Lecture 12

- Eigensystems: conclusion

  Ordinary differential equations:

- Initial and boundary value problems
- The Euler method
- The Runge-Kutta method.
- Stepsize control.

The QL decomposition we discussed last class is too costly computationally for the algorithm to be feasible.
Unless the matrix A has some special properties, for instance if it is tridiagonal.
The **Householder method** is a technique that can be used to bring any matrix to a tridiagonal form. The technique is close in spirit to the Sherman-Morrison formula we discussed last class. The basic ingredient is the Householder matrix,

$$\mathbf{P} = \mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T \qquad |\mathbf{w}|^2 = 1.$$

The matrix is orthogonal since it is symmetric and its square is one, meaning $P=P^{-1}$,

$$\begin{aligned} \mathbf{P}^2 &= (\mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T) \cdot (\mathbf{1} - 2\mathbf{w} \cdot \mathbf{w}^T) \\ &= \mathbf{1} - 4\mathbf{w} \cdot \mathbf{w}^T + 4\mathbf{w} \cdot (\mathbf{w}^T \cdot \mathbf{w}) \cdot \mathbf{w}^T \\ &= \mathbf{1} \end{aligned}$$

The interesting property of this matrix is that given a vector it zeroes out all its components except one. To see this, rewrite it as,

$$\mathbf{P} = \mathbf{1} - \frac{\mathbf{u} \cdot \mathbf{u}^T}{H}$$

$$H \equiv \frac{1}{2} |\mathbf{u}|^2$$

Now choose,

$$\mathbf{u} = \mathbf{x} \mp |\mathbf{x}| \mathbf{e}_1$$

$\mathbf{e}_1 = [1,0,\ldots 0]^T$

Then

$$\begin{aligned} \mathbf{P} \cdot \mathbf{x} &= \mathbf{x} - \frac{\mathbf{u}}{H} \cdot (\mathbf{x} \mp |\mathbf{x}| \mathbf{e}_1)^T \cdot \mathbf{x} \\ &= \mathbf{x} - \frac{2\mathbf{u} \cdot (|\mathbf{x}|^2 \mp |\mathbf{x}| x_1)}{2|\mathbf{x}|^2 \mp 2|\mathbf{x}| x_1} \\ &= \mathbf{x} - \mathbf{u} \\ &= \pm |\mathbf{x}| \mathbf{e}_1 \end{aligned}$$

We now consider a transformation formed by an (n-1) rank Householder matrix generated with the vector $[a_{21}, \ldots, a_{n1}]^T$

$$\mathbf{P}_1 \cdot \mathbf{A} = \left[\begin{array}{c|ccc} 1 & 0 & 0 & \cdots & 0 \\ \hline 0 & & & & \\ 0 & & & & \\ \vdots & & {}^{(n-1)}\mathbf{P}_1 & & \\ 0 & & & & \end{array}\right] \cdot \left[\begin{array}{c|ccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline a_{21} & & & & \\ a_{31} & & & & \\ \vdots & & \text{irrelevant} & & \\ a_{n1} & & & & \end{array}\right]$$

$$= \left[\begin{array}{c|ccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{irrelevant} & & \\ 0 & & & & \end{array}\right]$$

The complete transformation yields,

$$\mathbf{A}' = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P} = \left[\begin{array}{c|ccc} a_{11} & k & 0 & \cdots & 0 \\ \hline k & & & & \\ 0 & & & & \\ \vdots & & \text{irrelevant} & & \\ 0 & & & & \end{array}\right]$$

We now continue, applying an n-2 matrix, etc. It is clear that one will end up with a matrix in tridiagonal form.

