

PART E. NUMERIC ANALYSIS

Content. Basic concepts, solving equations, interpolation, integration (Chap. 19)
Gauss, Gauss-Seidel methods, LU, matrix inversion, methods for
eigenvalues (Chap. 20)
Euler, Runge-Kutta, Adams-Moulton methods for ODEs and systems,
methods for PDEs (Chap. 21)

Chapter 19

Numeric Methods in General

Content. Significant digits, quadratic equations (Ex. 19.1, Pr. 19.1)
Solution of equations (Bisection, Newton's method, etc.; Exs. 19.2–19.6,
Prs. 19.2–19.6)
Interpolation (Lagrange, Newton, splines; Exs. 19.7, 19.8,
Prs. 19.7–19.11)
Integration (Ex. 19.9, Prs. 19.12–19.15)

`evalf[k](f)` gives floating point representations with k S (k **significant digits**).

Examples for Chapter 19

EXAMPLE 19.1

LOSS OF SIGNIFICANT DIGITS. QUADRATIC EQUATION

Find the roots of the quadratic equation $2x^2 - 50x + 1 = 0$ to 4S (4 significant digits).

Solution. Type a general quadratic equation. Then obtain the usual formula by the command `solve`. Note that the discriminant could be obtained separately, by `discrim`. (Type `?solve`, `?discrim`.)

```
[ > eq := a*x^2 + b*x + c = 0;                                # Resp. eq := ax^2 + bx + c = 0
  > sol := solve(eq, x);
  sol := 1/2 * (-b + sqrt(-4ac + b^2))/a, -1/2 * (b + sqrt(-4ac + b^2))/a
  > SqrDisc := sqrt(discrim(lhs(eq), x));
  SqrDisc := sqrt(-4ac + b^2)
```

Now substitute the given $a = 2$, $b = -50$, $c = 1$ into the solution formula, obtaining (with 4 digits)

```

> Sqrdisc := evalf[4](subs(a = 2, b = -50, c = 1, SqrDisc));
                                49.92
> Sol := evalf[4](subs(a = 2, b = -50, c = 1,
    (-b + Sqrdisc)/(2*a), (-b - Sqrdisc)/(2*a)));
                                {0.02000, 24.98}

```

You got a 4S value for the larger root. But the other root shows **loss of significant digits** because it is a small difference of large numbers, $50.00 - 49.92$. An **improved formula** follows by noting that c/a is the product of the two roots. Note that `sol[1]` is used because $b = -50$ so we get $50 + 49.92$ in the denominator.

```

> improved := (c/a)/sol[1];      # Resp. improved := 2  $\frac{c}{-b + \sqrt{-4ac + b^2}}$ 
> ImpVal := evalf[4](subs(a = 2, c = 1, (c/a)/Sol[2]));
                                Imp := 0.02002

```

To avoid misunderstanding, 4S was used for convenience. For more digits the situation is the same in principle. For instance, you get 30S and 27S for the roots, respectively, from the usual formula, but 30S for the smaller root by the improved formula.

```

> S := evalf[30](solve(2*x^2 - 50*x + 1 = 0, x));
    S := 24.9799839743486850360549356722, 0.0200160256513149639450643278
> evalf(1/(2*max(abs(S[1]))),30);
                                0.0200160256513149639450643277873
> evalf[30](RootOf(2*x^2 - 50*x + 1, x));
                                0.0200160256513149639450643277873

```

Similar Material in AEM: Sec. 19.1

EXAMPLE 19.2

TESTING CONVERGENCE BY THE BISECTION METHOD: PROCEDURE

Compare the merits of the absolute error and the relative error in the bisection method.

Solution. The bisection method is one of the simplest (and slowest) methods for finding roots. The process involves choosing two points **a** and **b** ($b > a$) at which the continuous function, for which you want the root, has different signs, then bisecting the $[a, b]$ interval at a point c . Now, you have two intervals $[a, c]$ and $[c, b]$ and you select the one at whose endpoints the function f has different signs. This is the subinterval in which the zero lies. Repeat until the interval is sufficiently small. Because the initial interval has length $b - a$ and is bisected, after n steps you have an interval of length $(b - a)/2^n$. Provided that you have a continuous function and that $f(a)$ and $f(b)$ have different signs, the method will enclose the root within an interval as small as you require. It is easy to go through such an algorithm a step at a time and see when you think the solution is sufficiently accurate (smaller than your tolerance). The difficulty arises when you write a computer program and ask “when do you stop?” (i.e. the stopping criterion). A common answer is “when the difference between two successive approximations is small” (also called the error or, more correctly, the absolute error). It turns out that that answer is only correct when

the root is in the neighbourhood of 1. If the root is very much less or very much greater than 1 this produces an erroneous result as will be seen.

Consider the following procedure ([?proc](#) for details) that will compute the values obtained by the bisection method along with the absolute error ([err](#)) and the relative error ([relerr](#)). The absolute error is the *number of correct digits after the decimal*; the relative error is the *number of significant digits*. The procedure will also determine if the correct result would be obtained, with a specified accuracy, under each stopping criterion. The [printf](#) ([?printf](#); for details) command produces a nicely formatted output which makes the results easier to read.

```
> ERRTEST := proc(f, a, b, tol, N)
    local left := a;           # Left end of original interval
    local right := b;          # Right end of original interval
    local res := [0, 0]:
    local j, mid, err, relerr:
    printf("%s %s %s %s %s %s %s %s\n",
        "Ind", "Left", "Mid", "Right", "Err", "Err met", "RelErr",
        "RelErr met");
    for j from 1 to N do
        err := (right - left)/2: # Absolute error - width of interval
        mid := left + err;       # Bisection point
        errmet := evalb(abs(err) < tol);
        relerr := evalf[15](abs(err/mid)); # relative error
        relmet := evalb(relerr < tol);
        # If we have not previously reached the stopping point,
        # save 'mid' in res
        if (errmet and res[1] = 0) then res[1] := evalf(mid): end if:
        if (relmet and res[2] = 0) then res[2] := evalf(mid): end if:
        printf("%3d %-11.6g %-13.6g %-11.6g %-11.6g %-7s %-11.6g %-7s\n",
            j, left, mid, right, err, errmet, relerr, relmet);
        # Find which of [left, mid] or [mid, right] contain the root
        if evalf(subs(x = mid, f(x))*subs(x = left, f(x))) > 0
        then
            left := mid;         # f(left) has same sign as f(mid)
            # Make the midpoint the new left
        else
            right := mid;        # f(left) has opposite sign to f(mid)
            # Make the midpoint the new right
        end if
    end;
    res;
end:
Warning, 'errmet' is implicitly declared local to procedure 'ERRTEST'
Warning, 'relmet' is implicitly declared local to procedure 'ERRTEST'
```

The warnings tell us that the variables indicated are **local variables**, that is, they have values that are known only within the `ERRTEST` procedure. This means that we cannot, for example, use the value of `errmet` outside the procedure. To avoid such warnings we could use the command `local errmet, relmet;` after the line `ERRTEST := proc(f, a, b, tol, N)`. This method is used for other variables in the procedure.

There are many advantages to using a procedure. The first one is that you now merely need to insert your data, that is, the function f , the endpoints a and b of the

starting interval, and the required accuracy.

The first function that we will use (see below) has an obvious root at 0.000005

```
[ > Digits := 10:
```

```
[ > f := x -> (x - 0.000005)^3:
```

In order to keep the amount of output short, we will require a tolerance of $10^{(-5)}$. Greater accuracy will result in a longer output and we would need more digits to enable us to compare our result to the correct value. Now call the procedure.

```
[ > ERRTEST(f, 0, 1, 10^(-5), 35);
```

Ind	Left	Mid	Right	Err	Err met	RelErr	RelErr met
1	0	0.5	1	0.5	false	1	false
2	0	0.25	0.5	0.25	false	1	false
3	0	0.125	0.25	0.125	false	1	false
4	0	0.0625	0.125	0.0625	false	1	false
5	0	0.03125	0.0625	0.03125	false	1	false
:	:	:	:	:	:	:	:
16	0	1.52588e-05	3.05176e-05	1.52588e-05	false	1	false
17	0	7.62939e-06	1.52588e-05	7.62939e-06	true	1	false
18	0	3.81470e-06	7.62939e-06	3.81470e-06	true	1	false
19	3.81470e-06	5.72205e-06	7.62939e-06	1.90735e-06	true	0.333333	false
20	3.81470e-06	4.76837e-06	5.72205e-06	9.53674e-07	true	0.2	false
:	:	:	:	:	:	:	:
31	4.99934e-06	4.99981e-06	5.00027e-06	4.65661e-10	true	9.31359e-05	false
32	4.99981e-06	5.00004e-06	5.00027e-06	2.32831e-10	true	4.65658e-05	false
33	4.99981e-06	4.99992e-06	5.00004e-06	1.16415e-10	true	2.32834e-05	false
34	4.99992e-06	4.99998e-06	5.00004e-06	5.82077e-11	true	1.16416e-05	false
35	4.99998e-06	5.00001e-06	5.00004e-06	2.91038e-11	true	5.82076e-06	true

```
[0.000007629394531, 0.000005000008969]
```

Some lines of output have been removed to save space. Notice that the absolute error criterion would terminate after 17 steps (“true” in “Err met” column) and would give a root of 0.00000762939453. This is a **useless** approximation to 0.000005 even though it is accurate to the five digits after the decimal (all 0’s). The relative error criterion would terminate after 35 steps with an **excellent** approximation, correct to 5 significant digits.

The opposite effect occurs for large roots:

```

> f := x -> (x - 500000)^3:
  ERRTEST(f, 100000, 1000000, 10^(-5), 40);

```

Ind	Left	Mid	Right	Err	Err met	RelErr	RelErr met
1	100000	550000	1.00000e+06	450000	false	0.818182	false
2	100000	325000	550000	225000	false	0.692308	false
3	325000	437500	550000	112500	false	0.257143	false
4	437500	493750	550000	56250	false	0.113924	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
16	499985	499998	500012	13.7329	false	2.74659e-05	false
17	499998	500005	500012	6.86646	false	1.37328e-05	false
18	499998	500002	500005	3.43323	false	6.86643e-06	true
19	499998	500000	500002	1.71661	false	3.43323e-06	true
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
35	500000	500000	500000	2.61934e-05	false	5.23869e-11	true
36	500000	500000	500000	1.30967e-05	false	2.61934e-11	true
37	500000	500000	500000	6.54836e-06	true	1.30967e-11	true
38	500000	500000	500000	3.27418e-06	true	6.54836e-12	true
39	500000	500000	500000	1.63709e-06	true	3.27418e-12	true
40	500000	500000	500000	8.18545e-07	true	1.63709e-12	true

[5.000000000 10⁵, 5.000019073 10⁵]

In this case, the relative error criterion has found the root ($5.000019073 \cdot 10^5$ – five significant digits) at 18 steps but the absolute error is still quite large (it keeps going until it has five digits after the decimal – a total of 10 digits).

With this example you can see that the relative error is the best choice for our iterative methods.

EXAMPLE 19.3 SOLVING EQUATIONS BY THE BISECTION METHOD

After the previous example, you can now write a procedure for the bisection method using many of the same commands. As described in that example, you choose two points **a** and **b** ($b > a$) and a **tol**. Type the procedure as follows. Apply it to specific equations afterwards.

```

> BISECT := proc(f, a, b, tol)
  local left := a;           # Left end of original interval
  local right := b;          # Right end of original interval
  local mid, err:
  if (left > right) then error "Incorrect interval - left > right" end;
  if (evalf(subs(x = left, f(x))*subs(x = right, f(x))) < 0)
  then
    err := (right - left)/2:
    mid := left + err; # Bisection point
    while (abs(err) > abs(mid)*tol) do
      # Remove the "#" from the next line to see the steps
      # print('left' = left, 'mid' = mid, 'right' = right);
      if evalf(subs(x = mid, f(x))*subs(x = left, f(x))) > 0
      then
        # f(left) has same sign as f(mid)
        left := mid;           # Make the midpoint the new left
      else
        # f(left) has opposite sign to f(mid)
        right := mid;          # Make the midpoint the new right
      end if:
      err := (right - left)/2:
      mid := left + err;       # Bisection point
    end do;
    else error "Incorrect interval - f(left) has same sign as f(right)"
    end if;
    evalf(mid);
  end:

```

You should notice some things about the procedure. There are a couple of tests for errors: a) it tests to make sure that $a > b$ and b) it tests to make sure that the signs of $f(a)$ and $f(b)$ differ. Provided that f is continuous, the algorithm will converge but you may wish to insert `if (evalf(subs(x = left, f(x))*subs(x = right, f(x))) < 0) then error "Incorrect interval - f(left) has same sign as f(right)"` before the `end do;` in case f is not continuous. Notice also that the relative error test is `abs(err) > abs(mid)*tol` rather than `abs(err/mid) > tol` in case $mid = 0$.

Now see what happens with the functions that you used in Example 1 — along with the relative errors.

```

[ > f := x -> (x - 0.000005)^3:
[ > val := BISECT(f, 0, 1, 10^(-5), 35);           # Resp. 0.000005000008969
[ > (val - 0.000005)/0.000005;                     # Resp. 0.000001794
[ > f := x -> (x - 500000)^3:
[ > val := BISECT(f, 100000, 1000000, 10^(-5));    # Resp. 5.000019073 10^5
[ > (val - 500000)/500000;                          # Resp. 0.000003815

```

You can also try other functions,

```

[ > BISECT(sin, 0.1, 5, 10^(-6), 50);              # Resp. 3.141590643
[ > BISECT(sin, 0.1, 1, 10^(-10));
Error, (in BISECT) Incorrect interval - f(left) has same sign as f(right)

```

The reason for the error message is that the $\sin(0.1) = 0.0998$ and $\sin(1) = 0.8415$ i.e. they are both positive so the bisection method cannot be used for this interval.

```

[ > BISECT(sin, 5, 0.1, 10^(-10));
Error, (in BISECT) Incorrect interval - left > right
Here,  $a \geq b$ .
    As another example, let
[ > x := 'x': g := x^2 - 1:
[ > BISECT(g, 0.1, 5, 0.0002);
Error, (in BISECT) cannot determine if this expression is true or false:
((.1)(.1)^2-1.)*((5)(5)^2-1.) < 0
Here you need another kind of representation of a function, as shown.
[ > g := x -> x^2 - 1; # Resp.  $x \rightarrow x^2 - 1$ 
[ > BISECT(g, 0.1, 5, 0.0002); # Resp. 1.000057984
[ > BISECT(g, -2, 0.5, 0.0002); # Resp. -1.000091553
[ > g := x -> (10^(-10) - x):
[ > BISECT(g, 1, 2, 10^(-10));
Error, (in BISECT) Incorrect interval - f(left) has same sign as f(right)
[ > BISECT(g, 0, 1, 10^(-10)); # Resp. 1.000000000 10-10
[ > g := x -> (10^(-30) - x):
[ > BISECT(g, 0, 1, 10^(-10)); # Resp. 1.000000000 10-30

```

The second advantage of a procedure is that you can save it for future use. The choice of the name of the file is up to you. The command is

```
[ > save BISECT, bisect:
```

This will **save** the **BISECT** procedure in the file **bisect**. (Type **?save** for information.) To **use it again** (after having it filed as just shown), type

```
[ > read bisect:
```

(Type **?read** for information.) This will load it, ready for your new use.

