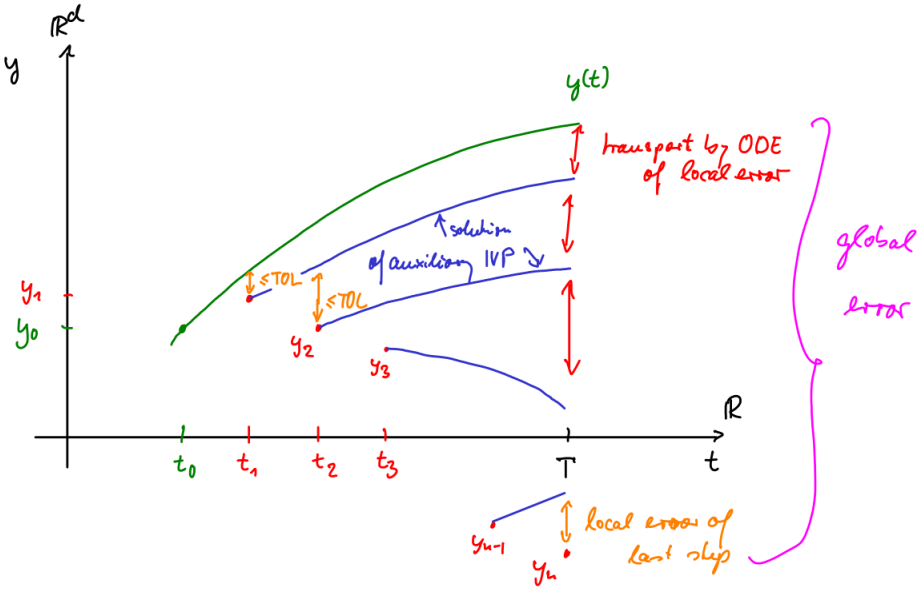
Initial Value Problems (IVP): $y(t_0) = y_0$.

Existence & Uniqueness for IVP

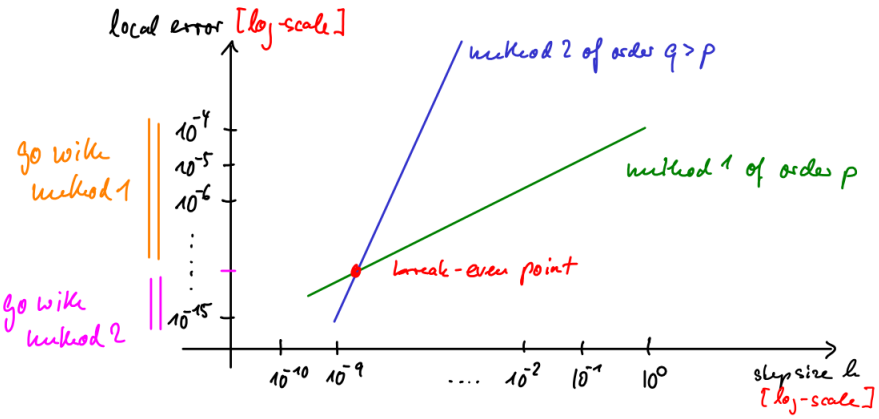
- If f is continuous, then it exists a solution
- If both f and $\frac{\partial f}{\partial y}$ are continuous, then it exists a unique solution

For numeric implementations it is important to consider the uniqueness criteria, for the machine to be able to obtain a valid solution, as well as the sensitivity to inputs of the ODE (take into account numerical error of initial values).

Error Estimation Local error is the difference between the solution of the ODE starting in the last grid-point and the new point y_k given by the numerical method. Global error is the accumulation of those local errors, resulting in the difference between the last y_k given by the numerical method and the analytical solution of the ODE.



The order p of the numerical method tells you how much the error reduces when you modify the stepsize. The local error is $\mathcal{O}(h_k^{p+1})$, while the global error is $\mathcal{O}(h_{max}^p)$. It is important to note that a bigger order does not necessarily imply a lower error. It depends on the stepsize:



In the IVP packages supplied for example by MATLAB, the machine controls the local error to match the user supplied TOL.

