Math 128A, Summer 2019
Lecture 21, Monday 7/29/2019

1 Review, Homework 5

1.1 Problem 2

The idea behind a difference equation is a lot simpler than a differential equation but is very parallel. Recall the problem is to find the general solution of

$$u_{j+2} = u_{j+1} + u_j$$

The space of equations is two-dimensional. There are two ways to approach this, and we do it the better way of the two.

First consider the separate problem:

$$\begin{aligned} y''' &= f(t, y, y', y'', z) \\ z' &= g(t, z) \end{aligned}$$

To solve this sort of problem, usually this is difficult to solve; however, for numerical purposes, we want to systematically convert this to a first-order system (which we know works). First, we want to convert the variables (with different levels of differentiation) into new variables. We can make this an autonomous system:

$$u' := \begin{bmatrix} t \\ y \\ y' \\ y'' \\ z \end{bmatrix}' = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}' = \begin{bmatrix} 1 \\ u_2 \\ u_3 \\ f(u_0, u_1, u_2, u_3, u_4) \\ g(u_0, u_4) \end{bmatrix} =: F(u)$$

Remember the idea behind (applied) mathematics is not to solve problems but to move problems around to somewhere we already have a big hammer to drop onto it (we already know how to solve first-order equations).

Remark: Technically our construction is dimensionally off, so we'll want to insert standardizing factors to match the dimensions of y:

$$u' = \begin{bmatrix} y_0 t/h \\ y \\ hy' \\ \frac{h^2}{2!} y'' \\ z \end{bmatrix}$$

So back to our original problem, we construct:

$$U_j = \begin{bmatrix} u_j \\ u_{j+1} \end{bmatrix} \quad U_{j+1} = \begin{bmatrix} u_{j+1} \\ u_{j+2} \end{bmatrix} = \begin{bmatrix} u_{j+1} \\ u_{j+1} + u_j \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} u_j \\ u_{j+1} \end{bmatrix} =: FU_j = F^{j+1} U_0$$

1.2 Question 5

Suppose $y(t)$ is the exact solution to the following initial value problem:

$$\begin{aligned} y' &= f(t, y(t)) \\ y(0) &= y_0 \\ e'(t) &= f(t, u(t) + e(t)) - u'(t) \quad \text{(we use this)} \\ f(y) &= \lambda y \end{aligned}$$

Because we have $f(t, y(t)) = y' = \lambda y$, we conclude $y(t) = e^{t\lambda} y(0)$ (this can be easily verified).

$$\begin{aligned} e' &= \lambda(u + e) - u' \\ &= \lambda e + \lambda u - u' \\ (u + e)' &= \lambda(u + e) \\ u + e &= e^{t\lambda} \left[u(0) + e(0)\right] = y(t), \end{aligned}$$

and we are done.

2 Homework: euler.m

There is no iteration within Euler's method; our step is simply from t to $t + 1$. The function is 4 in, 4 out.
Consider the Matlab function:

```
function v = f(t,u,p)
  v(1) = 0.7/(u(1)**2 + u(2)**2)^1.5
  v(2) = -0.3/(u(1)**2 + u(2)**2)^1.5
  v(3) = u(3)
  v(4) = u(4)
end
```

We put 4 in and get 4 out. We call this via `euler(a,b,y0,@f,p)`.

3 Multistep Methods

We haven't said everything about Runge-Kutta methods (that would take a few semesters), but we'll proceed past this.

The idea behind multistep methods is to use previous information and essentially be **unlike Runge-Kutta methods** (where we take samples and slope estimates in between t_n and t_{n+1}). In Runge-Kutta, we don't look backwards (i.e t_{n-17}). However, if we assume that historical data is indicative of the future, we may want to take Multistep methods into account.

To be more specific, suppose we have a bunch of points: $t_{n+1}, t_n, \dots, t_{n-k+1}$. We write it this way because if $k = 1$, then this yields a "one-step" method, and $k > 1$ yields a "k-step methods" . Hence Runge-Kutta methods are 1-step methods. One obvious idea is to take the values of y at $t_{n-k+1}, \dots, t_n$ (and possibly t_{n+1} which we don't know) and interpolate per our usual integral equation:

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + \int_{t_n}^{t_{n+1}} y'(s)\ ds \\ P(t_{n-j}) &= y'(t_{n-j}) \end{aligned}$$

We have two possibilities:
(1) $0 \le j \le k-1$ yields an **explicit** method as it does not use any unknown values. We call this a k-step Adams method.
(2) $-1 \le j \le k-1$ yields an **implicit** k-step Adams method.