Similar Material in AEM: Sec. 19.2

EXAMPLE 19.4 FIXED-POINT ITERATION

Find a solution of $x^3 + 4x^2 - 30 = 0$ by **fixed-point iteration**.

Solution. Transform the given $f(x) = x^3 + 4x^2 - 30 = 0$ algebraically into the form $x = g(x)$ and solve it by iteration, that is, start from a suitable x_0 and calculate

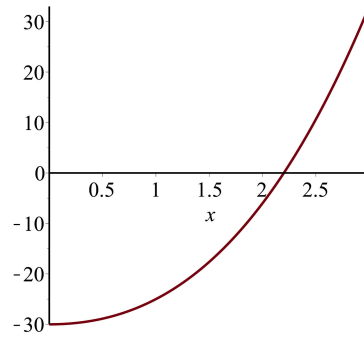
$$(A) \quad x_1 = g(x_0), \quad x_2 = g(x_1), \quad \dots, \quad x_n = g(x_{n-1}), \quad \dots$$

until the desired accuracy is reached. Now $f(x) = 0$ can be cast into the form $x = g(x)$ in several ways. For instance $4x^2 = 30 - x^3$ or

$$(B) \quad x = g(x) = 1/2\sqrt{(30 - x^3)}.$$

To find a suitable x_0 , sketch or plot $f(x) = x^3 + 4x^2 - 30$. Type

```
> plot(x^3 + 4*x^2 - 30, x = 0..3, xtickmarks = [0, 0.5, 1, 1.5, 2, 2.5]);
```



Example 19.4. Given function $f(x)$

Even from a rough sketch you see that there is a zero near 2. Hence you may start the iteration from $x_0 = 2$. You have to do the transformation of $f(x) = 0$ into $x = g(x)$ in such a way that you have convergence. If $|g'(x)| \leq K < 1$ in an interval which contains the solution and the starting value and in which $g'(x)$ is continuous, then you have convergence of the iteration (A). This is true if you choose (B) and, say, $x_0 = 2$. Type

```
> x(0) := 2;
```

```
> g := x -> 1/2*sqrt((30 - x^3));      # Resp.  $g := x \rightarrow 1/2 \sqrt{-x^3 + 30}$ 
```

We need to ensure that $|g'| < 1$ near the possible root.

```
> evalf(subs(x = 2, diff(g(x), x)));evalf(subs(x = 2.346, diff(g(x), x)));
      -0.6396021493
      -0.9985459958
```

Type the iteration as a **do-loop**, which is terminated by **end**.


```

> for n from 1 to 60 do
  x(n) := evalf[8](g(x(n-1)));
end:
seq(x(n), n = 0..60);
2, 2.3452079, 2.0676893, 2.2999960, 2.1114606, 2.2686200, 2.1403409,
2.2469408, 2.1596164, 2.2320210, 2.1725708, 2.2217857, 2.1813144,
2.2147808, 2.1872322, 2.2099952, 2.1912444, 2.2067298, 2.1939681,
2.2045035, 2.1958184, 2.2029867, 2.1970762, 2.2019535, 2.1979314,
2.2012501, 2.1985130, 2.2007713, 2.1989086, 2.2004454, 2.1991778,
2.2002235, 2.1993610, 2.2000725, 2.1994856, 2.1999697, 2.1995704,
2.1998998, 2.1996281, 2.1998522, 2.1996674, 2.1998198, 2.1996940,
2.1997978, 2.1997122, 2.1997828, 2.1997246, 2.1997726, 2.1997330,
2.1997656, 2.1997388, 2.1997609, 2.1997426, 2.1997578, 2.1997452,
2.1997556, 2.1997470, 2.1997542, 2.1997482, 2.1997532, 2.1997490

```

The last value is accurate to 6S, as you can see by typing

```

> evalf(solve(x^3 + 4*x^2 - 30, x));
2.199750914, -3.099875457 + 2.007157166I, -3.099875457 - 2.007157166I

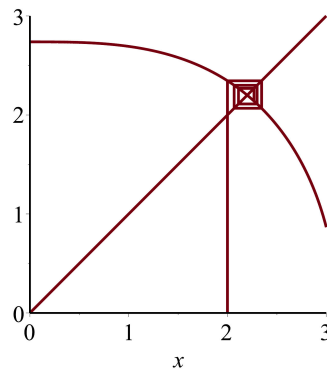
```

To represent the iteration graphically, type

```

> with(plots):
> P1 := plot(x, x = 0..3):
> P2 := plot(g(x), x = 0..3):
> P3 := plot([[x(0), 0], [x(0), x(1)], [x(1), x(1)], [x(1), x(2)],
  [x(2), x(2)], [x(2), x(3)], [x(3), x(3)], [x(3), x(4)], [x(4), x(4)],
  [x(4), x(5)], [x(5), x(5)], [x(5), x(6)]]], style = LINE,
  scaling = constrained):
> display(P1, P2, P3, view = [0..3, 0..3]);

```



Example 19.3. Graphical representation of the iteration

Similar Material in AEM: Sec. 19.2

EXAMPLE 19.5 SOLVING EQUATIONS BY NEWTON'S METHOD

Find the positive solution of $4 \sin 2x = x^2$.

Solution. Solve $f(x) = x^2 - 4 \sin 2x = 0$. To obtain a starting value, sketch $f(x)$. You see that your solution is near 1. As the starting value take $x_0 = 1$. You also need the derivative $f'(x)$.

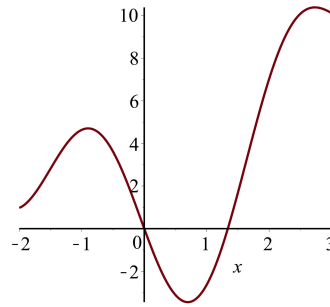
We will create a procedure for the Newton's method. Note that we have checked for small derivatives. Unlike bisection, which will always converge for a continuous function once we have an appropriate interval (i.e. the function has opposite signs at the end points), Newton's method can diverge if the derivative is very small. A better method might be to find an interval, as in bisection, and stop if Newton's method produces an iterate outside the interval.

```
> NEWTON := proc(f, df, p0, tol, max1)
  local k, dfVal;
  P(1) := p0;
  for k from 2 to max1 do
    dfVal := evalf(subs(x = P(k - 1), df));
    if (abs(dfVal) < 10^(-5))
    then
      error "Small derivative";
    end if;
    P(k) := evalf(P(k - 1) - subs(x = P(k - 1), f)/dfVal);
    # Remove the "#" from the next line to see the steps
    # printf ("P( %d ) = %f\n", k, P(k));
    if (abs(P(k) - P(k - 1)) <= tol*abs(P(k)))
    then return P(k);
    end if;
  end;
  error "Did not converge";
end;
```

Consider the function

```
[> f := x^2 - 4*sin(2*x);                                # Resp.  $f := x^2 - 4 \sin(2x)$ 
[> df := diff(f, x);                                     # Resp.  $df := 2x - 8 \cos(2x)$ 
```

```
> plot(f, x = -2..3);
```



Example 19.3. Curve of the function $f(x)$ whose zero is sought

Starting with a value close to the minimum tests the small derivative check.

```
> NEWTON(f, df, 0.6977330717, 10^(-7), 20);
```

```
Error, (in NEWTON) Small derivative
```

```
> NEWTON(f, df, 1, 10^(-7), 20);
```

```
# '#' removed to show steps
```

```
P( 2 ) = 1.494859
```

```
P( 3 ) = 1.345337
```

```
P( 4 ) = 1.338587
```

```
P( 5 ) = 1.338566
```

```
P( 6 ) = 1.338566
```

```
1.338566376
```

```
> fsolve(f = 0, x, 1..2);
```

```
# Resp. 1.338566375
```

```
> save NEWTON, newt:
```

Similar Material in AEM: Sec. 19.2

EXAMPLE 19.6

SOLVING EQUATIONS BY THE SECANT METHOD

The formula for the secant method is

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Using this method, find the positive solution of $4 \sin 2x = x^2$. Compare with the previous example.

Solution. Solve $f(x) = x^2 - 4 \sin 2x = 0$. You need two starting values x_0 and x_1 at which f has different signs. From a sketch of f (see the figure in the previous example) you see that you may take, say, $x_0 = 1$ and $x_1 = 2$. Accordingly, type

```
> f := x^2 - 4*sin(2*x);
```

```
# Resp. f := x^2 - 4 sin(2 x)
```

```
> x(0) := 1;
```

```
# Resp. x(0) := 1
```

```
> x(1) := 2;
```

```
# Resp. x(1) := 2
```

```

> for n from 1 to 9 do
  x(n + 1) := x(n) - evalf(subs(x = x(n), f)*(x(n) - x(n-1))/
    (subs(x = x(n), f) - subs(x = x(n-1), f)));
end;

x(2) := 1.272876722
x(3) := 1.332230072
x(4) := 1.338782501
x(5) := 1.338565731
x(6) := 1.338566375
x(7) := 1.338566376
x(8) := 1.338566376
x(9) := Float(undefined)
x(10) := Float(undefined)

```

This is typical. The denominator of the formula of the method is $f(x_n) - f(x_{n-1})$. It becomes 0 as soon as the two x -values are equal within the number of digits used in the calculation.

Similar Material in AEM: Sec. 19.2

EXAMPLE 19.7

POLYNOMIAL INTERPOLATION

Polynomial interpolation means the following. Given $n + 1$ ordered pairs of numbers $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ (Cartesian coordinates of points in the xy -plane), find the (unique) polynomial $p_n(x)$ of smallest degree that assumes those y -values at the corresponding x -values. In general, p_n will have degree n , but the degree may sometimes be less, depending on those values (on the location of the given points through which the curve of p_n is supposed to pass). p_n is obtained by the command `interp`. (Type `?interp`.)

For instance, given $(8.0, 2.0794), (8.5, 2.1401), (10.0, 2.3026)$, you obtain a quadratic polynomial p_2 by typing the vector **u** of the x -values, the vector **v** of the y -values, and then a letter for the independent variable of p_2 , say, x . Thus,

```

[ > u := [8.0, 8.5, 10.0]; # Resp. u := [8.0, 8.5, 10.0]
[ > v := [2.0794, 2.1401, 2.3026]; # Resp. v := [2.0794, 2.1401, 2.3026]
[ > p2 := interp(u, v, x);
    p2 := -0.006533333333 x^2 + 0.2292000000 x + 0.663933334

```

If you want a value of p_2 at an x somewhere between the given values, say, at $x = 9.2$, type

```

[ > evalf[7](subs(x = 9.2, p2)); # Resp. 2.104072

```

Rounded to 5S, this gives 2.1041. The given values are 5S-values of the natural logarithm where $\ln 8.2 = 2.104134154$. Hence your **quadratic interpolation** has given a correct 5S-value. Greater accuracy would require more given data so that an interpolation polynomial of higher degree could be obtained. No **not** use too high a degree.

For given data, **Lagrange's** and **Newton's interpolation formulas** give the same polynomial (except, perhaps, for small round-off errors), although both formulas look totally different.

Newton's divided difference formula gives p_2 as follows.

```
[ > a11 := (v[2] - v[1])/(u[2] - u[1]);      # Resp. a11 := 0.1214000000
[ > a12 := (v[3] - v[2])/(u[3] - u[2]);      # Resp. a12 := 0.1083333333
```

These are the $n = 2$ first divided differences (there would be more if you had more data). Now comes the $n - 1 = 1$ second divided difference (there is only one; again, there would be more if you had more data).

```
[ > a21 := (a12 - a11)/(u[3] - u[1]);      # Resp. a21 := -0.006533333350
[ > p := v[1] (x - u[1])*a11 + (x - u[1])*(x - u[2])*a21;
  p := 1.108200000 + 0.1214000000 x - 0.006533333350 (x - 8.0) (x - 8.5)
[ > p := expand(p);
  p := 0.6639333322 + 0.2292000003 x - 0.006533333350 x^2
```

This agrees with the above response to `interp`. The 1.1082, in the above, is $v[1] - u[1]*a11$, a simplification that disguises the actual formula.

Similar Material in AEM: Sec. 19.2

EXAMPLE 19.8 SPLINE INTERPOLATION

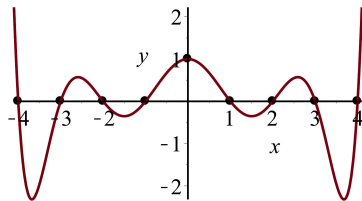
Higher order polynomials for interpolating data $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ tend to oscillate between nodes and become quite large outside of (x_0, x_n) ; see Fig. A, where $n = 8$ and the nodes are at $x = -4, -3, \dots, 4$, and $y(0) = 1$ and the other y 's are zero. Obtain this figure by typing (type `?interp`)

```
[ > u := [-4,-3,-2,-1,0,1,2,3,4];      # Resp. u := [-4,-3,-2,-1,0,1,2,3,4]
[ > v := [0,0,0,0,1,0,0,0,0];          # Resp. v := [0,0,0,0,1,0,0,0,0]
[ > p := interp(u, v, x);      # Resp. p := 1/576 x^8 - 5/96 x^6 + 91/192 x^4 - 205/144 x^2 + 1
[ > pts := seq([u[j], v[j]], j = 1..9);
  pts := [-4,0], [-3,0], [-2,0], [-1,0], [0,1], [1,0], [2,0], [3,0], [4,0]
[ > P1 := plot(pts, style = point, symbol = solidcircle, symbolsize = 20):
[ > P2 := plot(p, x = -4.1..4.1):
[ > with(plots):
[ > display(P1, P2, labels = [x, y], scaling = constrained);      # 19.8 A
```

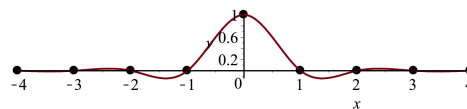
Large oscillations are avoided if, instead of a single polynomial of high degree, you choose n polynomials p_j of lower degree, say, of degree 3 (cubical polynomials), p_1 between (x_0, y_0) and (x_1, y_1) , then p_2 between (x_1, y_1) and (x_2, y_2) , and so on, such that the function g , consisting of these polynomials, interpolates the given data, that is, $g(x_j) = y_j$ ($j = 0, \dots, n$), and is twice continuously differentiable. This g is called a (cubic) **spline**. If you impose two further conditions, for instance, $g''(x_0) = 0$, $g''(x_n) = 0$, then g is uniquely determined. A spline satisfying these two additional conditions is called a **natural spline**. (Other additional conditions of practical interest are $g'(x_0) = k_0$, $g'(x_n) = k_n$, where k_0 and k_n are given numbers.) To obtain the natural spline for the above data, type `?spline` for information and

```
[ > u := [-4,-3,-2,-1,0,1,2,3,4];      # Resp. u := [-4,-3,-2,-1,0,1,2,3,4]
```

```
[ > v := [0,0,0,0,1,0,0,0]; # Resp. v := [0,0,0,0,1,0,0,0]
[ > spl := spline(u, v, x, 3);
Because  $n = 8$ , this spline consists of 8 polynomials, as you can see from the response,
which we omit for reasons of space. Plot spl by typing
[ > P1 := plot(pts, style = point, symbol=solidcircle, symbolsize = 20):
[ > P2 := plot(spl, x = -4..4):
[ > with(plots):
[ > display(P1, P2, labels = [x, y], scaling = constrained); # 19.8 B
```



Example 19.8.A. Oscillation of a high-order polynomial



Example 19.8.B. Cubic spline for the data in Fig. A

Similar Material in AEM: Sec. 19.4

EXAMPLE 19.9 NUMERIC INTEGRATION

Integrate $f(x) = \exp(-x^2)$ from 0 to 1, by the usual methods, choosing $n = 10$ steps of size $h = 0.1$.