Instead of actually carrying out the matrix multiplications in $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}$, we compute a vector

$$\mathbf{p} \equiv \frac{\mathbf{A} \cdot \mathbf{u}}{H}$$

Then

$$\mathbf{A} \cdot \mathbf{P} = \mathbf{A} \cdot (1 - \frac{\mathbf{u} \cdot \mathbf{u}^T}{H}) = \mathbf{A} - \mathbf{p} \cdot \mathbf{u}^T$$

$$\mathbf{A}' = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P} = \mathbf{A} - \mathbf{p} \cdot \mathbf{u}^T - \mathbf{u} \cdot \mathbf{p}^T + 2K\mathbf{u} \cdot \mathbf{u}^T$$

where the scalar K is defined by

$$K = \frac{\mathbf{u}^T \cdot \mathbf{p}}{2H}$$

If we write

$$\mathbf{q} \equiv \mathbf{p} - K\mathbf{u}$$

then we have

$$\mathbf{A}' = \mathbf{A} - \mathbf{q} \cdot \mathbf{u}^T - \mathbf{u} \cdot \mathbf{q}^T$$

This is the computationally useful formula.

Improving by iteration:

$$(\mathsf{A} - \tau \mathsf{1}) \cdot \mathsf{y} = \mathsf{b}$$

With τ close to eigenvalue, **b** random. Solve for **y**. It will be close to an eigenvector.

$$\mathbf{y} = \sum_j \alpha_j \mathbf{x}_j \qquad \mathbf{b} = \sum_j \beta_j \mathbf{x}_j$$

$$\sum_j \alpha_j (\lambda_j - \tau) \mathbf{x}_j = \sum_j \beta_j \mathbf{x}_j$$

$$\alpha_j = \frac{\beta_j}{\lambda_j - \tau}$$

$$\mathbf{y} = \sum_j \frac{\beta_j \mathbf{x}_j}{\lambda_j - \tau}$$

So if τ is close to λ_j, then y will be close to x_j. Each iteration gives an extra power of λ_j-τ in the denominator.

Ordinary differential equations:

The generic problem in ordinary differential equations is given by a coupled set of N first order ODE's,

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N$$

since any higher-order system can be brougth to this form by re-defining the variables, e.g.,

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \qquad \begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned}$$

The problem is not entirely specified just giving the equations. To finalize characterizing the problem one needs to provide boundary conditions. The type of boundary conditions determines the numerical method one will use to attack the problem.

Boundary conditions can be simply given by pre-specified values of the functions at certain points or can have complicated forms, for instance non-linear relations among the variables at some points.

Two broad types of boundary conditions can be distinguished:

-In *initial value problems*, one specifies the variables y at some point x_s, and wishes to obtain the values of y at some "future" point x_f.

-In *two point boundary value problems* the variables y are specified at two points, perhaps some y's at the first one and some at the second one, and one is interested in values of y in the intermediate points. We will concentrate on these further on in the course.

The general strategy for solving an initial value problem is to replace the dx's and dy's in the equation by finite increments Δx and Δy, and multiplying the equations times Δx. Then one is left with a set of algebraic equations for the change in the variables y when the independent variable x is "stepped" a step-size Δx.

Straight implementation of this idea, however, fails. It leads to Euler's method, which we will see is impractical. It is conceptually important however, as a building block for constructing the really successful methods. We will discuss three such methods: a) The Runge-Kutta method, b) The Bulirsch-Stoer method and
c) the predictor-corrector method.

The Runge-Kutta method is based on considering several Euler steps and then matching to a Taylor series expansion of a given order.

The Bulirsch-Stoer method is based on the idea we already encountered of Richardson extrapolation, in particular extrapolating Δx to zero

The predictor-corrector methods store the function along the way and use those values to predict the value at the next stepsize. They then compare with the actual stepsize and correct. They work very well where extrapolation works: for smooth functions.

In practice you use Runge-Kutta if you either a) don't know better (!) b) the other methods fail or c) the problem is not computationally demanding and therefore you don't bother doing better. Most of us are typically in a combination of a) and c)

Bulirsch-Stoer methods are comparatively new, but they are overtaking fast the predictor corrector methods, which are more cumbersome.

Euler's method:

Is what first comes to mind: $y_{n+1} = y_n + hf(x_n, y_n)$

However, we already learnt in this course that this is not too smart a way to proceed: the derivative is being evaluated at a midpoint between n and n+1, yet the right hand side is being evaluated at the beginning point n. The method is therefore un-symmetrical and as a consequence not as accurate as it could be with the given number of operations performed.