Example: Implicit 2-step Adams Method:
Consider the following:

$$\begin{aligned} p(t_{n+1}) &= y'(t_{n+1}) \cong f(t_{n+1}, u_{n+1}) \\ p(t_n) &= y'(t_n) \cong f(t_n, u_n) \end{aligned}$$

We can write:

$$\begin{aligned} \int_{t_n}^{t_{n+1}} P(s)\ ds &= \int_{t_n}^{t_{n+1}} \frac{s - t_{n+1}}{t_n - t_{n+1}} f_n + \frac{s - t_n}{t_{n+1} - t_n} f_{n+1}\ ds \\ &= \frac{1}{2}\left(t_{n+1} - t_n\right) f_n + \frac{1}{2}\left(t_{n+1} - t_n\right) f_{n+1} \\ &= b_n \left[\frac{1}{2} f_n + \frac{1}{2} f_{n+1}\right] \end{aligned}$$

This 2-step Adams method yields exactly equivalent to the Trapezoidal rule. For multistep methods, we always want to keep open the possibility that the length of the step h changes at each step. The reason is that to get these multi-step methods, we start with a single method. If every step is the same length, then we start off with way too much error from our first step. A better idea would be to take a very tiny step for our 1-step first step, then gradually larger steps as we go forward.

We have

$$\begin{aligned}
p(s) &= \sum_{j=0}^{k-1} L_j(s) f_{n-j} \quad \text{(explicit)} \\
\implies u_{n+1} &= u_n + \int_{t_n}^{t_{n+1}} p(s)\ ds \\
&= u_n + \int_{t_n}^{t_{n+1}} \sum_{j=0}^{k-1} L_j(s) f_{n-j}(s)\ ds \\
&= u_n + \sum_{j=0}^{k-1} p_j(s) f_{n-j}(s); \qquad p_j := \int_{t_n}^{t_{n+1}} L_j^p(s)\ ds
\end{aligned}$$

And of course, p_j changes with each step, so we have to re-do this last line calculation at each step (accurately and efficiently). One such way is to write out the equations and perform explicit calculations (Gauss used Taylor expansions). Later, people didn't like starting with Taylor expansions, so they use Runge-Kutta methods. The idea is to use Runge-Kutta to start off the first 10 or so steps, then proceed with multi-step methods.
On the other hand, for Implicit methods:

$$\begin{aligned}
q(s) &= \sum_{j=-1}^{k-1} L_j(s) f_{n-j} \\
q_j &= \int_{t_n}^{t_{n+1}} L_j(s)\ ds
\end{aligned}$$

This gives us:

$$\begin{aligned}
L_j^p(s) &= \prod_{l=0, l\neq j}^{k-1} \frac{s - t_{n-l}}{t_{n-j} - t_{n-l}}, \qquad \deg k \\
L_j^q(s) &= \prod_{l=-1, l\neq j}^{k-1} \frac{s - t_{n-l}}{t_{n-j} - t_{n-l}}. \qquad \deg k+1
\end{aligned}$$

We don't really want to integrate these guys, so we recall we have Gaussian integration with m points evaluates $\int p$ exactly if $\deg(p) \leq 2m-1$. Recall in our homework, with a degree-5 polynomial, we can get the weights exactly and integrate exactly for degree up to 9.

If the degree of the Gaussian integration is high, numerically when we are evaluating the Legendre polynomials, we accumulate a lot of error. What Strain does for his own projects is store the points and weights for distinct Gaussian quadratures in different files and pull them in when needed.

If we look at books covering Multistep methods, we take a lot of time to explain how to exactly calculate the coefficients q_j, L_j. Strain reminds us (as it is not mentioned anywhere) to simply use Gaussian Quadrature when it delivers an exact result.

3.1 Questions on Multistep Methods: Order of Accuracy (Local Truncation Error), Stability, Getting Started

Before we talk about how to get started, we want to first talk about the accuracy of the scheme. We know

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} y'(s)\ ds$$

where y is the exact solution. To compare, we replace y' with the interpolation polynomial at a bunch of points:

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + \int_{t_n}^{t_{n+1}} \left[y'(s) - p(s) + \underline{p(s)} \right]\ ds \\ &= y(t_n) + \sum_{j=0}^{k-1} p_j f(t_{n-j}, y_{n-j}) + \underbrace{\int_{t_n}^{t_{n+1}} y'(s) - p(s)\ ds}_{h\cdot\tau,\text{ local truncation error}} \end{aligned}$$

It turns out that $y'(s) - p(s)$ is not very difficult to estimate, because p is the interpolating polynomial for y'. Interpolating at k points, for $0 \le j \le k-1$, degree $p = k-1$ (explicit Adams), we have:

$$y'(t_{n-j}) = p(t_{n-j}),$$

so

$$y'(s) - p(s) = \frac{y^{(k+1)}(\xi)}{k!} \left[(s - t_n)(s - t_{n-1}) \cdots (s - t_{n-k+1}) \right]$$