Solution. Type the integrand

```
[ > f(x) := exp(-x^2); # Resp. f(x) := e^{-x^2}
```

Rectangular rule. This rule is usually not accurate enough. For information type `?leftsum`, `?middlesum`, `?rightsum`. Then type

```
[ > with(Student[Calculus1]):
```

The `output = sum` produces the formula for the approximation.

```
[ > ApproximateInt(f(x), x = 0..1, method = midpoint, partition = 13,
  output = sum);
```

$$\frac{1}{13} \sum_{i=0}^{12} e^{-\left(\frac{1}{13}i + \frac{1}{26}\right)^2}$$

```
[ > evalf[7](%); # Resp. 0.7470056
```

There are 13 exponentials that are computed.

Trapezoidal rule. Type `?trapezoid` and then

```
[ > ApproximateInt(f(x), x = 0..1, method = trapezoid, partition = 12,
    output = sum);
```

$$\frac{1}{24} \sum_{i=0}^{11} \left(e^{-\frac{1}{144} i^2} + e^{-\left(\frac{1}{12} i + \frac{1}{12}\right)^2} \right)$$

```
[ > evalf[7](%); # Resp. 0.7463984
```

There are 13 exponentials that are computed.

Simpson's rule. Type `?simpson` and then (use half as many partitions)

```
[ > ApproximateInt(f(x), x = 0..1, method = simpson, partition = 6,
    output = sum);
```

$$\frac{1}{36} \sum_{i=0}^5 \left(e^{-\frac{1}{36} i^2} + 4 e^{-\left(\frac{1}{6} i + \frac{1}{12}\right)^2} + e^{-\left(\frac{1}{6} i + \frac{1}{6}\right)^2} \right)$$

There are 13 exponentials that are computed.

```
[ > Sim := evalf[8](%); # Resp. Sim := 0.74682453
```

```
[ > ExInt := evalf(int(exp(-x^2), x = 0..1));
```

ExInt := 0.7468241330

```
[ > (ExInt - Sim)/ExInt; # Resp. -5.267639095 10^-7
```

This is exact to 5S with 12 function evaluations.

Gauss integration generally requires less work. For nodes and coefficients see [1], pp. 916–919, and AEM, p. 878. (The nodes are not equally spaced.) Setting $x = (t + 1)/2$, you integrate over t from -1 to 1 ; because $dx = dt/2$, you obtain

$$(A) \quad \int_0^1 \exp(-x^2) dx = \frac{1}{2} \int_{-1}^1 \exp\left(-\frac{1}{4}(t+1)^2\right) dt.$$

Choosing $n = 3$, the nodes (t -values) are $-\sqrt{\frac{3}{5}}$, 0 , $\sqrt{\frac{3}{5}}$, and the coefficients are $5/9, 8/9, 5/9$. You thus obtain, from (a),

```
[ > 1/2*(5/9*exp(-1/4*(-sqrt(3/5) + 1)^2) + 8/9*exp(-1/4)
    + 5/9*exp(-1/4*(sqrt(3/5) + 1)^2));
```

$$\frac{5}{18} e^{-\frac{1}{4} \left(-\frac{1}{5} \sqrt{15} + 1\right)^2} + \frac{4}{9} e^{-\frac{1}{4}} + \frac{5}{18} e^{-\frac{1}{4} \left(\frac{1}{5} \sqrt{15} + 1\right)^2}$$

```
[ > Gau := evalf(%); # Resp. Gau := 0.7468145842
```

```
[ > (ExInt - Gau)/ExInt; # Resp. 0.00001278587498
```

The 4S accuracy obtained from this short (3 function evaluations) calculation is astonishing.

Similar Material in AEM: Sec. 19.5

Problem Set for Chapter 19

Pr.19.1 (Quadratic equation) Solve $x^2 - 77x + 0.1 = 0$ by the two methods in Example 19.1 in this Guide. (AEM Sec. 19.1)

Pr.19.2 (Bisection method) Solve $\cos x - \sin x$. (See Example 19.3 in this Guide. AEM Sec. 19.2)

Pr.19.3 (Fixed-point iteration) Solve $x \cosh x = 1$ (appearing in connection with vibrations of beams). (*AEM* Sec. 19.2)

Pr.19.4 (Fixed-point iteration) Find a solution of $x^4 - x^2 - 10 = 0$, starting from $x_0 = 1$. (*AEM* Sec. 19.2)

Pr.19.5 (Newton's iteration method) Design a Newton iteration for cube roots and compute the cube root of 11, starting from $x_0 = 2$. (*AEM* Sec. 19.2)

Pr.19.6 (Secant method) Find a zero of $f(x) = 1 - x^2/4 + x^4/64 - x^6/2304 + x^8/147456 - x^{10}/14745600$ near $x = 5$ by using $x_0 = 4.0$ and $x_1 = 8$. ($f(x)$ is a partial sum of the Maclaurin series of the Bessel function $J_0(x)$, whose zeros are important in connection with vibrations of membranes.) (*AEM* Sec. 19.2)

Pr.19.7 (Polynomial interpolation) Find the values of the Gamma function at 1.02 and 1.07 by cubic interpolation of the values 1.00000, 0.97844, 0.96874, 0.95973 at 1.00, 1.04, 1.06, 1.08, respectively, using the command `interp`. (*AEM* Sec. 19.3)

Pr.19.8 (Newton's forward difference interpolation formula) Solve Pr.19.7 by the formula

$$f(x) = f_0 + r \Delta f_0 + \frac{1}{2!} r(r-1) \Delta^2 f_0 + \frac{1}{3!} r(r-1)(r-2) \Delta^3 f_0 + \cdots$$

where $r = (x - x_0)/h$ and h is the distance between the equally spaced adjacent nodes. (*AEM* Sec. 19.3)

Pr.19.9 (High-order polynomial) Find the polynomial that interpolates (j, y_j) , where $j = -5, -4, \dots, 0, \dots, 4, 5$ and $y_0 = 1$, $y_j = 0$ otherwise. (*AEM* Sec. 19.4)

Pr.19.10 (Experiment on oscillation of interpolation polynomials) Explore, by experimentation and plotting, how the oscillations of interpolation polynomials for equally spaced nodes and function values 0 except for a single 1 (as in Example 19.8 in this Guide and in the previous problem) are growing with increasing degree n .

Pr.19.11 (Spline interpolation) Find the natural cubic spline for the data in Pr.19.9. (*AEM* Sec. 19.4)

Pr.19.12 (Rectangular rule) Integrate $2 \exp(-x^2)/\sqrt{\pi}$ from 0 to 2 by the rectangular rule with $h = 0.2$ that use the function value (i) at the left endpoint, (ii) at the midpoint, (iii) at the right endpoint of each of the ten subintervals. Give reasons why (i) yields the largest result and (iii) the smallest. Type `?ApproximateInt;` for information (see also Example 19.9 in this Guide). (*AEM* Sec. 19.5)

Pr.19.13 (Experiment on accuracy of integration rules) Integrate $\cos(x^2)$ from 0 to $\sqrt{\frac{\pi}{2}}$, first by the command `evalf(int(...))` and then by the rectangular rule ((ii) in the previous problem), the trapezoidal rule, and Simpson's rule. In each case find, experimentally, the number of subintervals that gives 6S accuracy (compute the relative errors), thereby noting the superiority of Simpson's rule. ($\cos(x^2)$ is the integrand of the **Fresnel integral** $C(x)$. Maple uses $\cos(\pi t^2/2)$ as the integrand of the Fresnel integral instead of the more common $\cos(x^2)$. Type `?FresnelC`.) (*AEM* Sec. 19.5)

- Pr.19.14 (Simpson's rule)** Obtain a 10S-value of $\ln 5$ by evaluating a suitable integral by Simpson's rule. (*AEM* Sec. 19.5)
- Pr.19.15 (Gauss integration)** Calculate the sine integral $\text{Si}(x)$ (defined as the integral of $(\sin u)/u$ from $u = 0$ to $u = x$) for $x = 1$ by Gauss integration with $n = 3$ nodes and find the accuracy by comparing with the value obtained by the command `int`. (*AEM* Sec. 19.5)

Chapter 20

Numeric Linear Algebra

Content. Gauss, Doolittle, Cholesky methods (Exs. 20.1–20.3, Prs. 20.1–20.5)
Gauss-Jordan, Gauss-Seidel methods (Exs. 20.4, 20.5, Prs. 20.6–20.8)
Norms, condition numbers (Ex. 20.6, Prs. 20.9–20.12)
Fit by least squares (Ex. 20.7, Prs. 20.13–20.16)
Approximation of eigenvalues (Exs. 20.8–20.10, Prs. 20.17–20.20)

LinearAlgebra package. Load it by typing `with(LinearAlgebra)`. Type `?LinearAlgebra`. In addition to the commands used in Chaps. 6 and 7 you may use `LUDecomposition`, `ReducedRowEchelonForm`, `Norm`, `Norm(...,1)`, `Norm(...,Frobenius)`, `LeastSquares`.

Examples for Chapter 20

EXAMPLE 20.1 GAUSS ELIMINATION. PIVOTING

To keep this chapter independent of Chaps. 7 and 8 on matrices and eigenvalue problems, we begin by describing the implementation of the Gauss elimination on the computer.

Solve the linear system

$$\begin{aligned} 2x_1 + 6x_2 + 12x_3 &= 19 \\ 4x_1 + 7x_2 + 11x_3 &= 9, \\ 10x_1 + 0x_2 + 6x_3 &= -7 \end{aligned}$$

in matrix form $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = [a_{jk}] = \begin{bmatrix} 2 & 6 & 12 \\ 4 & 7 & 11 \\ 10 & 0 & 6 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 19 \\ 9 \\ -7 \end{bmatrix}.$$

Solution. Load the `LinearAlgebra` package by typing

```
[ > with(LinearAlgebra):
```

Type the matrix and the vector and apply the command `LinearSolve`. (Type `?LinearSolve` for information.)

```
[ > A := Matrix([[2, 6, 12], [4, 7, 11], [10, 0, 6]]);
```

$$A := \begin{bmatrix} 2 & 6 & 12 \\ 4 & 7 & 11 \\ 10 & 0 & 6 \end{bmatrix}$$

```

[ > b := <19, 9, -7>;                                     # Resp. b :=  $\begin{bmatrix} 19 \\ 9 \\ -7 \end{bmatrix}$ 

  > x := LinearSolve(A, b);                                # Resp. x :=  $\begin{bmatrix} -\frac{5}{2} \\ -2 \\ 3 \end{bmatrix}$ 

```

This gave the solution but no indication on the method by which it was obtained.

The **Gauss elimination** and **back substitution** can be done by the commands `GaussianElimination` (applied to the augmented matrix) and `BackSubstitution`. (Type

`?GaussianElimination` and `?BackSubstitution` (found in `Student[NumericalAnalysis]`) for information.)

```

[ > with(Student[NumericalAnalysis]):

  > A1 := <A | b>;                                         # Resp. A1 :=  $\begin{bmatrix} 2 & 6 & 12 & 19 \\ 4 & 7 & 11 & 9 \\ 10 & 0 & 6 & -7 \end{bmatrix}$ 

  > B := GaussianElimination(A1);                         # Resp. B :=  $\begin{bmatrix} 2 & 6 & 12 & 19 \\ 0 & -5 & -13 & -29 \\ 0 & 0 & 24 & 72 \end{bmatrix}$ 

  > BackSubstitution(SubMatrix(B, 1..3, 1..3),
    convert(SubMatrix(B, 1..3, 4), Vector));
     $\begin{bmatrix} -\frac{5}{2} \\ -2 \\ 3 \end{bmatrix}$ 

```

Details of the Gauss elimination. You must do **partial pivoting**. Use Row 3 as pivot row because it contains the largest entry in the first column. Thus interchange Rows 1 and 3 by typing

```

[ > with(Student[LinearAlgebra]):

  > A2 := SwapRow(A1, 1, 3);                               # Resp. A2 :=  $\begin{bmatrix} 10 & 0 & 6 & -7 \\ 4 & 7 & 11 & 9 \\ 2 & 6 & 12 & 19 \end{bmatrix}$ 

```

Eliminate $a_{21} = 4$ from Row 2 by the command `-A2[2,1]/A2[1,1]`, which adds $-4/10 = -2/5$ times Row 1 to Row 2,

```
[ > A3 := AddRow(A2, 2, 1, -A2[2,1]/A2[1,1]);
```

$$A3 := \begin{bmatrix} 10 & 0 & 6 & -7 \\ 0 & 7 & \frac{43}{5} & \frac{59}{5} \\ 2 & 6 & 12 & 19 \end{bmatrix}$$

then $a_{31} = 2$ from Row 3

```
[ > A4 := AddRow(A3, 3, 1, -A3[3,1]/A3[1,1]);
```

$$A4 := \begin{bmatrix} 10 & 0 & 6 & -7 \\ 0 & 7 & \frac{43}{5} & \frac{59}{5} \\ 0 & 6 & \frac{54}{5} & \frac{102}{5} \end{bmatrix}$$

The pivot row in the next step is Row 2 of **A4**. Add $-6/7$ times Row 2 to Row 3.

```
[ > A5 := AddRow(A4, 3, 2, -A4[3,2]/A4[2,2]);
```

$$A5 := \begin{bmatrix} 10 & 0 & 6 & -7 \\ 0 & 7 & \frac{43}{5} & \frac{59}{5} \\ 0 & 0 & \frac{24}{7} & \frac{72}{7} \end{bmatrix}$$

Now **back substitution** begins with x_3 and ends with x_1 . Denote the components of the solution vector \mathbf{x} by $\mathbf{x}[1]$, $\mathbf{x}[2]$, $\mathbf{x}[3]$. Remember that **A4[j,k]** picks the entry of **A4** in Row j and Column k .

```
[ > x[3] := A5[3,4]/A5[3,3]; # Resp. x3 := 3
[ > x[2] := 1/A5[2,2]*(A5[2,4] - A5[2,3]*x[3]); # Resp. x2 := -2
[ > x[1] := 1/A5[1,1]*(A5[1,4] - A5[1,3]*x[3] - A5[1,2]*x[2]);
x1 := -5/2
```

This is the end. The results agree.