As in real life one does not have the analytical solution for the ODE, one can modify the stepsize to accomplish a desired TOL using a better numerical method as the "real" solution. The standard today is consider this as the error (which would correspond to the error of the "worse" numerical method) but use as a solution the one provided by the "better" numerical method. This results in a conservative error estimate, and is called **local extrapolation mode**.

Choosing Tolerances In practice, for the error estimation, one uses the sum of two tolerances: the Relative Tolerance, which takes into account the value of the solution itself, and the Absolute Tolerance, for not being too strict when the value of the solution (and thus the RelTOL) is very small. To choose the tolerance a rule of thumb:

- **RelTOL** adjusted to the measurement errors in the physical system
- **AbsTOL** adjusted to the problem and physical units.

Here is a summary of numerical methods for ODE's in MATLAB.

Name	**Order** (stepsize control)	**Single/Multi step**	**Stiff/Nonstiff**
ode45	5(4)	s	nonstiff
ode23	3(2)	s	nonstiff
ode113	1-13	m	nonstiff
ode23s	3(2)	s	stiff
ode15s	1-5	m	stiff

7.2 Runge Kutta and Multistep Methods

The idea behind Runge Kutta and multistep methods is doing a local estimation of the next value for the solution with the following exact expression:

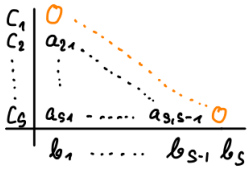
$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\mathrm{d}\tau.$$

Runge Kutta methods use a quadrature formula for approximating the integral. In a code-friendly form for evaluating the expression, one uses already-taken-care-of tables of coefficients c_i, a_{ij}, and b_j to compute:

$$y_{n+1} = y_n + h_n \sum_{j=1}^{n} b_j k_j$$

$$k_i = f(t_n + c_i h_n, y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_j$$

The tables are commonly written in the Butcher-scheme:



As we estimate the local error with two numerical methods of numerical order, we have to compute two different quadratures for each step. A nice thing of this approach is that between the two different methods only the b_j coefficients change, and thus we get to recycle the f-evaluations.

Another trick used is the FSAL (first-same-as-last), where one uses the last f-evaluation of the last step as the first evaluation of the current step, to save one evaluation.

One thing to consider is that the f-evaluations are made in the interval from t_n to t_{n+1}. We could instead of evaluating a complicated function, evaluate an interpolated polynomial and save computational effort. We can construct the polynomials to match our desired local errors. Here the coefficients b_j become $b_j(\theta)$, and the method reduces to:

$$y_{n+1} = y_n + h_n \sum_{i=1}^{s} b_j(\theta) k_j$$

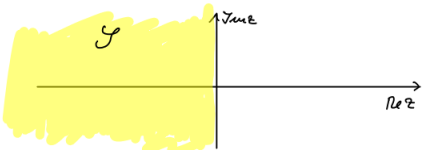
On the other side, **multistep methods** do not use a quadrature to approximate the integral. They instead replace f with a polynomial obtained interpolating its values in the last m steps. The interpolation is $\mathcal{O}(h^m)$ for f, and the method is order m. This needs the computing of only one f-evaluation per step, and thus is useful for very expensive f. You can make a numerical method varying in order very easily, however rapid step size changes degrade the quality of the interpolated polynomial, and thus step sizes have to be changed in a very conservative way.