Worse, the method can be surprisingly unstable. Consider the equation,

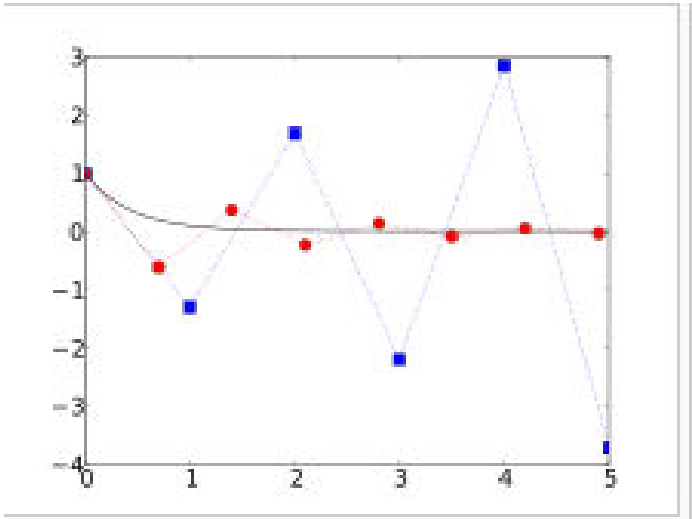
$y' = -100y$ Solution is obviously $y = e^{-100x}$

And in the interval, say x=[0,5], is smooth and well behaved.

Yet the Euler method gives $y_{n+1} = y_n - 100hy_n = (1-100h)y_n$

And if the stepsize is bigger than 0.02 $|y_{n+1}|>|y_n|$ for all n and therefore the solution **diverges**! It "bounces around" the correct one with diverging amplitude.

From Wikipedia:



Solution of $y' = -2.3y$ computed with the Euler method with step size $h = 1$ (blue squares) and $h = 0.7$ (red circles). The black curve shows the exact solution.

Runge-Kutta method:

Suppose we try to fix the unsymmetric nature of the Euler method. To achieve this, we use the Euler method to "go to the midpoint" in terms of y, and then use the resulting value to re-evaluate the right hand side,

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$

$$y_{n+1} = y_n + k_2 + O(h^3)$$

The last equation tells us that we corrected the problem, i.e., we gained one power of h in the error by centering the derivative, as we learnt earlier in the course. The method is called "second order" accurate based on thinking of the Euler method as "first order". (A method with error $O(h^{n+1})$ is therefore called n-th order accurate).

Of course, there is no reason to stop at second order. We can write expressions that approximate the right hand side in the same way as the ones before to low order, but arranged in such a way that higher orders also cancel. A very common implementation is the fourth-order Runge-Kutta,

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
$$

One can straightforwardly derive similar formulae, but they have been tabulated for many years, for instance in Abramowitz & Stegun.

Is fourth order better than second? We know the answer to that one: high order does not necessarily mean high accuracy, it depends on the kind of function being considered. The popularity of fourth order Runge-Kutta suggests that for many practical application it is the case that it is better, but this is not a theorem!

Since we have agreed that there is no universal answer in terms of accuracy, what is one to do? The smart thing to do is to supplement the basic routine with any extra information we might have about the function. For instance, we might want to make the stepsize smaller in regions where the function varies rapidly whereas the basic integration scheme remains the same.

This suggests that it is rational to implement, programming-wise, an ODE solver as a two level set of programs: one that implements the basic routine, perhaps returning some sort of error estimate on the solutiong given, and a higher level program that “drives” it, making decisions about stepsize, etc, based on the errors. One can even think of a third-level “super-driver” that interacts with the user, etc.