Pivot command. The command **Pivot(A1, j, k)** picks equation j as the pivot equation (the pivot row) and the entry in column k as the pivot and eliminates x_k from all the other equations. Then your analysis takes the form (look back at **A1** and also note that in using the pivot command the rows are not interchanged)

```
[ > B1 := Pivot(A1, 3, 1); # Resp. B1 :=
```

$$\begin{bmatrix} 0 & 6 & \frac{54}{5} & \frac{102}{5} \\ 0 & 7 & \frac{43}{5} & \frac{59}{5} \\ 10 & 0 & 6 & -7 \end{bmatrix}$$

```
[ > B2 := DeleteRow(B1, 3..3); # Resp. B2 :=
```

$$\begin{bmatrix} 0 & 6 & \frac{54}{5} & \frac{102}{5} \\ 0 & 7 & \frac{43}{5} & \frac{59}{5} \end{bmatrix}$$

```

> B3 := Pivot(B2, 1, 2);           # Resp. B3 :=  $\begin{bmatrix} 0 & 6 & \frac{54}{5} & \frac{102}{5} \\ 0 & 0 & -4 & -12 \end{bmatrix}$ 

```

Do back substitution as before, with the necessary changes of notation.

```

> x[3] := 1/B3[2, 3]*B3[2, 4];      # Resp.  $x_3 := 3$ 
> x[2] := 1/B2[2, 2]*(B2[2, 4] - B2[2,3]*x[3]);      # Resp.  $x_2 := -2$ 
> x[1] := 1/B1[3,1]*(B1[3,4] - B1[3,3]*x[3] - B1[3, 2]*x[2]);

```

$$x_1 := -\frac{5}{2}$$

SwapRow moved the first pivot equation up, whereas **Pivot** did not. This causes the difference in notations shown in the present formulas, compared to the previous ones.

Similar Material in AEM: Sec. 20.1

EXAMPLE 20.2 DOOLITTLE LU-FACTORIZATION

A linear system $\mathbf{Ax} = \mathbf{b}$ of n equations in n unknowns can be solved by first factorizing $\mathbf{A} = \mathbf{LU}$, where \mathbf{L} is lower triangular and \mathbf{U} is upper triangular. Then $\mathbf{Ax} = \mathbf{L(Ux)} = \mathbf{b}$. Set $\mathbf{Ux} = \mathbf{y}$ and solve $\mathbf{L(Ux)} = \mathbf{Ly} = \mathbf{b}$ for \mathbf{y} . Then solve $\mathbf{Ux} = \mathbf{y}$ for \mathbf{x} . It can be proved that, if \mathbf{A} is nonsingular, its rows can be reordered such that the resulting matrix has an **LU**-factorization. This factorization is obtained by the command **LUdecomposition**, as shown below. Type **?LUdecomposition** for information. The response is made up of three matrices: **p** – the permutation matrix that indicates what rows had to be swapped, **L** – the lower triangular matrix, and **U** – the upper triangular matrix. Note that **p.L.U = A**.

Solve the linear system in Example 20.1 in this Guide by **LU**-factorization.

Solution. Type the matrix **A** and the vector **b**. Load the **LinearAlgebra** package. Type the command **LUdecomposition**.

```

> A := Matrix([[2, 6, 12], [4, 7, 11], [10, 0, 6]]);

```

$$A := \begin{bmatrix} 2 & 6 & 12 \\ 4 & 7 & 11 \\ 10 & 0 & 6 \end{bmatrix}$$

```

> b := <19, 9, -7>;

```

Resp. $b := \begin{bmatrix} 19 \\ 9 \\ -7 \end{bmatrix}$

```

> with(LinearAlgebra):
> (p, L, U) := LUdecomposition(A);

```

$$p, L, U := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 5 & 6 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 6 & 12 \\ 0 & -5 & -13 \\ 0 & 0 & 24 \end{bmatrix}$$

Because **L** has the main diagonal 1, 1, 1, this is **Doolittle's factorization**. Note that **U** is the triangular matrix as obtained by the Gauss elimination, and **L** is the

matrix of the multipliers in that method. $\mathbf{p} = \mathbf{I}$ indicates no swaps were used. The two triangular systems $\mathbf{L}\mathbf{y} = \mathbf{b}$ and $\mathbf{U}\mathbf{x} = \mathbf{y}$ can now readily be solved, for instance, by

$\left[\begin{array}{l} & & \\ & & \\ & & \end{array} \right]$	$\begin{array}{l} > \mathbf{y} := \text{LinearSolve}(\mathbf{L}, \mathbf{b}); \\ \\ > \mathbf{x} := \text{LinearSolve}(\mathbf{U}, \mathbf{y}); \end{array}$	$\begin{array}{l} \# \text{ Resp. } \mathbf{y} := \\ \\ \# \text{ Resp. } \mathbf{x} := \end{array}$	$\begin{array}{l} \left[\begin{array}{c} 19 \\ -29 \\ 72 \end{array} \right] \\ \\ \left[\begin{array}{c} \frac{5}{2} \\ -2 \\ 3 \end{array} \right] \end{array}$
---	--	--	---

Similar Material in AEM: Sec. 20.2

EXAMPLE 20.3 CHOLESKY FACTORIZATION

Solve the following system by Cholesky's method.

$$\begin{array}{rcl} 15x_1 + 6x_2 - 11x_3 & = & 10 \\ 6x_1 + 18x_2 - 3x_3 & = & -58 \\ -11x_1 - 3x_2 + 13x_3 & = & 30 \end{array}$$

Solution. This method solves linear systems $\mathbf{Ax} = \mathbf{b}$ with symmetric and positive definite \mathbf{A} . The present matrix is symmetric. You need not check for positive definiteness; if \mathbf{A} were not positive definite, the calculations would lead into complex entries.

The method uses the factorization $\mathbf{A} = \mathbf{LL}^T$ and then solves $\mathbf{Ax} = \mathbf{L}(\mathbf{L}^T\mathbf{x}) = \mathbf{b}$ in two steps as in the previous example, namely, $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{L}^T\mathbf{x} = \mathbf{y}$. Type the coefficient matrix \mathbf{A} , then \mathbf{b} , and then the factorization command with `method = 'Cholesky'`.

$\left[\begin{array}{l} & & \\ & & \\ & & \end{array} \right]$	$\begin{array}{l} > \mathbf{A} := \text{Matrix}([[16, 6, -11], [6, 18, -3], [-11, -3, 13]]); \\ \\ > \mathbf{b} := \langle 10, -58, 30 \rangle; \\ \\ > \text{with}(\text{LinearAlgebra}): \\ \\ > \mathbf{L} := \text{LUdecomposition}(\mathbf{A}, \text{method} = \text{'Cholesky'}); \end{array}$	$\begin{array}{l} \\ \\ \\ \\ \\ \end{array}$	$\begin{array}{l} \\ \\ \\ \\ \\ \end{array}$
---	--	---	---

From this you can read the answer $x_1 = -5/2$, $x_2 = -2$, $x_3 = 3$.

Inverse. Gauss-Jordan reduces the 3×6 matrix $\mathbf{B} = [\mathbf{A}, \mathbf{I}]$ (\mathbf{I} the unit matrix) to $\mathbf{A}^{-1}\mathbf{B} = [\mathbf{A}^{-1}\mathbf{A}, \mathbf{A}^{-1}\mathbf{I}] = [\mathbf{I}, \mathbf{A}^{-1}]$, so that you can read the inverse of \mathbf{A} directly from the transformed matrix. Type [?DiagonalMatrix](#) for information and then

```

> Unit := DiagonalMatrix(<1, 1, 1>);      # Resp. Unit :=
      [ 1  0  0 ]
      [ 0  1  0 ]
      [ 0  0  1 ]

> B1 := <A | Unit>;                        # Resp. B1 :=
      [ 2  6  12  1  0  0 ]
      [ 4  7  11  0  1  0 ]
      [ 10 0   6  0  0  1 ]

> B2 := ReducedRowEchelonForm(B1);
      B2 :=
      [ 1  0  0  -7/40  3/20  3/40 ]
      [ 0  1  0 -43/120 9/20  -13/120 ]
      [ 0  0  1  7/24  -1/4  1/24 ]

> InvA := SubMatrix(B2, 1..3, 4..6);
      InvA :=
      [ -7/40  3/20  3/40 ]
      [ -43/120 9/20  -13/120 ]
      [ 7/24  -1/4  1/24 ]

```

In the command `SubMatrix`, `1..3` refers to the rows and `4..6` to the columns of `B2` of which the SubMatrix of `B2` (the inverse of \mathbf{A}) consists.

Similar Material in AEM: Sec. 20.2

EXAMPLE 20.5 GAUSS-SEIDEL ITERATION FOR LINEAR SYSTEMS

Solve the following linear system by **Gauss-Seidel iteration**, starting from the vector $[100 \ 100 \ 100 \ 100]^T$. (Systems of this kind appear in connection with numeric methods for partial differential equations.)

$$\begin{aligned}
 2.00x_1 - 0.35x_2 - 0.46x_3 + 0.20x_4 &= 22 \\
 -0.25x_1 + 5.00x_2 + 0.60x_3 - 0.25x_4 &= 44 \\
 -0.65x_1 + 0.99x_2 + 3.00x_3 - 0.33x_4 &= 55 \\
 1.00x_1 - 0.25x_2 - 0.25x_3 + 4.00x_4 &= 66
 \end{aligned}$$

Solution. In $\mathbf{Ax} = \mathbf{b}$ assume that \mathbf{A} has the main diagonal entries 1, ..., 1 (which can be accomplished by division, provided all the main diagonal entries are originally different from 0). Write $\mathbf{A} = \mathbf{I} + \mathbf{L} + \mathbf{U}$, where \mathbf{I} is the unit matrix, \mathbf{L} is lower

triangular part of \mathbf{A} , and \mathbf{U} is upper triangular part of \mathbf{A} . Then $(\mathbf{I} + \mathbf{L} + \mathbf{U})\mathbf{x} = \mathbf{b}$, so that $\mathbf{x} = \mathbf{b} - \mathbf{L}\mathbf{x} - \mathbf{U}\mathbf{x}$. This lead to the iteration $\mathbf{x}^{(m+1)} = \mathbf{b} - \mathbf{L}\mathbf{x}^{(m+1)} - \mathbf{U}\mathbf{x}^{(m)}$ because you always use the latest available approximation, which is the $(m+1)$ st below the main diagonal and the m th above it. In terms of components,

$$x_j^{(m+1)} = b_j - \sum_{k=1}^{j-1} a_{jk} x_k^{(m+1)} - \sum_{K=j+1}^n a_{jK} x_K^{(m)}.$$

Hence type $\mathbf{A}, \mathbf{b}, \mathbf{x}_0$ and then the program (which includes divisions to make the main diagonal entries 1). n is the number of unknowns and N is the number of steps. Note that you need no superscripts $(m+1)$ and (m) because if a component has been computed, it is used at once, so that you can overwrite its previous value, which is no longer needed.

```
[ > with(LinearAlgebra):
> A := Matrix([[2, -0.35, -0.46, 0.2], [-0.25, 5, 0.6, -0.25],
               [-0.65, 0.99, 3, -0.33], [1.0, -0.25, -0.25, 4]]);
               A := 
$$\begin{bmatrix} 2 & -0.350000 & -0.460000 & 0.200000 \\ -0.250000 & 5 & 0.600000 & -0.250000 \\ -0.650000 & 0.990000 & 3 & -0.330000 \\ 1.000000 & -0.250000 & -0.250000 & 4 \end{bmatrix}$$

> b := <22, 44, 55, 66>; # Resp. b := 
$$\begin{bmatrix} 22 \\ 44 \\ 55 \\ 66 \end{bmatrix}$$

> n := 4: N := 8:
  x := [100, 100, 100, 100]:
  for m from 0 to N-1 do
    S := add(A[1,K].x[K], K = 2..n):
    x[1] := evalf((b[1] - S)/A[1,1]):
    for j from 2 to n do
      S := add(A[j,k].x[k], k = 1..j - 1)
        + add(A[j,K].x[K], K = j + 1..n):
      x[j] := evalf((b[j]-S)/A[j,j]):
    end do:
    print(x);
  end do:
               [41.500000, 3.875000, 37.046250, 8.682578]
               [19.330505, 5.755104, 21.577509, 13.375662]
               [15.632404, 7.661102, 20.663513, 14.362187]
               [15.657082, 7.821342, 20.724499, 14.369845]
               [15.698385, 7.816472, 20.735897, 14.359927]
               [15.701146, 7.814746, 20.735974, 14.359133]
               [15.700941, 7.814687, 20.735862, 14.359174]
               [15.700901, 7.814700, 20.735853, 14.359184]
```



```
[ > Norm(A, infinity);                                # Resp. 18
  > Norm(A, Frobenius);                                # Resp. 17
Condition numbers. The definition of the condition number of a matrix A is
 $\kappa(A) = \|A\| \|A^{-1}\|$  and depends on the choice of a norm, but if  $\kappa(A)$  is large in one
norm, it tends to be large in others, indicating that A is ill-conditioned. This is
illustrated by the following values.
  > evalf[5](ConditionNumber(A, 1));                    # Resp. 23.016000
  > ConditionNumber(A, infinity);                        # Resp.  $\frac{108}{5}$ 
  > evalf[5](ConditionNumber(A, Frobenius));            # Resp. 15.830000
```

Similar Material in AEM: Sec. 20.4

EXAMPLE 20.7 FITTING DATA BY LEAST SQUARES

Fit a straight line and a quadratic polynomial through the four points $(-1.9, 0.108)$, $(-0.9, 1.099)$, $(0.2, 0.808)$, $(1.3, 2.3)$ by the **principle of least squares**. For information type [?LeastSquares](#). Also find the interpolation polynomial for these data. Plot the results on common axes.

Solution. Load the [CurveFitting](#) package. Then give the command for obtaining the straight line. Note that, in this command, you first type all four x -values in brackets [...] and then all four y -values.

```
[ > restart:
  > with(CurveFitting):
  > data := [[-1.9, -0.9, 0.2, 1.3], [0.108, 1.099, 0.808, 2.3]]:
  > p1 := LeastSquares(data[1], data[2], x);
                                p1 := 1.269391 + 0.586587x
```

If you want a second or higher order polynomial, you must include the corresponding equation (with arbitrary coefficients) in the command.

```
[ > p2 := LeastSquares(data[1], data[2], x, curve = a*x^2 + b*x + c);
                                p2 := 1.123962 + 0.650453x + 0.108088x^2
```

If you request a cubic polynomial, because you have four points, you get the unique interpolation polynomial passing precisely through the points.

```
[ > p3 := LeastSquares(data[1], data[2], x, curve=a*x^3 + b*x^2 + c*x + d);
                                p3 := 0.825886 - 0.203419x + 0.4865289x^2 + 0.417080x^3
```

To compute the data points, type

```
[ > points := [seq([data[1, j], data[2, j]], j = 1..4)];
                                points := [[-1.900000, 0.108000], [-0.900000, 1.099000], [0.200000, 0.808000],
                                [1.300000, 2.300000]]
```

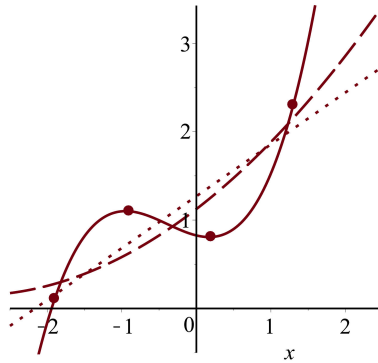
Here, `data[1, 1]` gives the first x -value and `data[2, 1]` gives the first y -value. Because $j = 1, \dots, 4$, you get 4 pairs of coordinates of the 4 data points. Then plot the three polynomials and the points by the commands following the figure.

```
[ > P1 := plot(p1, x = -2.5..2.5, linestyle = dot):
```

```

[ > P2 := plot(p2, x = -2.5..2.5, linestyle = dash):
[ > P3 := plot(p3, x = -2.5..2.5, y = -0.5..2.5):
[ > P4 := plot(points, x = -2..2, style = point, symbol = solidcircle,
[       symbolsize = 20):
[ > with(plots):
[ > display(P1, P2, P3, P4);

```



Example 20.7. Least squares straight line and quadratic polynomial.
Cubic interpolation polynomial

Similar Material in AEM: Sec. 20.5

EXAMPLE 20.8 APPROXIMATION OF EIGENVALUES: COLLATZ METHOD

Collatz proved the following. Let $\mathbf{A} = [a_{jk}]$ be a real $n \times n$ matrix with all entries a_{jk} positive. Choose any $\mathbf{x} = [x_j]$ with all n components positive. Compute $\mathbf{y} = [y_j] = \mathbf{A}\mathbf{x}$ and the n quotients $p_j = y_j/x_j$. Then the smallest and largest of these quotients are the endpoints of a closed interval J which contains an eigenvalue λ of \mathbf{A} . Hence the midpoint of J is an approximation of λ , and half the length of J is a corresponding error bound.

