

20

Conventional Newton Methods

TABLE OF CONTENTS

	Page
§20.1. Introduction	20-3
§20.2. Stage Analysis Review	20-3
§20.3. Problem Statement	20-3
§20.4. The Corrective Phase	20-4
§20.5. Solving the Newton Systems	20-5
§20.6. Termination Tests	20-5
§20.7. The Ordinary Newton Method	20-6
§20. Exercises	20-10
§20. Solutions to Exercises.	20-11

§20.1. Introduction

In the overview of solution methods given in Chapter 16 it was noted that solution methods based on continuation generally included two phases: incremental and corrective. In the purely incremental methods covered in Chapter 17–19 the corrective phase is absent. If the corrective phase is present, the incremental formula simply functions as a *predictor* that provides a starting point for the corrective iteration. The purpose of this iteration is to eliminate (or at least reduce) the drifting error by moving towards the equilibrium path along the constraint hypersurface.

Solution methods that include a corrective phase will be collectively called *corrective methods*, although perhaps a more appropriate name would be predictor-corrector methods. There are *purely corrective methods* that lack a predictor phase entirely (for example, the orthogonal trajectory accession method) but they have not proven important in practical applications.

The most important class of corrective methods pertains to the Newton-Raphson method and its numerous variants: modified, modified-delayed, damped, quasi, and so forth. These are collectively called *Newton-like methods*, and only require *access to the past solution*. In the present section we study the conventional Newton method under general increment control.

§20.2. Stage Analysis Review

Let us recall that our purpose is to solve the residual equations

$$\mathbf{r}(\mathbf{u}, \lambda) = \mathbf{0} \quad (20.1)$$

over a loading stage as the control parameter is incremented from 0. As previously discussed the additional equation that makes (20.1) determinate is the increment constraint equation

$$c(\mathbf{u}, \lambda) = 0. \quad (20.2)$$

Starting from $\lambda = 0$, we want to calculate a series of solutions

$$\mathbf{u}_0, \lambda_0, \quad \mathbf{u}_1, \lambda_1, \quad \dots \quad \mathbf{u}_n, \lambda_n \quad \dots \quad (20.3)$$

that characterizes numerically the response $\mathbf{u} = \mathbf{u}(\lambda)$ while satisfying the residual equations (20.1) within prescribed accuracy.

The purely incremental methods covered in the three previous Chapters compute a sequence of values such as (20.3) by direct integration of the first-order rate equations $\mathbf{K}\mathbf{u}' = \mathbf{q}$ or $\mathbf{K}_{ref}\mathbf{u}' = \mathbf{q} - \mathbf{f}'$. The methods considered here implement a *corrective phase* in which one iterates for equilibrium while satisfying the increment constraint. The starting point for the corrective phase is the solution *predicted* by the incremental method. Consequently, these methods are often called *incremental-iterative* methods.

§20.3. Problem Statement

Assume that n incremental steps of the stage analysis have been performed. The last accepted solution is

$$\mathbf{u}_n, \quad \lambda_n \quad (20.4)$$

We want to compute the solution

$$\mathbf{u}_{n+1}, \quad \lambda_{n+1} \quad (20.5)$$

that satisfies the nonlinear algebraic system

$$\begin{aligned} \mathbf{r}(\mathbf{u}_{n+1}, \lambda_{n+1}) &= \mathbf{0}, \\ c(\Delta \mathbf{u}_n, \Delta \lambda_n) &= 0, \end{aligned} \quad (20.6)$$

where

$$\Delta \mathbf{u}_n = \mathbf{u}_{n+1} - \mathbf{u}_n, \quad \Delta \lambda_n = \lambda_{n+1} - \lambda_n \quad (20.7)$$

Although the above increment constraint is a special case of (20.2), it befits those most commonly used in practical calculations.

The *predicted solution*

$$\mathbf{u}_n^0, \quad \lambda_n^0, \quad (20.8)$$

is typically obtained by performing an incremental step as described in Chapters 17 and 18.

§20.4. The Corrective Phase

All that computations that follow pertain to the n^{th} incremental step. Hence for simplicity we shall omit the subscript n from the formulas.