7.3 Stiffness of IVP

Stiffness imply that one needs certain restrictions on the stepsize for the numerical method to be stable. Specifically, we need the real part of the eigenvalues of the linearized operator $D_y f$ around the point of interest to be negative. This defines some regions of stability for each numerical method. In general, the restrictions have the following form:

$$h|\lambda| \leq c_\alpha$$

A method is called A-stable if the stability region is all space such that $\mathrm{Re} z < 0$:

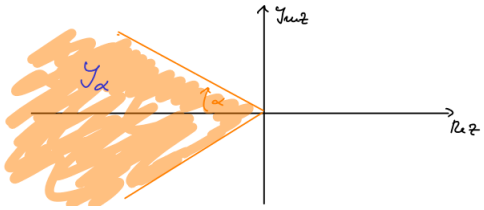


For example, for the `ode45` method in MATLAB, $c_\alpha = 3.31$.

Stiff solvers use implicit methods to overcome the stability problem. This methods imply solving nonlinear systems of equations, which we should solve using Newton iterations (iterate by fixed-point iteration would be a disguised explicit method). The cost for each step of this kind of methods is:

- $\#f$-evaluations = s
- Build the $D_y f$ (Jacobian) matrix. (Automatic differentiation)
- # steps in Newton iteration

Multistep methods add an additional restriction for the stability region. Depending on the number of steps taken, there is an angle α that restricts the region. Here the method is called A(α)-stable.



8 ODE's: Boundary Value Problems

Initial Value Problems have one or more conditions in one point of the independent variable domain. Boundary Value Problems have conditions in more than one point of the independent variable domain.

Unlike in IVP's, there is no general theory for BVP's regarding existence and uniqueness. There can be infinitely many solutions to a problem or none.

Many times, the problem is formulated with an additional parameter, which is part of the ODE and is going to change with the solution.

8.1 Solution Schemes

Shooting Method One solution technique to solving BVP's is the **shooting method**. It solves an IVP for an initial parameter in the ODE, setting with that a nonlinear system of equations and using Newton's method tries to satisfy all the boundary conditions. This method can lead to numerical instability.

Multiple Shooting Another solution scheme is **multiple shooting**. This adds multiple shooting points and solves a more stable but bigger nonlinear system of equations.

Global Solvers for BVP's This is the most general scheme. They take an implicit method (for example the trapezoidal rule) and with that method applied to every point on the grid, plus the boundary conditions, define a $(N+1)d$-dimensional nonlinear system of equations, where N is the number of grid points and d the dimension of the problem. The nonlinear system is solved by Newton iteration. This methods are available in MATLAB with the names bvp4c and bvp5c, where the number 4 or 5 refer to the order of the numerical scheme used (for example the trapezoidal rule would be order 2).

8.2 Practical Issues

8.2.1 Initial Guesses

Newton's method is iterative, and thus we need a useful first guess for the solutions and for the parameters needed. These guesses are made with some preliminary mathematical analysis (as taking approximations), or prior experience from the problem.

Take for example the following ODE:

$$\epsilon y' = \sin^2(x) - \lambda \frac{\sin^4(x)}{y}$$

where λ is the parameter, and ϵ is a small number. As ϵ is small, we could approximate $\epsilon y' = 0$ and with that get an initial guess for $\lambda = 1$ and $y(x) = \sin^2(x)$. We could generalize this to $\epsilon y' = \mathcal{O}(\epsilon)$ to see the order of magnitude of the approximation. This is called **asymptotic analysis.**

Another technique is **Homotopy (parameter continuation).** Imagine we have a problem that has a fixed parameter (for example a fixed Reynolds number in a fluid problem), and that value makes the ODE difficult to solve numerically. One way to deal with this is to solve the BVP for a different and easier value of the parameter, and take that solution as an initial guess for solving the problem with a parameter a little closer to the real value, and iterate.

8.2.2 Parameters not part of the ODE

Imagine the case where the parameter is not part of the ODE, but it is the length of the interval of the independent variable e.g. parameter T in $t \in [0, T]$. In this case we could make a change of variable, using an **auxiliary independent variable** $t = Ts$, $s \in [0, 1]$, and modify the BVP accordingly.