Write a program for this method and, starting from $[1 \ 1 \ 1]^T$, apply it to the matrix (20 steps)

$$\mathbf{A} = \begin{bmatrix} 0.34 & 0.13 & 0.28 \\ 0.13 & 0.17 & 0.30 \\ 0.28 & 0.30 & 0.39 \end{bmatrix}.$$

Solution. Calculate $\mathbf{y} = \mathbf{A}\mathbf{x}$ and those quotients. Then sort the latter in ascending order, so that p_1 and p_n are the endpoints of that interval. Note that both quotients are positive because all entries of \mathbf{A} and all components of \mathbf{x} are positive. In the last line of the program copy \mathbf{y} to make it the vector \mathbf{x} in the next step. (The response shows steps 1, 2, 6, and 9.)

```

[ > with(LinearAlgebra): interface(displayprecision = 6):
[ > A := Matrix([[0.34, 0.13, 0.28], [0.13, 0.17, 0.30], [0.28, 0.30, 0.39]]):

```

```

> n := 3: N := 9:
  x := <1, 1, 1>:
  for m from 1 to N do
    y := A.x:
    print('y' = Transpose(y));
    for j from 1 to n do
      p[j] := evalf[8](y[j]/x[j]);
    end do;
    p := sort(convert(p, list)):
    print('p' = p);
    print('Interval' = [p[1], p[n]]);
    x := copy(y):
    print('x' = Transpose(x));
  end do:

      y = [ 0.750000  0.600000  0.970000 ]
      p = [0.60000, 0.750000, 0.970000]
      Interval = [0.600000, 0.970000]
      x = [ 0.750000  0.600000  0.970000 ]
      y = [ 0.604600  0.490500  0.768300 ]
      p = [[0.792062, 0.806133, 0.817500]]
      Interval = [0.792062, 0.817500]
      x = [ 0.604600  0.490500  0.768300 ]

      ⋮

      y = [ 0.249308  0.201968  0.317002 ]
      p = [0.801342, 0.801342, 0.801344]
      Interval = [0.801342, 0.801344]
      x = [ 0.249308  0.201968  0.317002 ]

      ⋮

      y = [ 0.1282892  0.103929  0.163124 ]
      p = [0.801343, 0.801343, 0.801343]
      Interval = [0.801343, 0.801343]
      x = [ 0.128289  0.103929  0.163124 ]

> Eigenvalues(A);

      [ 0.801343 + 0.000000 I ]
      [ 0.149786 + 0.000000 I ]
      [ -0.051129 + 0.000000 I ]

```

Similar Material in AEM: Sec. 20.7

EXAMPLE 20.9

APPROXIMATION OF EIGENVALUES: POWER METHOD

Given a real symmetric square matrix \mathbf{A} , compute $\mathbf{y} = \mathbf{A}\mathbf{x}$, where \mathbf{x} is any nonzero vector. Compute the dot products $m_0 = \mathbf{x} \cdot \mathbf{x}$, $m_1 = \mathbf{x} \cdot \mathbf{y}$, $m_2 = \mathbf{y} \cdot \mathbf{y}$. Then the **Rayleigh quotient**, $q = m_1/m_0$, is an approximation of an eigenvalue of \mathbf{A} . A

corresponding error bound is $\delta = \sqrt{\frac{m_2}{m_0} - q^2}$.

This is the **first step** of the **power method**. In the **second step**, your new \mathbf{x} is the value of \mathbf{y} (just determined) divided by the component of \mathbf{x} that is largest in absolute value. Proceed as before. (This “**scaling**” will make one component of the new \mathbf{x} equal to 1 and the others less than or equal to 1 in absolute value. It has the effect that the sequence of vectors obtained will converge to an eigenvector of that λ approximated by the Rayleigh quotients.) In the further steps you proceed in the same fashion.

Apply this power method to the matrix in the previous example, starting from $\mathbf{x} = [1 \ 1 \ 1]^T$.

Solution. The matrix \mathbf{A} is symmetric. Hence the error bounds apply. The command `x := ...` scales \mathbf{y} and makes it the \mathbf{x} for the next step. Here `sort(y)` arranges the components of \mathbf{y} in ascending order. `sign(sort(y)[1] + sort(y)[3])` equals -1 if the absolutely largest component of \mathbf{y} is negative, and equals $+1$ if that component is positive. Hence, in both cases, the next \mathbf{x} will have a component $+1$. For reasons of space we show only the responses to steps 1, 2, and 10.

```
[ > with(LinearAlgebra): Digits = 6:
[ > A := Matrix([[0.34, 0.13, 0.28],
                  [0.13, 0.17, 0.30],
                  [0.28, 0.30, 0.39]]):
[ > x := <1, 1, 1>:    N := 10:
[ > for j from 1 to N do
    y := A.x;
    print('y' = Transpose(y));
    q := evalf[8]((x.y)/(x.x));
    print('q' = q);
    delta := evalf(sqrt((y.y)/(x.x) - q^2));
    print('delta' = delta);
    x := copy(y)*sign(sort(y)[1] + sort(y)[3])/max(abs(y[1]),
        abs(y[2]), abs(y[3]));
    print('x' = Transpose(x));
end:
      y = [ 0.750000  0.600000  0.970000 ]
           q = 0.773333
           delta = 0.151950
      x = [ 0.773196  0.618557  1.000000 ]
      y = [ 0.623299  0.505670  0.792062 ]
           q = 0.801224
           delta = 0.010042
      x = [ 0.786932  0.638422  1.000000 ]
           :
      y = [ 0.630220  0.510549  0.801343 ]
           q = 0.801343
           delta = 0.000224
      x = [ 0.786455  0.637117  1.000000 ]
```

Similar Material in AEM: Sec. 20.8

EXAMPLE 20.10

APPROXIMATION OF EIGENVALUES: QR-FACTORIZATION

This is another iteration method. We explain it for a real symmetric tridiagonal matrix $\mathbf{A} = \mathbf{A}_0$. In the **first step**, factor $\mathbf{A}_0 = \mathbf{Q}_0 \mathbf{R}_0$, where \mathbf{Q}_0 is orthogonal and \mathbf{R}_0 is upper triangular. Then compute $\mathbf{A}_1 = \mathbf{R}_0 \mathbf{Q}_0$. In the **second step**, factor $\mathbf{A}_1 = \mathbf{Q}_1 \mathbf{R}_1$. Then compute $\mathbf{A}_2 = \mathbf{R}_1 \mathbf{Q}_1$. And so on. These are similarity transformations, so that the spectrum is preserved. If the eigenvalues of \mathbf{A} are different in absolute value, the process converges to a diagonal matrix whose diagonal entries are the eigenvalues of \mathbf{A} . (For a proof, see Wilkinson [4] in Appendix 1.)

Obtain approximations of the eigenvalues of the following matrix by the method of QR-factorization.

$$\mathbf{A} = \begin{bmatrix} 5.0 & 6.7 & 0.0 \\ 6.7 & 8.0 & -1.9 \\ 0.0 & -1.9 & 4.0 \end{bmatrix}$$

Solution. The response shown corresponds to steps 1, 2, and 5 of the iteration.

```
> restart;
```

```
> with(Student[LinearAlgebra]):
```

```
> A := Matrix([[5, 6.7, 0 ],
```

```
               [6.7, 8.0, -1.9],
```

```
               [0, -1.9, 4 ]]);
```

$$A := \begin{bmatrix} 5 & 6.700000 & 0 \\ 6.700000 & 8 & -1.900000 \\ 0 & -1.900000 & 4 \end{bmatrix}$$

```
> for s from 1 to 5 do
```

```
    (Q, R) := QRDecomposition(A):
```

```
    A:= R.Q;
```

```
end;
```

$$Q, R := \begin{bmatrix} -0.598084 & 0.235805 & 0.765958 \\ -0.801433 & -0.175974 & -0.571610 \\ -0.000000 & -0.955735 & 0.294229 \end{bmatrix}, \begin{bmatrix} -8.360024 & -10.418630 & 1.522723 \\ 0.000000 & 1.987999 & -3.488590 \\ 0.000000 & 0.000000 & 2.262975 \end{bmatrix}$$

$$A := \begin{bmatrix} 13.349835 & -1.593248 & -7.897992 \cdot 10^{-10} \\ -1.593248 & 2.984332 & -2.162804 \\ 0.000000 & -2.162804 & 0.665832 \end{bmatrix}$$

$$Q, R := \begin{bmatrix} -0.992953 & -0.093463 & 0.072857 \\ 0.118505 & -0.783124 & 0.610469 \\ -0.000000 & 0.614802 & 0.788682 \end{bmatrix}, \begin{bmatrix} -13.444573 & 1.935679 & -0.256303 \\ 0.000000 & -3.517889 & 2.103099 \\ 0.000000 & 0.000000 & -0.795196 \end{bmatrix}$$

$$\begin{aligned}
 & \left[\begin{array}{l} A := \begin{bmatrix} 13.579223 & -0.416887 & -9.813654 \cdot 10^{-10} \\ -0.416887 & 4.047934 & -0.488888 \\ 0.000000 & -0.488888 & -0.627157 \end{bmatrix} \\ \\ Q, R := \begin{bmatrix} -0.999996 & -0.002751 & 0.000009 \\ 0.002751 & -0.999991 & 0.003339 \\ 0.000000 & 0.003339 & 0.999994 \end{bmatrix}, \begin{bmatrix} -13.597358 & 0.048639 & -0.000037 \\ 0.000000 & -4.080421 & 0.011361 \\ 0.000000 & 0.000000 & -0.677866 \end{bmatrix} \\ \\ A := \begin{bmatrix} 13.597441 & -0.011227 & 7.327175 \cdot 10^{-10} \\ -0.011227 & 4.080421 & -0.002263 \\ 0.000000 & -0.002263 & -0.677862 \end{bmatrix} \end{array} \right] \\
 & \left[\begin{array}{l} > \text{Eigenvalues}(A); \\ \\ \begin{bmatrix} 13.597454 \\ 4.080409 \\ -0.677863 \end{bmatrix} \end{array} \right]
 \end{aligned}$$

You see that the accuracy of the diagonal entries of the last matrix \mathbf{A} as approximations of the eigenvalues is greater than you would expect from the size of the off-diagonal entries of \mathbf{A} .

Similar Material in AEM: Sec. 20.9

Problem Set for Chapter 20

Pr.20.1 (Gauss elimination) Solve the following linear system by each of the three methods in Example 20.1 in this Guide. (*AEM* Sec. 20.1)

$$\begin{aligned}
 2x_1 & \quad -x_3 = 7 \\
 & 9x_2 + 11x_3 = -3 \\
 -x_1 + 11x_2 + 14x_3 & = 9
 \end{aligned}$$

Pr.20.2 (Gauss elimination) Solve the following linear system. (*AEM* Sec. 20.1)

$$\begin{aligned}
 4x_1 + 4x_2 + 2x_3 & = 0 \\
 3x_1 - x_2 + 2x_3 & = 0 \\
 3x_1 + 7x_2 + x_3 & = 0
 \end{aligned}$$

Pr.20.3 (Doolittle factorization) Solve the following linear system by Doolittle's method. (Sec. 20.2)

$$\begin{aligned}
 2x_1 + 5x_2 + 2x_3 & = 5.1 \\
 6x_1 + 16x_2 + 7x_3 & = 7.3 \\
 10x_1 + 32x_2 + 21x_3 & = 15.7
 \end{aligned}$$

Pr.20.4 (Doolittle factorization) Solve Pr.20.1 by Doolittle's method. (*AEM* Sec. 20.2)

Pr.20.5 (Cholesky factorization) Solve the following linear system by Cholesky's method. (*AEM* Sec. 20.2)

$$\begin{aligned} 16x_1 + 9x_2 + 8x_3 &= 12.3 \\ 9x_1 + 16x_2 + 12x_3 &= 13.6 \\ 8x_1 + 12x_2 + 13x_3 &= 20.7 \end{aligned}$$

Pr.20.6 (Gauss-Jordan elimination) Solve Pr.20.5 by Gauss-Jordan elimination. (*AEM* Sec. 20.2)

Pr.20.7 (Inverse) Find the inverse of the coefficient matrix \mathbf{A} in Pr.20.5 by the Gauss-Jordan method. (*AEM* Sec. 20.2)

Pr.20.8 (Gauss-Seidel iteration) Solve the following linear system by Gauss-Seidel iteration, starting from $[1 \ 1 \ 1]^T$. Do 5 steps. (*AEM* Sec. 20.3)

$$\begin{aligned} 5x_1 + x_2 + 2x_3 &= 19 \\ x_1 + 4x_2 - 2x_3 &= -2 \\ 2x_1 + 3x_2 + 8x_3 &= 39 \end{aligned}$$

Pr.20.9 (Vector norms) Find the l_1 , l_2 , l_∞ norms of the vectors $\mathbf{x} = [9 \ -17 \ 8 \ 3]^T$ and $\mathbf{y} = k\mathbf{x}$ on the computer. Here, k is any real number. (*AEM* Sec. 20.4)

Pr.20.10 (Matrix norms) Find the row "sum", column "sum", and Frobenius norms of the coefficient matrix in Pr.20.8 and of its square. (*AEM* Sec. 20.4).

Pr.20.11 (Condition number) Find the condition numbers κ of the coefficient matrix in Pr.20.5 for the three matrix norms defined in Example 20.6 in this Guide, by the command `ConditionNumber` and check the results by using the definition of κ . (*AEM* Sec. 20.4)

Pr.20.12 (Experiment on Hilbert matrix) The $n \times n$ Hilbert matrix is $\mathbf{H} = [h_{jk}]$, where $h_{jk} = 1/(j+k-1)$. Find the condition numbers for the three matrix norms defined in Example 20.6 in this Guide when $n = 3$. Conclude that \mathbf{H} is ill-conditioned. Go on to $n = 4$ and larger n . Study the decrease of the sequence of the values of the determinants of these matrices. (*AEM* Sec. 20.4)

Pr.20.13 (Least squares, straight line) Fit a straight line to the data $(x, y) = (0, 0), (1, 2), (2, 1), (3, 3), (5, 2), (7, 4), (9, 4), (10, 4), (11, 3)$ by least squares. Plot the points and the straight line. (See Example 20.7 in this Guide.)