Integrating both sides and dividing by $\frac{1}{h_n}$ to get the actual local truncation error, we have:

$$\int_{t_n}^{t_{n+1}} y'(s) - p(s) = \int_{t_n}^{t_{n+1}} \frac{y^{(k+1)}(\xi)}{k!} \left[(s - t_n)(s - t_{n-1}) \cdots (s - t_{n-k+1}) \right]\ ds$$

We can use the MVT for integrals:

$$\frac{1}{b-a} \int_a^b f(x) g(x)\ dx = f(\xi) \frac{1}{b-a} \int_a^b g(x)\ dx$$

The key hypothesis (besides continuity and differentiability is $g(x) > 0$. For our original integral above, we only run into trouble at the left endpoint, where $(s - t_n)$ gives $(t_n - t_n) = 0$, and all other points are in the past. We claim we're good to proceed.
Hence we write:

$$\int_{t_n}^{t_{n+1}} y'(s) - p(s) = \frac{y^{(k+1)}(\xi)}{k!} \underbrace{\left[\frac{1}{h_n} \int_{t_n}^{t_{n+1}} (s - t_n) \cdots (s - t_{n-k+1})\ ds \right]}_{= P_k},$$

where we name this bracketed expression due to its importance. A Gaussian quadrature scheme would give the constant P_k. Our question is how big is P_k? We claim that if $h_j \le O(h)$, then we should have $P_k = O(h^k)$, and hence

$\tau = O(h^k)$ for a k-step explicit method.

The local-truncation error is:

For k-step explicit:

$$\tau_p = \frac{P_k}{k!} y^{(k+1)}(\xi)_P$$

For $(k-1)$-step implicit:

$$\tau_q = \frac{q_k}{k!} y^{(k+1)}(\xi_q)$$

Hence we write:

$$\begin{aligned} q_k \frac{\tau_p - \tau_q}{p_k - q_k} &= q_k \frac{1}{k!} y^{k+1}(\xi) + \cdots \\ &\cong \tau_q \end{aligned}$$

If we compute:

$$\begin{aligned} &\text{explicit} \qquad v_{n+1} = v_n + \sum_{i=0}^{k-1} p_j f_{n-j} + h\tau^{(p)} \\ &\text{implicit} \qquad u_{n+1} = u_n + q_{-1} f_{n+1} + \sum_{j=0}^{k-1} q_j f_{n-j} + h\tau_n^{(q)} \end{aligned}$$

Subtracting these gives:

$$\begin{aligned} v_{n+1} - u_{n+1} &= h_n \tau_n^{(p)} - h_n \tau_n^{(q)} \\ \frac{v_{n+1} - u_{n+1}}{h_n} &= \tau_n^{(p)} - \tau_n^{(q)}, \end{aligned}$$

and we write, for the error estimate:

$$\frac{q_k}{p_k - q_k} \frac{v_{n+1} - u_{n+1}}{h_n} \cong \tau_n^{(q)}.$$

The interesting thing about this scheme is that if we perform both the explicit and implicit schemes (where the latter is a lot of work via bisection, Newton's or fixed point iteration), we can express the error as the difference of the two as above.

Remark: We call this "Predictor-Corrector": we use explicit method (requires two steps) to give an error estimate, order $O(h^k)$ tells us how big our next step should be. We feed our result from the explicit method into the implicit method.

Lecture ends here.

Next time, we'll talk about stiff equations (going back to Runge-Kutta methods because they work better).

4 Office Hours

4.1 Deferred Correction

Recall our scheme was

$$y' = f(y)$$

$$u_{n+j+1} = u_{n+j} + hf(u_{n+j}), \qquad 0 \le j \le p-1$$

We call our interpolation $u(t)$, and we have:

$$e'(t) = f(u+e) - u' \tag{1}$$

The error equation is simply what we add to our approximate solution to get the exact solution. That is,

$$\begin{aligned}(u+e)' &= f(u+e)\\ e(t_n) &= 0\end{aligned}$$

We use (1) to actually solve the equation. We already know what u is, and therefore we can solve via Euler's method:

$$\begin{aligned}e_{n+j+1} &= e_{n+j} + h\left[f(u_{n+j} + e_{n+j}) - u'(t_{n+j})\right]\\ e_n &:= 0, \qquad 0 \le j \le p-1\end{aligned}$$

Because Euler's method is first-order accurate, these errors are like the exact errors in the relative sense. To see this, write out the Butcher array to verify the order conditions are satisfied.
Now that we have the estimated error, we want to **deliver** the corrected solution, $1 \le j \le p$:

$$u^1_{n+j} = u_{n+j} + e_{n+j}$$

Notice that we put an interpolating polynomial through u, where u' is simply the differentiation coefficients:

$$u'(t_{n+j}) = \sum_{k=0}^{p} d_{j,k} u_{n+k}$$

We run Euler's through all our points of interest $t_n, \dots, t_{n+p}$ first before interpolating.