8.3 BVPs as optimization problems

Suppose we want to optimize a function R. In the optimal point:

$$\nabla R = 0$$

Now, if we want to have an additional restriction on the solution, we could express the restriction as $G = 0$, where G is a function. To get a solution for the minimization problem that satisfies the restriction condition, we need the following expression to be true:

$$\nabla R = \lambda \nabla G$$

where λ is a scalar called **Lagrange multiplier**. This means that the gradient of R has the same direction as the gradient of G. For compact notation, we can define an additional function:

$$I = R - \lambda G,$$

and state the problem as simply

$$\nabla I = 0$$

Now imagine that the problem can be stated as an integral equation. Then our optimization problem could be expressed as:

$$I = \int_0^1 F\left(x, y, y'\right) dx = \min!$$

The solution to that problem is provided by **Calculus of Variations** techniques, and is given by the y that solves the **Euler-Lagrange equation**:

$$\frac{\partial F}{\partial y} = \frac{\mathrm{d}}{\mathrm{d}x} \frac{\partial F}{\partial y'}$$

This transforms the problem into a differential problem. If we additionally had some restrictions for y in different parts of the domain of x, as $y(a) = y_a$ and $y(b) = y_b$, then this optimization problem is equivalent to solving a BVP.

If our problem needs to satisfy some restrictions, then our function F should be of the form $R(x, y, y') - \lambda G(x, y, y')$.

One common trick for solving this kind of problems, when the restriction is given in an integral form, is to define an additional differential equation with boundary problems, which represents the restriction.

8.4 BVPs as optimal control problems

Suppose we have a physical phenomenon described by an ODE:

$$x' = f\left(x(t), u(t)\right),$$

with its respective boundary conditions $x(0) = x_0$, $x(1) = x_1$. Here the differential part of the model is for $x(t)$, however, the problem also considers an additional function $u(t)$, which is called **control**. This could be for example a car moving, where the control is how much the accelerator is being pushed.

Suppose we have a quantity of interest from the problem which we want to optimize, for example, minimize the pollution of the car. And that that quantity can be expressed as a function of $x(t)$ and $u(t)$. Then we can formulate the optimization problem:

$$\mathcal{J} = \int_0^1 \varphi\left(x(t), u(t)\right) dt = \min!$$

We want to find the function $u^*(t)$ which optimizes $\mathcal{J}$.

This problem could be equivallently expressed as an optimization problem for $\mathcal{J}$,with a restriction given by $x' = f$. So we can define a Lagrangian:

$$\mathcal{L} = \varphi(x, u) + \lambda\left(f(x, u) - x'\right)$$

Applying Euler-Lagrange equations with respect to x, λ and u (considering all of them as functions) we get:

- w.r.t. x: We call it the **adjoint equation**
- w.r.t. λ : The original ODE of the problem
- w.r.t. u: An ODE on u, which solves for u^*

For easying notation, we can define the **Hamilton function** as:

$$H(x, \lambda, u) = \varphi(x, u) + \lambda f(x, u)$$

Note that this is very similar to the Lagrangian, just removing the x'.

Then the three equations are:

- Adjoint equation: $\lambda' = -\frac{\partial H}{\partial x}$
- Original ODE: $x' = f$
- Equation for u^*: $\frac{\partial H}{\partial u} = 0 \Rightarrow u^* = g(x, \lambda)$

In practice we need to solve this problem with the computer. We have a system of ODE's (adjoint and original) and we have the computation of the optimal u for each of the numerical method's solutions for x and λ. As this is a boundary value problem, we need an initial solution. This is generally done proposing a clever guess for $u(t)$,and with that guess solve for an initial solution for x with the original ODE and an initial solution for λ with the adjoint equation.

9 Partial Diferential Equations

PDE's depend on more than one independent variable on a domain Ω with boundary $\delta\Omega$. This is often stated as one independent vector $x \in \mathbb{R}^n$, and time is often separated: $x = (x', t)$. There is no general theory for solving PDE's, even in the linear case i.e. when there is a spacial dimension and a time dimension.