Pr.20.14 (Least squares, straight line) Fit a straight line through the points $(x, y) = (300, 470), (400, 580), (500, 1030), (600, 1420), (700, 1880), (750, 2000)$ by least squares. (*AEM* Sec. 20.5)

Pr.20.15 (Least squares, quadratic parabola) Fit a quadratic parabola to the points $(x, y) = (2, 0), (3, 3), (5, 4), (6, 3), (7, 1)$ by least squares. Plot the points and the parabola. (*AEM* Sec. 20.5)

Pr.20.16 (Least squares, cubic parabola) Fit a cubic parabola to the points $(x, y) = (-2, -28), (-1, 0), (0, 3), (1, 7), (2, 15), (4, 70)$ by least squares. Plot the points and the parabola. (*AEM* Sec. 20.5)

Pr.20.17 (Eigenvalues, eigenvectors) Find the eigenvalues and eigenvectors of the matrix in Example 20.8 in this Guide by the commands [Eigenvalues](#), [Eigenvectors](#). Also find the characteristic polynomial. (Type [?CharacteristicPolynomial](#) for information.) Note that Maple's characteristic matrix $\lambda \mathbf{I} - \mathbf{A}$ differs from the more usual $\mathbf{A} - \lambda \mathbf{I}$. Accordingly, Maple's characteristic polynomial differs from the more usual form by a factor $(-1)^n$, where n is the number of rows and columns of \mathbf{A} . (*AEM* Sec. 20.6)

Pr.20.18 (Collatz method) Find an approximation of an eigenvalue of the following matrix by Collatz's method, starting from $[1 \ 1 \ 1]^T$ and doing 10 steps. (See Example 20.8 in this Guide. *AEM* Sec. 20.7)

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & 3 \\ 2 & 6 & 6 \\ 3 & 6 & 3 \end{bmatrix}$$

Pr.20.19 (Power method for eigenvalues) Find an approximation and error bounds for an eigenvalue of the matrix in Pr.20.18 by the power method, starting from $[1 \ 1 \ 1]^T$ and doing 10 steps. (*AEM* Sec. 20.8)

Pr.20.20 (QR-factorization) Using the QR-method (5 steps), find approximations of the eigenvalues of the following matrix. (*AEM* Sec. 20.9)

$$\mathbf{A} = \begin{bmatrix} 14.2 & -0.1 & 0.0 \\ -0.1 & -6.3 & 0.2 \\ 0.0 & 0.2 & 2.1 \end{bmatrix}$$

Chapter 21

Numeric Methods for ODEs and PDEs

Content. Euler methods (Exs. 21.2, 21.3, Prs. 21.1–21.5)
Runge-Kutta methods (Exs. 21.4, 21.6, 21.7, Prs. 21.6, 21.7, 21.10–21.12)
Multistep method (Ex. 21.5, Prs. 21.8, 21.9)
Laplace equation (Ex. 21.8, Pr. 21.13)
Heat equation (Ex. 21.9, Prs. 21.14, 21.15)

The commands in this chapter are those used in Chaps. 1 and 2 and (for systems of ODEs) in Chaps. 4 and 8.

Examples for Chapter 21

EXAMPLE 21.1 TWO WAYS OF WRITING AN ODE

In connection with numerics for an ODE $y' = f(x, y)$, writing, for instance,

```
[ > f := (x, y) -> 7*x^2*y + x; # Resp. f := (x, y) -> 7x^2y + x
```

seems preferable to the more usual

```
[ > g(x, y) := 7*x^2*y + x; # Resp. g(x, y) := 7x^2y + x
```

because if, say,

```
[ > h := 0.1: x(0) := 1: y(0) := 3.4:
```

then

```
[ > y(1) := y(0) + h*f(x(0), y(0)); # Resp. y(1) := 5.88
```

whereas g leaves the expression unevaluated,

```
[ > y(1) := y(0) + h*g(x(0), y(0)); # Resp. y(1) := 3.4 + 0.1g(1, 3.4)
```

The same differential equation is used in examples 20.1 20.2, and 20.3. In each case the step size and number of steps have been chosen to keep the total number of function evaluations the same. This ensures a fair comparison of the methods.

EXAMPLE 21.2 EULER METHOD

The Euler method solves **initial value problems** $y' = f(x, y)$, $y(x_0) = y_0$ numerically, according to the formula

$$(A) \quad y_{n+1} = y_n + h f(x_n, y_n) \quad (n = 0, 1, 2, \dots).$$

It is a **step-by-step method** which calculates approximate values of the solution at $x = x_0$, $x_1 = x_0 + h$, $x_2 = x_1 + h = x_0 + 2h$, etc. You choose the **stepsize** h , e.g., $h = 0.1$. Geometrically, the idea of the method is the approximation of the unknown solution curve by a polygon whose first side is tangent to the curve at x_0 .

The method is too inaccurate in practice, but is a nice illustration of the idea of step-by-step methods.

For instance, solve $y' - xe^x = 0$, $y(0) = 1$. Choose $h = 0.1$. Do $N = 16$ steps. Plot the solution curve and the approximate values obtained. To assist in later problems, we will use a procedure.

Solution. Type f in the ODE $y' = f(x, y) = xe^x$ as just explained in the previous example. Use h , N , and the initial condition $y(0) = 1$. We need to be careful with $y(0)$. Outside the procedure, $y(0)$ refers to the value of $y(x)$ at $x = 0$. Inside the procedure it refers to $y(n)$ at $n = 0$. Then calculate the approximate y -values by the procedure involving (A). Next use `printf` to make up a table showing $x(n)$, $y(n)$, and the error. The solution is $(x - 1)e^x + 2$.

```
[ > EULER := proc(f, x0, y0, h, N)
    local n;
    x(0) := x0: y(0) := y0:
    for n from 0 to N do
        y(n + 1) := y(n) + h*f(x(n), y(n));
        x(n + 1) := x(n) + h;
    end:
end:
```

Note one function evaluation per step.

```
[ > save EULER, euler:
```

```
[ > f := (x, y) -> x*exp(x);                                # Resp.  $f := (x, y) \rightarrow xe^x$ 
```

```
[ > x := 'x': y := 'y':
```

```
[ > sol := dsolve(diff(y(x), x) = f(x, y(x)), y(0) = 1);
                                sol :=  $y(x) = (x - 1)e^x + 2$ 
```

```
[ > N := 16: x0 := 0: y0 := 1: h := 0.1:
```

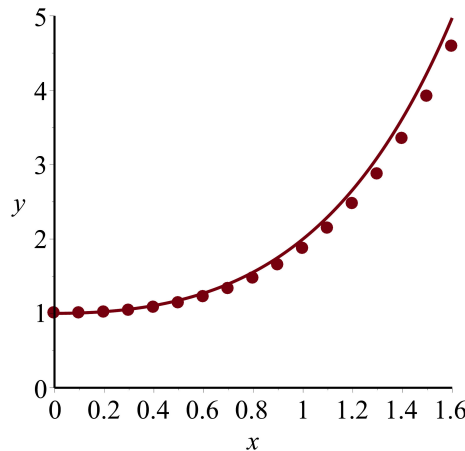
```
[ > EULER(f, x0, y0, h, N):
    printf(" x          y          err y^n");
    for n from 0 to N do
        printf("%4.2f %10.6f %10.8f^n", x(n), y(n),
            evalf(subs(x = x(n), rhs(sol))) - y(n));
    end;

    x          y          err y
0.00          1.000000    0.00000000
0.10          1.000000    0.00534617
0.20          1.011052    0.01182608
0.30          1.035480    0.01961907
```

0.40	1.075976	0.02892965
0.50	1.135649	0.03999085
0.60	1.218085	0.05306790
0.70	1.327412	0.06846248
0.80	1.468374	0.08651742
0.90	1.646418	0.10762202
1.00	1.867782	0.13221805
1.10	2.139610	0.16080647
1.20	2.470068	0.19395499
1.30	2.868482	0.23230657
1.40	3.345491	0.27658899
1.50	3.913219	0.32762554
1.60	4.585472	0.38634710

Try it. Now plot the approximate values obtained as well as the solution curve on common axes. (The $y = 0..5$ is needed so that Maple starts the y -axis at 0.)

```
> P1 := plot(seq([x(n), y(n)], n = 0..N), style = point,
               symbol = solidcircle, symbolsize = 20):
> P2 := plot(rhs(sol), x = 0..1.6, y = 0..5):
> with(plots):
> display(P1, P2, labels = [x, y]);
```



Example 21.2. Solution $(x - 1)e^x + 2$ and approximation by Euler's method

Similar Material in AEM: Sec. 21.1

EXAMPLE 21.3

IMPROVED EULER METHOD

In the **improved Euler method** (or **Euler-Cauchy method**) you compute first an auxiliary value $y_{n+1}^* = y_n + h f(x_n, y_n)$ and then the new value $y_{n+1} = y_n + \frac{1}{2}h[f(x_n, y_n) + f(x_{n+1}, y_{n+1}^*)]$.

For instance, apply this method to the initial value problem in Example 21.1. Do $N = 8$ steps with $h = 0.2$.

Solution. Type the right-hand side of the differential equation, using \rightarrow as before,

```
[ > f := (x, y) -> x*exp(x);                                # Resp.  $f := (x, y) \rightarrow xe^x$ 
[ > sol := dsolve(diff(y(x), x) = f(x, y(x)), y(0) = 1):
```

Use a program involving a do-loop similar to that in the previous example. You will see that the values obtained lie almost on the curve of the exact solution. The printout gives the x -values, y -values, y^* -values, and errors.

```
[ > MODEULER := proc(f, x0, y0, h, N)
    local n;
    x(0) := x0: y(0) := y0: ystar(0) := 0:
    for n from 0 to N do
        ystar(n + 1) := y(n) + h*f(x(n), y(n));
        y(n + 1) := y(n) + h/2*(f(x(n), y(n)) + f(x(n) + h, ystar(n + 1)));
        x(n + 1) := x(n) + h;
    end:
end:

Note two function evaluations per step —  $y$  and  $ystar$ .

[ > save MODEULER, modeuler:

[ > N := 8: x0 := 0: y0 := 1: h := 0.2:

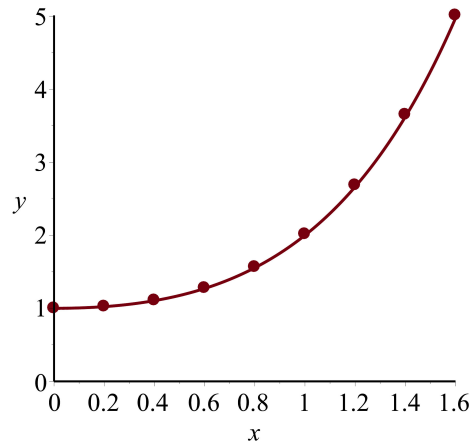
[ > MODEULER(f, x0, y0, h, N):

[ > printf(" x          y          err y\n");
    for n from 0 to N do
        printf("%4.2f %10.6f %10.8f\n", x(n), y(n), evalf(subs(x = x(n), rhs(sol)))
            - y(n));
    end;
    x          y          err y
0.00          1.000000    0.00000000
0.20          1.024428   -0.00155026
0.40          1.108529   -0.00362392
0.60          1.277529   -0.00637673
0.80          1.564900   -0.01000780
1.00          2.014771   -0.01477107
1.20          2.685013   -0.02098990
1.40          3.651155   -0.02907532
1.60          5.011368   -0.03954904
```

Error in Euler method is 0.3863. For plotting the solution curve and the values obtained, type

```
[ > P1 := plot(rhs(sol), x = 0..1.6, y = 0..5):
[ > P2 := plot(seq([x(n), y(n)], n = 0..N), style = point,
    symbol = solidcircle, symbolsize = 15):
[ > with(plots):
```

```
> display(P1, P2, labels = [x, y]);
```



Example 21.3. Solution $(x - 1)e^x + 2$ and values obtained by the improved Euler method

Similar Material in AEM: Sec. 21.1

EXAMPLE 21.4

CLASSICAL RUNGE-KUTTA METHOD (RK).

The (classical) Runge-Kutta method is an important and very accurate step-by-step method for solving initial value problems $y' = f(x, y)$, $y(x_0) = y_0$. Use this method to solve $y' = f(x, y) = x e^x$, $y(0) = 0$ (as used in the previous examples), doing $N = 4$ steps of size $h = 0.4$. Determine the error by using the exact solution $(x - 1)e^x + 2$.

Explanation of the method and solution. In each step of the method you first calculate four auxiliary values **k1**, **k2**, **k3**, **k4** by the four formulas in Lines 5-8 of the subsequent program. The new **y(n + 1)** is then obtained from **y(n)** by adding the linear combination of these **k**'s in Line 9. In Line 10 you set **x** to the new value **x(n + 1)**. This is a do-loop which begins in Line 4 and ends with **end** in Line 11.

Line 3 contains the given initial values, where **x(0)** and **y(0)** mean x and y for $n = 0$, as in the previous examples.

Lines 4-11 is the **actual program**. Lines 1-3 and 12 make it into a procedure. The opening Line 1 shows a name for it (whose choice is up to you) and the list of quantities which you must specify later (in Lines 13 and 14).

Line 2 makes the five quantities shown **local variables**. This means that they are used in **proc** without interference with earlier or later use of these letters.

Line 12 is always needed to mark the end of **proc**.

```

> RK := proc(f, x0, y0, h, N)                                # Line 1
  local n, k1, k2, k3, k4;                                    # Line 2
  x(0) := x0: y(0) := y0:                                     # Line 3
  for n from 0 to N do                                        # Line 4
    k1 := h*f(x(n), y(n));                                    # Line 5
    k2 := h*f(x(n) + h/2, y(n) + k1/2);                      # Line 6
    k3 := h*f(x(n) + h/2, y(n) + k2/2);                      # Line 7
    k4 := h*f(x(n) + h, y(n) + k3);                          # Line 8
    y(n + 1) := y(n) + 1/6*(k1 + 2*k2 + 2*k3 + k4);          # Line 9
    x(n + 1) := x(n) + h;                                     # Line 10
  end:                                                        # Line 11
end:                                                          # Line 12

```

Note four function evaluations per step.

```

> f := (x, y) -> x*exp(x);                                # Resp.  $f := (x, y) \rightarrow xe^x$ 
> sol := dsolve(diff(y(x), x)=f(x, y(x)), y(0)=1);
                                      $sol := y(x) = (x - 1)e^x + 2$ 
> N := 4: x0 := 0: y0 := 1: h := 0.4:
> RK(f, x0, y0, h, N):

```

In the next line you use the command `printf` to obtain the desired values. And, because you know the exact solution (which will *not* be the case in practice), you can obtain the errors of the approximate values computed and see that the Runge-Kutta method is much more accurate than, say, the method in the previous example.

```

> printf(" x          y          err y\n");
  for n from 0 to N do
    printf("%4.2f %10.6f %10.8f\n", x(n), y(n), evalf(subs(x = x(n),
      rhs(sol))) - y(n));
  end;

```

x	y	err y
0.00	1.000000	0.00000000
0.40	1.104923	-0.00001829
0.80	1.554940	-0.00004817
1.20	2.664120	-0.00009663
1.60	4.971994	-0.00017469

Error in modified Euler method (with the same number of function evaluations) was -0.0396 .

```

> save RK, rk:

```

where `RK` is the name of the procedure to be saved and `rk` is your choice for the file name. To **use it again** (after having filed it as just shown), type `read rk:`.