Initial value

Initial derivatives

User-supplied subroutine that computes RHSs.

```
      subroutine rk4(y,dydx,n,x,h,yout,derivs)
      parameter (nmax=10)
      dimension y(n),dydx(n),yout(n),yt(nmax),dyt(nmax),dym(nmax)
      hh=h*0.5
      h6=h/6.
      xh=x+hh
      do 11 i=1,n
        yt(i)=y(i)+hh*dydx(i)
11    continue
      call derivs(xh,yt,dyt)
      do 12 i=1,n
        yt(i)=y(i)+hh*dyt(i)
12    continue
      call derivs(xh,yt,dym)
      do 13 i=1,n
        yt(i)=y(i)+h*dym(i)
        dym(i)=dyt(i)+dym(i)
13    continue
      call derivs(x+h,yt,dyt)
      do 14 i=1,n
        yout(i)=y(i)+
     !h6*(dydx(i)+dyt(i)+2.*dym(i))
14    continue
      return
```

```
      subroutine rkdumb(vstart,nvar,x1,x2,nstep,derivs)
      parameter (nmax=10)
      common /path/ xx(200),y(10,200)
      dimension vstart(nvar),v(nmax),dv(nmax)
      do 11 i=1,nvar
        v(i)=vstart(i)
        y(i,1)=v(i)
11    continue
      xx(1)=x1
      x=x1
      h=(x2-x1)/nstep
      do 13 k=1,nstep
        call derivs(x,v,dv)
        call rk4(v,dv,nvar,x,h,v,derivs)
        if(x+h.eq.x)pause 'stepsize not significant'
        x=x+h
        xx(k+1)=x
        do 12 i=1,nvar
          y(i,k+1)=v(i)
12      continue
13    continue
      return
```

"Dumb driver"

Adaptive step control:

A good ODE integrator should exert control on the stepsize. The difference can obviously be abysmal in functions with detailed features in some region but smooth elsewhere.

To control the stepsize one needs to estimate the error. This has computational cost.

With the RK method a strategy is to double the stepsize. That is, advance from x to x+2h first as a single step, then as two steps of length h. Since each RK step requires four evaluations, one needs a total of 11 functional evaluations (excluding reprogramming). This has to be compared with 8 evaluations, that is, a factor 1.375. So we consider,

$$y(x+2h) = y_1 + (2h)^5\phi + O(h^6) + \dots$$

$$y(x+2h) = y_2 + 2(h^5)\phi + O(h^6) + \dots$$

And the truncation error is,

$$\Delta \equiv y_2 - y_1$$

Of course, one could have combined the above formulae to produce a result that is fifth-order accurate. But then we would not have any control on its error.

Once we know Δ, we can use it to tune the stepsize. Since we know that Δ goes as h^5 we can tune our stepsize. Suppose you take a stepsize h_1 that yields an error Δ_1 then the stepsize h_0 that would have given an error Δ_0 is,

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}$$

Error control is, as usual, more subtle than this. First, we have ignored the fact that one usually has many Δ's, one per each variable in the problem. Here one faces a choice: perhaps take the smallest one.

Of course, one can have functions of different amplitude, in such case one probably wants to evaluate errors percentage-wise.But things can be worse: one can have oscillatory functions. In such case one probably wants an error that is a fraction of the maximum amplitude.

A pragmatic method to handle this is to set the error to,

$$\Delta_0 = \texttt{eps} \times \texttt{yscal(i)}$$

Where yscal(i) is a user-supplied vector of scales that will be problem-dependent. For an easy fractional scaling set yscal=y.

A worse situation is encountered in a case in which one is sensitive to accumulation of errors. In such case taking more stepsizes may increase the error! In such case one probably wants to set the scaling to,

$$\Delta_0 = \epsilon h \times \texttt{dydx(i)}$$

In this way we take into account that a smaller h will also require a more stringent tolerance of errors.

But one has to be careful. If Δ_0 depends on h linearly, then the power in our estimator of h should not be 0.2 but 0.25! When writing a driver routine has to prepare for this contingency, perhaps by using the most conservative number: 0.2/0.25 if we increase/decrease the stepsize.

Summary

- Householder method is the one of choice to bring a matrix to tridiagonal form.
- The Euler method fails.
- 4^{th} Order Runge-Kutta is a popular method for treating ODEs.
- Adaptive step control can help enormously.