9.1 General Aspects

9.1.1 Types of PDE's

There are several types of PDE's, and each one has a different kind of set conditions which lead to proper problems.

- **Elliptic PDE's**, such as Laplace Equation or Poisson Equation, need stationary **boundary conditions**, such as Dirichlet or Neumann conditions on $\delta\Omega$.
- **Hiperbolic PDE's** need a time variable, for example the wave equation $u_{tt} = c^2\Delta u$, and need only an **initial condition** $u(x, t = 0) = u_0(x)$, $x \in \Omega$.
- Parabolic PDE's model infinite speed of information transport, for example the heat equation $u_t = k\Delta u$. They need **initial and boundary conditions**, i.e. they need $u(x, t = 0) = u_0(x)$ on Ω, and $u(x, t) = \varphi_t(x)$ on $\delta\Omega$ for $t \geq 0$.

9.1.2 Well-Posedness

The PDE is well posed if and only if there **exists** a **unique** solution for every input, and the solution **depends continuously** on the input i.e. noise in data does not blow up in the solution.

9.1.3 Principles of deducing PDE's

One common way to deduce PDE's is the **variational principle**, which uses the Euler-Lagrange equations for reformulating an integral optimization problem into a differential problem. It may be useful to use Taylor expansions or other simplifications to be able to formulate a simpler problem.

One important tool used in PDE theory is multidimensional integration by parts:

$$\int_\Omega \frac{\partial u}{\partial x_j} v \mathrm{d}x = \int_{\delta\Omega} uv \cdot n_j d\sigma - \int_\Omega u \frac{\partial v}{\partial x_j} \mathrm{d}x$$

When u is a scalar field, this yelds the **divergence theorem.**

9.2 Poisson Equation

9.2.1 Fast Poisson Solvers

One of the most common PDE's is the Poisson equation. It has the following form:

$$-\Delta u = f, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$u|_{\delta\Omega} = 0$$

We will consider the case of the hypercube $\Omega = [-1, 1]^d$

1D problem In the 1D problem, one can discretize and get the operator Δ as a matrix T_n with stencil $[-1, 2, -1]$, a tridiagonal symmetric positive definite matrix.

$$T_n u_n = h^2 f_n,$$

where h is the discretization step. This needs $\mathcal{O}(n)$ flops for solving.

$\mathcal{O}(n)$ is pretty cheap, however in the problem with more dimension, this is nice complexity not achieved. As an alternative we can diagonalize T_n in the following form:

$$S_n T_n S_n = D_n,$$

where $D_n$is a diagonal matrix with the eigenvalues of T_n, and S_n is a unitary matrix with the eigenvectors of T_n, called the **discrete sine transform (DST)**. This transformation is useful because the DST can be easily obtained from to the Discrete Fourier Transform, and we can obtain the DFT with the Fast Fourier Transform algorithm in $\mathcal{O}(n \log n)$ operations (note that $n \log n$ is bigger than n, so for 1D this is not very useful).

2D problem Now we have

$$-\Delta u = (-\frac{\partial^2}{\partial x_1^2} - \frac{\partial^2}{\partial x_2^2})u = f$$

Discretizing, u becomes a matrix, where x_1 varies in the first dimension and x_2 varies in the second dimension. We can apply the T_n matrix to the row indices by multiplying u by the left, and apply it to the columns by multiplying it by the right. The linear system is now:

$$T_n u_n + u_n T_n = h^2 f_n,$$

which has $N = n^2$ unknowns and equations to be solved, and as the system is not tridiagonal anymore, it needs $\mathcal{O}(N^3)$ flops to be solved. But we can transform the problem with the DST. Multiplying the equation from both sides with S_n :

$$D_n \tilde{u}_n + \tilde{u}_n D_n = h^2 \tilde{f}_n$$

As D_n is diagonal, we can solve easily for the element j, k:

$$(\tilde{u}_n)_{jk} = h^2 \frac{(\tilde{f_n})_{jk}}{\lambda_j + \lambda_k}$$

As an algorithm:

1. Compute $\tilde{f}_n = S_n f_n S_n$ $[\mathcal{O}(N \log N)]$
2. Compute $\tilde{u}_n = h^2 \tilde{f} / D_n$ (elementwise division) $[\mathcal{O}(N)]$
3. Backtransform $u_n = S_n \tilde{u}_n S_n$ $[\mathcal{O}(N \log N)]$

So in total this has $\mathcal{O}(N \log N)$, which is useful even for orders of 10^{6-9}

Hypercube The same algorithm is used for $d > 2$, where we have $N = n^d$. Here the DST is successily applied to each coordinate, and instead of matrices we have d-dimensional tensors.

In general, the error is given by the error of the central difference approximation made: $\mathcal{O}(h^2)$. As the number of points in each dimension is $N^{1/d}$, we have $h = N^{-1/d}$, and thus our error is $\mathcal{O}(N^{-2/d})$. Note that this is bad, because when $d \to \infty$, $N^{-2/d} \to 1$, and thus our algorithm has error of $\mathcal{O}(1)$. This is called **curse of dimensionality**. If we want to acheieve an absolute error of $TOL = N^{-2/d} \Rightarrow N = TOL^{-d/2}$, then we need a complexity of $\mathcal{O}(N \log N) = \mathcal{O}(d \cdot TOL^{-d/2} \log TOL)$.

9.2.2 Monte-Carlo Methods for nD Poisson

We take as a model the homogeneous Poisson equation (Laplace equation):

$$-\Delta u = 0, \qquad \text{on } \Omega \subset \mathbb{R}^d$$

$$u|_{\delta\Omega} = w$$

Functions that satisfy this equation are called harmonic, and satisfy the **mean-value property**. This property says that the value of $u(x_0)$ is equal to the mean value of $u(x)$ on a sphere with center on x_0. This added with brownian motion magic lead to the following algorithm, called Random Walk on Spheres (WoS):

1. Get the radius of the largest sphere that fits into Ω around x_j
2. Sample uniformily from that sphere to get x_{j+1}
3. If $x_{j+1} \in \delta\Omega$ (with a tolerance h), stop and return $w(x_{j+1})$ as an estimate of $u(x)$

Error Monte-Carlo methods have a statistical error of $\mathcal{O}(1/\sqrt{N})$, where N is the number of samples taken, and the error of introducing a tolerance in the boundary is $\mathcal{O}(h)$, thus the total error is $\mathcal{O}(1/\sqrt{N}) + \mathcal{O}(h)$. To make both error contributions equally large, we define $h \coloneqq 1/\sqrt{N}$, and finally our error is $\mathcal{O}(1/\sqrt{N})$.

Complexity The expected number of steps to teach the boundary scales as $\mathcal{O}(d\,|\log h|) = \mathcal{O}(d\,\left|\log N^{-1/2}\right|) = \mathcal{O}(d \log N)$. Additionally, the number of flops per step is $\mathcal{O}(d)$. So, our total complexity is $N \cdot \mathcal{O}(d \log N) \cdot \mathcal{O}(d) = \mathcal{O}(d^2 N \log N)$

If we want to acheive an absolute error $TOL = 1/\sqrt{N}$, then we have a complexity of $\mathcal{O}(d^2 TOL^{-2} \log TOL)$. Note that for WoS we do not have d in the exponent, whereas in the best grid-based method we have a complexity of $\mathcal{O}(TOL^{-d/2})$. This sets a break even point. For example for $TOL = 10^{2-3}$ the breakeven is in $d \approx 5 - 6$, and thus for that tolerance, if $d \geq 7$, we would go with WoS.

Summary WoS is nice because it is mildly dependent on d, embarassingly parallel, simple to program and uses only pointwise evaluations. However it has a limited accuracy.