Similar Material in AEM: Sec. 21.1

EXAMPLE 21.5**ADAMS-MOULTON MULTISTEP METHOD**

The Adams-Moulton method is a step-by-step method for solving initial value problems $y' = f(x, y)$, $y(x_0) = y_0$. It is obtained as follows. Integrate the equation on both sides from x_n to x_{n+1} , obtaining

$$y(x_{n+1}) - y(x_n) = \int_{x_n}^{x_{n+1}} f(x, y(x)) dx.$$

On the right replace f by an interpolation polynomial p . Then, on the left, you have an approximation $y_{n+1} - y_n$. For p take the cubic polynomial that at $x_n, x_{n-1}, x_{n-2}, x_{n-3}$ has the values $f(x_n, y_n), \dots, f(x_{n-3}, y_{n-3})$, respectively. Then you get the **predictor formula**, `ystar(n + 1)`, in the procedure. Next take for p the cubic polynomial that at $x_{n+1}, x_n, x_{n-1}, x_{n-2}$ has the values $f(x_{n+1}, y_{n+1}^*), f(x_n, y_n), f(x_{n-1}, y_{n-1}), f(x_{n-2}, y_{n-2})$. Then you get the **corrector formula**, `y(n + 1)`, in the procedure.

The method is called a **multistep method** because, in each step, it uses the results of several previous steps. Because, at the beginning, no values are available, the method needs **starting values** at x_0 (there you can take the initial value y_0), x_1, x_2, x_3 which you must obtain by some other method, say, by Runge-Kutta, so that they are as accurate as possible. You can load the RK procedure saved in the previous example, where it was saved by `save RK, rk:`. There it was explained that you can load it for further use by the command `read rk:`. Hence you can now design a procedure for Adams-Moulton with built-in Runge-Kutta, as follows. (Type the first command first with `;` instead of `:`, to make sure that the RK procedure is available.)

```
[ > read rk:
> AdamsMoultonRK := proc(f, x0, y0, h, N)
  local n;
  x(0) := x0: y(0) := y0: ystar(0) := 0:
  RK(f, x(0), y(0), h, 2):
  for n from 3 to N do
    x(n + 1) := x(n) + h;
    ystar(n + 1) := y(n) + h/24*(55*f(x(n), y(n)) - 59*f(x(n - 1),
      y(n - 1)) + 37*f(x(n-2), y(n-2)) - 9*f(x(n-3), y(n-3)));
    y(n + 1) := y(n) + h/24*(9*f(x(n + 1), ystar(n + 1))
      + 19*f(x(n), y(n)) - 5*f(x(n-1), y(n-1)) + f(x(n-2), y(n-2)));
  end:
end:
```

You can save the procedure for later use by typing

```
[ > save AdamsMoultonRK, amrk:
```

Solve the initial value problem

$$y' = (y - x - 1)^2 + 2, \quad y(0) = 1$$

by Adams-Moulton, doing 7 steps of size 0.1.

Solution. Type the right-hand side of the ODE, then the initial values and $h = 0.1$ and $N = 7$. Then call the Adam-Moulton procedure with built-in Runge Kutta. Finally, print the values of interest, also showing the accuracy of the method.

```

[ > Digits := 10:
[ > f := (x, y) -> (y - x - 1)^2 + 2;
[ > sol := dsolve(diff(y(x), x) = f(x, y(x)), y(0) = 1);
      sol := y(x) = x + 1 + tan(x)
[ > x0 := 0: y0 := 1: h := 0.1: N := 7:
[ > AdamsMoultonRK(f, x0, y0, h, N):
[ > printf(" x          ystar          y          err y\n");
    for n from 0 to 3 do
      printf("%4.2f %10.6f %10.6f %11.8f\n", x(n), 0, y(n),
        evalf(subs(x = x(n), rhs(sol))) - y(n));
    end; for n from 4 to N do
      printf("%4.2f %10.6f %10.6f %11.8f\n", x(n), ystar(n), y(n),
        evalf(subs(x = x(n), rhs(sol))) - y(n));
    end;

```

x	ystar	y	err y
0.00	0.000000	1.000000	0.00000000
0.10	0.000000	1.200335	0.00000008
0.20	0.000000	1.402710	0.00000016
0.30	0.000000	1.609336	0.00000021
0.40	1.822715	1.822798	-0.00000486
0.50	2.046197	2.046315	-0.00001242
0.60	2.283978	2.284161	-0.00002424
0.70	2.542027	2.542332	-0.00004350

This table gives x , the predictor y^* , the corrector y , and the error, which you obtain by setting $u = y - x - 1$ and separating variables in the ODE, or by `dsolve` as shown,

Note that the error is much less in absolute value than the absolute value of the difference between predictor and corrector. No plotting is done because the errors are so small that they would not show.

EXAMPLE 21.6 CLASSICAL RUNGE-KUTTA METHOD FOR SYSTEMS (RKS)

RKS solves initial value problems for first-order systems $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$, $\mathbf{y}(x_0) = \mathbf{y}_0$, where $\mathbf{y} = [y_1, y_2, \dots, y_m]$ and $\mathbf{f} = [f_1, f_2, \dots, f_m]$. (We write m because n will denote the steps of the iteration.)

```

[ > RKS := proc (f, x, y, h, N)
    local k1, k2, k3, k4, n;
    for n from 0 to N do
      k1 := h*f(x[n], y[n]);
      k2 := h*f(x[n] + h/2, y[n] + k1/2);
      k3 := h*f(x[n] + h/2, y[n] + k2/2);
      k4 := h*f(x[n] + h, y[n] + k3);
      y[n + 1] := y[n] + (k1 + 2*k2 + 2*k3 + k4)/6;
      x[n + 1] := x[n] + h;
    end;
end:

```

For later use, you can save this procedure by typing

```
[ > save RKS, rks:
```

Use this procedure to obtain the solution (the **Airy function**) $\text{Ai}(x)$ of the **Airy equation** $y'' = xy$. Do 5 steps with $h = 0.2$.

Solution. Setting $y_1 = y$, $y_2 = y_1' = y'$ (the usual conversion of a second-order equation to a system), you obtain $y_1' = y_2$, $y_2' = xy_1$. Hence the vector function \mathbf{f} on the right-hand side of $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ is $\mathbf{f} = [y_2, xy_1]$.

```
[ > f := (x, y) -> [y[2], x*y[1]]; # Resp. f := (x, y) -> [y2, xy1]
```

Now, standard linearly independent solutions of Airy's ODE are $\text{Ai}(x)$ and $\text{Bi}(x)$ (type `?Ai`: see also Ref. [1], pp. 446 and 475, in Appendix 1 of AEM). Hence you obtain $\text{Ai}(x)$ by choosing the initial conditions $y_1(0) = \text{Ai}(0)$, $y_2(0) = \text{Ai}'(0)$. Thus type, in vector form,

```
[ > y[0] := [evalf[8](AiryAi(0)), evalf[8](subs(x = 0,
diff(AiryAi(x), x)))];
y0 := [0.35502806, -0.25881940]
```

```
[ > x[0] := 0: h := 0.2: N := 5:
```

```
[ > RKS(f, x, y, h, N):
```

```
[ > printf(" x y\n");
for n from 0 to N do
printf("%4.2f %12.8f\n", x[n], Vector(y[n]));
end;
```

x	y
0.00	0.35502806 -0.25881940
0.20	0.30370304 -0.25240464
0.40	0.25474212 -0.23583073
0.60	0.20979974 -0.21279184
0.80	0.16984597 -0.18641170
1.00	0.13529209 -0.15914686

```
[ > [evalf[10](AiryAi(1)), evalf(subs(x = 1, diff[10](AiryAi(x), x)))];
[0.1352924163, -0.1591474413]
```

EXAMPLE 21.7

CLASSICAL RUNGE-KUTTA-NYSTROEM METHOD (RKN)

The RKN method extends the classical Runge-Kutta method to initial value problems for second-order ODEs $y'' = f(x, y, y')$, $y(x_0) = K_0$, $y'(x_0) = K_1$. The program below shows the following. In each step you have to compute four quantities k_1, k_2, k_3, k_4 , in turn involving two quantities K and L (introduced merely for simplifying the notations of the k 's). From these you calculate the new y -value $\mathbf{y(n + 1)}$ and — this is the new feature of RKN over RK — a new value $\mathbf{yp(n + 1)}$ for the derivative needed in the next step, where \mathbf{p} suggests 'prime'.

Using RKN (5 steps of size 0.2), solve **Airy's equation** $y'' = xy$ for the initial values $y(0) = 0.35502806$, $y'(0) = -0.25881940$. The exact solution is $\text{Ai}(x)$. Type `?Ai`. Obtain the exact values by Maple and calculate the error of the RKN values.

Solution. Type $f = xy$, then the initial values, and then a do-loop that will calculate

the auxiliary values as well as the values for y and y' . You may also use `dsolve` to see a general solution involving the **Airy functions** Ai and Bi. Finally, use `printf` to get the results as well as the errors.

```

> RKN := proc(f, x, y, yp, h, N)
  local n, k1, k2, k3, k4, K, L;
  for n from 0 to N do
    k1 := h/2*f(x(n), y(n), yp(n));
    K := h/2*(yp(n) + k1/2);
    k2 := h/2*f(x(n) + h/2, y(n) + K, yp(n) + k1);
    k3 := h/2*f(x(n) + h/2, y(n) + K, yp(n) + k2);
    L := h*(yp(n) + k3);
    k4 := h/2*f(x(n) + h, y(n) + L, yp(n) + 2*k3);
    x(n + 1) := x(n) + h;
    y(n + 1) := y(n) + h*(yp(n) + 1/3*(k1 + k2 + k3));
    yp(n + 1) := yp(n) + 1/3*(k1 + 2*k2 + 2*k3 + k4);
  end:
end:

> f := (x, y, yp) -> x*y;

> x(0) := 0: y(0) := 0.35502806: yp(0) := -0.25881940:

> h := 0.2: N := 5:

> RKN(f, x, y, yp, h, N):

> printf(" x y err y\n");
  for n from 0 to N do
    printf("%4.2f    %10.6f    %11.8f\n", x(n), y(n),
      evalf(evalf(AiryAi(x(n)))) - y(n));
  end;
  x          y          err y
0.00      0.355028      -0.00000001
0.20      0.303703      0.00000011

0.40      0.254742      0.00000024
0.60      0.209800      0.00000032
0.80      0.169846      0.00000032
1.00      0.135292      0.00000023

> evalf[10](AiryAi(1));                                # Resp. 0.1352924163

```

The present result is somewhat more accurate than 0.1352920858 in the previous example. For a general solution type

```

> dsolve(diff(z(t), t, t) = t*z(t));

```

$$z(t) = _C1 \text{ AiryAi}(t) + _C2 \text{ AiryBi}(t)$$

($_C1$ and $_C2$ may appear interchanged.) You can save the procedure for later use by typing

```

> save RKN, rkn:

```

Similar Material in AEM: Sec. 21.3

EXAMPLE 21.8

LAPLACE EQUATION. BOUNDARY VALUE PROBLEM

Solve the **Dirichlet problem** for the **Laplace equation** $u_{xx} + u_{yy} = 0$, numerically, in the square in the figure with the grid shown, when the boundary potential equals 0 on the upper edge, 100 volts on the left edge, 200 on the bottom edge and 150 on the right edge.

Solution. The Laplace equation is replaced by a *difference equation*. This gives a linear system of 4 equations in the 4 unknown potentials $u_{jk} = u(x, y) = u(P_{jk})$, $j = 1, 2$ and $k = 1, 2$, at the 4 inner points of the grid in the figure. At each such point, -4 times the potential plus the sum of the potentials at the 4 closest neighbors equals 0. In these equations you take the 4 inner points in the order P_{11} , P_{21} , P_{12} , P_{22} . Thus for P_{11} the equation is $-4u_{11} + u_{10} + u_{21} + u_{12} + u_{01} = -4u_{11} + 100 + u_{21} + u_{12} + 200 = 0$. Hence the coefficient matrix of the linear system $\mathbf{Ax} = \mathbf{b}$ has $[-4 \ 1 \ 1 \ 0]$ as the first row and -300 as the first component of \mathbf{b} . Similarly for the other equations.

You thus obtain $\mathbf{Ax} = \mathbf{b}$ and its solution (the potential at the 4 inner points) by typing

```
[ > A := Matrix([[ -4, 1, 1, 0], [ 1, -4, 0, 1], [ 1, 0, -4, 1], [ 0, 1, 1, -4]]);
```

$$A := \begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix}$$

```
[ > b := <-300, -350, -100, -150>;
```

```
[ > with(LinearAlgebra):
```

```
[ > x := evalf[4](LinearSolve(A, b));
```

$$x := \begin{bmatrix} 131.2 \\ 143.8 \\ 81.25 \\ 93.75 \end{bmatrix}$$

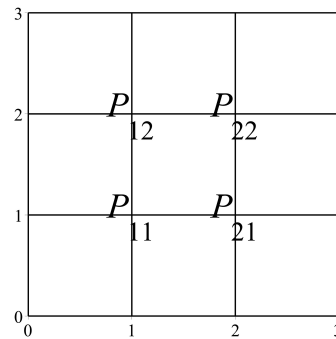
The figure is obtained by the commands (type `?textplot`

```
[ > with(plots):
```

```

> textplot([[1, 1, typeset(P[11]), 'font' = ['courier', 20]],
  [1, 2, typeset(P[12]), 'font' = ['courier', 20]],
  [2, 1, typeset(P[21]), 'font' = ['courier', 20]],
  [2, 2, typeset(P[22]), 'font' = ['courier', 20]]],
  'view' = [0..3, 0..3], axis = [gridlines = [4, color = black]]);

```



Example 21.8. Grid in the xy -plane

Program. Band matrices. Type `?BandMatrix`. We start with `p := 2:`. (A refinement of the grid will follow later by using `p := 3:`, etc.) Construct **A** as follows (see the commands below, which will be explained). Create a band matrix `c := [1, -4, 1]`: The middle component of `c` gives the diagonal entries of the band matrix **C**. “Double **C** up” to get **E**. Four 1’s from **A** are still missing in **E**. To get them, take the zero vector **v** with $2p + 1 = 5$ components. Change its first and last components to 1 by the commands `v[1] := 1:` and `v[2*p + 1] := 1:`. This gives `v := [1, 0, 0, 0, 1]`. (What looks like a detour will enable you to use it for other $p = 3$, etc. later.) The middle 0 of **v** gives the main diagonal entries of the 4×4 matrix **F**. Hence the first row of **F** is 0, 0, 1 from **v** and another 0. The second row is 0, 0, 0, 1 all from **v**. The next row is 1, 0, 0, 0 all from **v**. The last row is an extra 0 and then 1, 0, 0 from **v**. You now obtain `A := E + F`.

```

> with(LinearAlgebra):

```

```

> p := 2:

```

```

> C := BandMatrix([1, -4, 1], 1, p);           # Resp. C :=  $\begin{bmatrix} -4 & 1 \\ 1 & -4 \end{bmatrix}$ 

```

```

> E := DiagonalMatrix([seq(C, j = 1..p)]);

```

$$E := \begin{bmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 0 & 0 \\ 0 & 0 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{bmatrix}$$

```

> v := [seq(0, i = 1..(2*p + 1))];           # Resp. v := [0, 0, 0, 0, 0]

```

```

> v[1] := 1: v[2*p + 1] := 1:

```

```

> F := BandMatrix(v, p, p^2);           # Resp.  $F := \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$ 

> A := E + F;                           # Resp.  $A := \begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix}$ 

```

Now turn to the **boundary values**. These values were originally on the left-hand sides of the linear equations and were then transferred to the right, hence multiplied by -1 . Start with the vector $\mathbf{b} := \text{vector}(p^2, 0)$, and use a loop to add the boundary values to the appropriate components of \mathbf{b} . To see what is going on, write this program out for $j = 1$ and then for $j = 2$ and recall that $\mathbf{b}[1]$ comes from P_{11} and gets $\text{bot} + \text{left}$. $\mathbf{b}[2]$ comes from P_{21} and gets $\text{bot} + \text{right}$. $\mathbf{b}[3]$ comes from P_{12} and gets $\text{top} + \text{left}$. $\mathbf{b}[4]$ comes from P_{22} and gets $\text{top} + \text{right}$.

```

> top := 0: left := 100: bot := 200: right := 150:

> b := <seq(0, i = 1..p^2)>:

> for j from 1 to p do
    b[j] := b[j] - bot:
    b[(j-1)*p + 1] := b[(j-1)*p + 1] - left:
    b[j*p] := b[j*p] - right:
    b[j + p*(p-1)] := b[j + p*(p-1)] - top:
end do:

> x := convert(evalf[4](LinearSolve(A, b)), list);
x := [131.2, 143.8, 81.25, 93.75]

```

Finally, arrange the result as a $p \times p = 2 \times 2$ matrix so that the location of these values corresponds to the location of the four inner gridpoints. To achieve this, type \mathbf{X} and then use `SwapRow` (which is done here in such a way that it can be used for any p). Hence `Matrix(p, p, x)` is a $p \times p (= 2 \times 2)$ matrix with the $p^2 (= 4)$ components of \mathbf{x} as entries.

```

> X := Matrix(p, p, x);                 # Resp.  $X := \begin{bmatrix} 131.2 & 143.8 \\ 81.25 & 93.75 \end{bmatrix}$ 

> with(Student[LinearAlgebra]):

> for j from 1 to p/2 do
    SwapRow(X, j, p + 1 - j);
end;

 $\begin{bmatrix} 81.25 & 93.75 \\ 131.2 & 143.8 \end{bmatrix}$ 

```

Grid refinement. Consider $p = 3$ (9 inner gridpoints). Type $\mathbf{p} := 3$: and use the program just designed.

```

[ > p := 3:
  > C := BandMatrix([1, -4, 1], 1, p);
                                
$$C := \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{bmatrix}$$

  > E := DiagonalMatrix([seq(C, j = 1..p)]);
                                
$$E := \begin{bmatrix} -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4 \end{bmatrix}$$

  > v := [seq(0, i = 1..(2*p + 1))]; # Zero vector: 2p + 1 = 7 components
  > v[1] := 1: v[2*p + 1] := 1: # [1,0,0,0,0,0,1]
  > F := BandMatrix(v, p, p^2);
  > A := E + F;
                                
$$A := \begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}$$

  > b := <seq(0, i = 1..p^2)>:
  > for j from 1 to p do
    b[j] := b[j] - bot:
    b[(j - 1)*p + 1] := b[(j - 1)*p + 1] - left:
    b[j*p] := b[j*p] - right:
    b[j + p*(p-1)] := b[j + p*(p - 1)] - top:
  end do:

```



```

> x := convert(evalf[5](LinearSolve(A, b)), list);
      x := [139.29, 152.23, 157.14, 104.91, 112.50, 126.34, 67.857, 66.518, 85.714]

> X := Matrix(p, p, x);           # Resp. X :=
                                     [ 139.29  152.23  157.14 ]
                                     [ 104.91  112.50  126.34 ]
                                     [ 67.857  66.518  85.714 ]

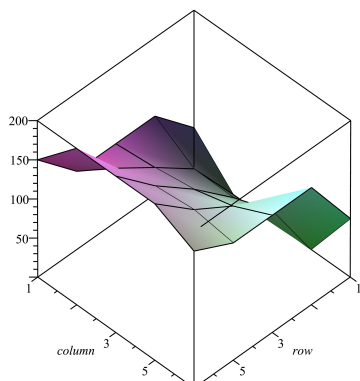
We can build a temperature grid. There are temperature discontinuities at the corners
so we can use the averages

> X1 := <X, <0 | 0 | 0>>:
> X2 := <<200 | 200 | 200>, X1>:
> X3 := <<150, 100, 100, 100, 50> | X2>:
> X4 := <X3 | <175, 150, 150, 150, 75>>:
> X5 := SubMatrix(X4, [5,4,3,2,1], [1..5]);

      X5 :=
      [ 50    0    0    0    75 ]
      [ 100  67.857  66.518  85.714  150 ]
      [ 100  104.91  112.50  126.34  150 ]
      [ 100  139.29  152.23  157.14  150 ]
      [ 150   200    200    200   175 ]

> matrixplot(X5);

```



Example 21.8. Temperature surface

Similar Material in AEM: Sec. 21.4

EXAMPLE 21.9

HEAT EQUATION. CRANK–NICOLSON METHOD

The **one-dimensional heat equation** $u_t = c^2 u_{xx}$ is a **parabolic equation** that governs, for instance, the heat flow in a bar, where $u(x, t)$ is the temperature at a point x and time t . Solve the corresponding difference equation with $c^2 = 1$ on the interval $0 \leq x \leq 1$ (the bar extending from $x = 0$ to $x = L = 1$ along the x -axis) subject to the initial temperature $u(x, 0) = \sin \pi x$ by the **Crank–Nicolson method**. Use $n = 4$ grid points (5 intervals) giving an x -step $h = L/(n + 1) = 0.2$ and time step

$k = h^2 = 0.04$, doing 5 time steps from 0 to $T = 0.2$. (Note that $r = k/h^2$ has been taken as 1.)

Solution. The method proceeds by time steps. For each $t = 0, 0.04$, etc. you have 4 points of the grid at which the 4 values of the temperature are to be determined from a linear system of 4 equations in these unknown values. In this method the discretization of the heat equation is done in such a way that the resulting linear system $\mathbf{A}\mathbf{v} = \mathbf{b}$ has the following coefficient matrix (see the previous example for band matrices; type `?BandMatrix`), where $r = k/h^2$ ($= 1$ in the present case).

```
[ > with(LinearAlgebra):

[ > n := 4:  r := 1:  L := 1:  h := L/(n + 1):  k := r*h^2:  T := 0.2:
    N := T/k:

[ > A := BandMatrix([-r, 2 + 2*r, -r], 1, n);
                                A := 
$$\begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}$$

[
```

Obtain the initial temperature distribution and the preparation for a plot by typing

```
[ > u[1] := Transpose(<evalf(seq(sin(Pi*m*h), m = 0..(n + 1)))>);
u1 := [ 0.0  0.5877852524  0.9510565165  0.9510565165  0.5877852524  0.0 ]
```

Now obtain the values of the temperature $u(x, t)$ for $t = 0.04, 0.08, \dots, 0.20$ by iteration with respect to time, solving $\mathbf{A}\mathbf{v} = \mathbf{b}$ in each step. The use of `Vector[row]` creates the right type of vectors for the computations.

```
[ > for j from 1 to N do
    b := [seq(u[j][i - 1] + u[j][i + 1], i = 2..n + 1)];
    v := Vector[row](evalf[8](LinearSolve(A, Vector(b))));
    u[j + 1] := convert(<0 | v | 0>, Vector[row]);
end:

[ > for j from 1 to N + 1 do
    printf("t = %5.2f u = %10.8f\n", (j-1)*k, u[j]);
end;

t = 0.00 u = 0.00000000 0.58778525 0.95105652 0.95105652 0.58778525 0.00000000
t = 0.04 u = 0.00000000 0.39927376 0.64603851 0.64603851 0.39927376 0.00000000
t = 0.08 u = 0.00000000 0.27122071 0.43884433 0.43884433 0.27122071 0.00000000
t = 0.12 u = 0.00000000 0.18423618 0.29810041 0.29810041 0.18423618 0.00000000
t = 0.16 u = 0.00000000 0.12514889 0.20249516 0.20249516 0.12514889 0.00000000
t = 0.20 u = 0.00000000 0.08501177 0.13755194 0.13755194 0.08501177 0.00000000
```

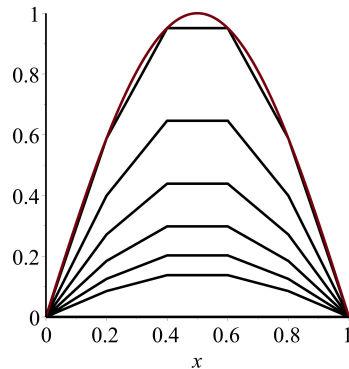
Obtain polygon plots of these values by typing

```
[ > with(plots):
```

```

> PS := plot(sin(Pi*x), x = 0..1):
for j from 1 to N + 1 do
  poly[j] := [seq([(i - 1)*h, u[j][i]], i = 1..n + 2)];
  P[j] := polygonplot(poly[j], scaling = constrained, style = line);
end:
display(PS, P[1], P[2], P[3], P[4], P[5], P[6]);

```



Example 21.9. Temperatures for time $t = 0, 0.04, 0.08, \dots$

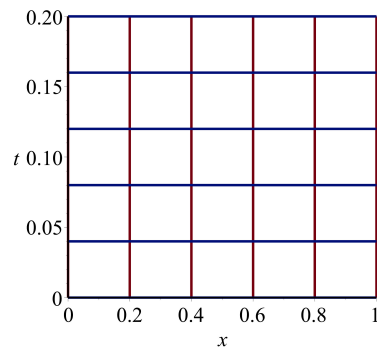
Each polygon corresponds to one of the times t_j considered, and is obtained by joining the temperatures $v(x, t_j)$ just computed for $x = 0.2, 0.4, 0.6, 0.8$

To obtain the grid, type the following, where [6, 6] gives the numbers of horizontal and vertical lines of the grid. Type ?conformal.

```

> with(plots):
> conformal(z, z = 0..1 + 0.2*I, grid = [6, 6], labels = [x, t]);

```



Example 21.9. Grid for a parabolic equation (the heat equation with $c^2 = 1$)

Problem Set for Chapter 21

Pr.21.1 (Euler's method) Solve $y' + 0.1y = 0$, $y(0) = 2$, doing 10 steps of size 0.1. Find the error. Plot the solution curve and the approximate values obtained. (*AEM* Sec. 21.1)

Pr.21.2 (Euler's method, Riccati equation) Solve $y' = (x + y)^2$, $y(0) = 0$, 10 steps of size 0.1. Find the exact solution and determine the error. (*AEM* Sec. 21.1)

- Pr.21.3 (Experiment on stepsize for Euler's method)** In Pr.21.1 find out experimentally for what stepsize the error of $y(1)$ will decrease to 0.0001 (approximately). (*AEM* Sec. 21.1)
- Pr.21.4 (Improved Euler method.)** Solve the equation $y' = y - xy^3$ subject to the initial condition $y(0) = 0.5$. Do 10 steps of size 0.1. Find the exact solution and the error. Plot the exact solution. (*AEM* Sec. 21.1)
- Pr.21.5 (Experiment on improved Euler method. Unbounded solution)** The ODE $y' = f(x, y) = 1 + y^2$ with initial value $y(0) = 0$ has solution $y = \tan x$. Use the improved Euler method to find the solution (and error) for x -values near $\pi/2$ (where the solution becomes infinite). Experiment with $h = 0.1, 0.05, 0.025$ and study the increase of the error with x for constant h and the decrease of the error under halving h for constant x . (*AEM* Sec. 21.1)
- Pr.21.6 (Classical Runge-Kutta method RK)** Solve $y' = (y - x - 1)^2 + 2$, $y(0) = 1$ by RK. Do 10 steps of size 0.1. Solve the problem exactly by setting $y - x - 1 = u$ and separating variables, or by [dsolve](#). Calculate the error. (*AEM* Sec. 21.1)
- Pr.21.7 (RK, step halving)** Solve $y' = -0.2xy$, $y(0) = 1$ by RK. Do 50 steps of size 0.2. Do 100 steps of size 0.1. Solve the problem exactly. Compare the errors of the two values for $x = 10$. (*AEM* Sec. 21.1)
- Pr.21.8 (Adams-Moulton method)** Solve the initial value problem $y' = x^2/y$, $y(1) = 3$, by the procedure [AdamsMoultonRK](#). Do 10 steps of size 0.4. Find the exact solution. Determine the errors. (*AEM* Sec. 21.2)
- Pr.21.9 (Multistep method)** Using the procedure in Example 21.5 in this Guide, solve $y' = x^2 + y$, $y(0) = 0$. Do 10 steps of size 0.1. (*AEM* Sec. 21.2)
- Pr.21.10 (Mass-spring system)** Solve $y'' + 2y' + 0.75y = 0$, $y(0) = 3$, $y'(0) = -2.5$ by RKS (Runge-Kutta for systems). Do 5 steps of size 0.2. Find the exact solution and compute the error. (*AEM* Sec. 21.3)
- Pr.21.11 (RKS Runge-Kutta for systems)** Solve $y_1' = 2y_1 - 4y_2$, $y_2' = y_1 - 3y_2$, $y_1(0) = 3$, $y_2(0) = 0$ by RKS for x from 0 to 5 with $h = 0.1$. Find the exact solution and the errors of y_1 and y_2 . Plot y_1 and y_2 . (*AEM* Sec. 21.3)
- Pr.21.12 (Runge-Kutta-Nystroem method)** Solve the initial value problem in Pr.21.10 by RKN (see Example 21.7 in this Guide) and compare the accuracy with that in Pr.21.10. (*AEM* Sec. 21.3)
- Pr.21.13 (Laplace equation. Dirichlet problem)** Solve the Laplace equation numerically for the refined grid in Example 21.8 in this Guide and boundary values 0 on the bottom, 220 V on the right edge, 110 V on the top, 220 V on the left edge. (*AEM* Sec. 21.4)
- Pr.21.14 (Crank-Nicolson. "Trapezoidal" initial temperature)** Solve the heat equation in Example 21.9 in this Guide when the initial temperature is $4x$ for $0 \leq x \leq 1/4$, 4 for $1/4 \leq x \leq 3/4$, and $4 - 4x$ for $3/4 \leq x \leq 1$, using the same grid and r and h as in that example. (*AEM* Sec. 21.6)

Pr.21.15 (Crank–Nicolson method, grid refinement) In Example 21.9 in this Guide choose $n = 24$ (instead of $n = 4$), thus $h = 1/25 = 0.04$, $r = k/h^2 = 1$ and $k = 1/625 = 0.0016$. Do enough steps so that you obtain values corresponding to those given in Example 21.9 and compare. (*AEM* Sec. 21.6)