Starting from the predicted approximation (20.8),

$$\mathbf{u}^0 \equiv \mathbf{u}_n^0, \quad \lambda^0 \equiv \lambda_n^0 \quad (20.9)$$

the conventional Newton method applied to (20.6) generates a sequence of iterates

$$\mathbf{u}^k, \quad \lambda^k, \quad (20.10)$$

where $k = 1, 2 \dots$ is an iteration step index.

The conventional Newton method is based on the truncated Taylor expansion of the system $\mathbf{r} = \mathbf{0}$, $c = 0$ about $(\mathbf{u}^k, \lambda^k)$:

$$\begin{aligned} \mathbf{r}^{k+1} &= \mathbf{r}^k + \frac{\partial \mathbf{r}}{\partial \mathbf{u}}(\mathbf{u}^{k+1} - \mathbf{u}^k) + \frac{\partial \mathbf{r}}{\partial \lambda}(\lambda^{k+1} - \lambda^k) + H.O. = \mathbf{0}, \\ c^{k+1} &= c^k + \frac{\partial c}{\partial \mathbf{u}}(\mathbf{u}^{k+1} - \mathbf{u}^k) + \frac{\partial c}{\partial \lambda}(\lambda^{k+1} - \lambda^k) + H.O. = 0. \end{aligned} \quad (20.11)$$

where ‘H.O.’ denote higher order terms that are quadratic or higher in the changes $\mathbf{u}^{k+1} - \mathbf{u}^k$ and $\lambda^{k+1} - \lambda^k$, and all derivatives are evaluated at $(\mathbf{u}^k, \lambda^k)$. Discarding such terms and recalling that $\mathbf{K} = \partial \mathbf{r} / \partial \mathbf{u}$, $\mathbf{q} = -\partial \mathbf{r} / \partial \lambda$, $\mathbf{a}^T = \partial c / \partial \mathbf{u}$, $g = \partial c / \lambda$, we obtain for the corrections

$$\mathbf{d} = \mathbf{u}^{k+1} - \mathbf{u}^k, \quad \eta = \lambda^{k+1} - \lambda^k \quad (20.12)$$

the linear algebraic system

$$\begin{bmatrix} \mathbf{K} & -\mathbf{q} \\ \mathbf{a}^T & g \end{bmatrix} \begin{bmatrix} \mathbf{d} \\ \eta \end{bmatrix} = - \begin{bmatrix} \mathbf{r} \\ c \end{bmatrix}, \quad (20.13)$$

where

$$\mathbf{K} = \frac{\partial \mathbf{r}}{\partial \mathbf{u}}, \quad \mathbf{q} = -\frac{\partial \mathbf{r}}{\partial \lambda}, \quad \mathbf{a}^T = \frac{\partial c}{\partial \mathbf{u}}, \quad g = \frac{\partial c}{\partial \lambda}, \quad (20.14)$$

and all known quantities are evaluated at \mathbf{u}^k, λ^k . Note that for notational simplicity this superscript has been kept out of $\mathbf{d}, \eta, \mathbf{r}$, etc, unless it is desirable to make the dependency on the iteration index k explicit. If the tangent stiffness matrix \mathbf{K} is of order N , the coefficient matrix of the linear system (20.13) has order $N + 1$. This matrix is called the *augmented stiffness* matrix.

Note that although generally \mathbf{K} is symmetric and sparse, the augmented stiffness is generally unsymmetric (but see Exercise 20.2), and its sparseness is detrimentally affected by the augmentation. It is therefore of interest to treat the linear system (20.13) with techniques that preserve those attributes. The solution procedures described below make use of *auxiliary systems of equations* to achieve that goal. The number of auxiliary systems depends on whether the tangent stiffness \mathbf{K} is nonsingular (regular points) or singular (critical points). For the latter we have to distinguish between limit points and bifurcation points. In the present section we shall concentrate on the treatment of regular points.

§20.5. Solving the Newton Systems

Recall from Chapter 4 that regular points of the system (20.1) are equilibrium solutions (\mathbf{u}, λ) at which the tangent stiffness matrix \mathbf{K} is nonsingular. If this property holds, we can perform forward Gauss elimination on (20.13) to get rid of \mathbf{d} and produce the following scalar equation for η :

$$(g + \mathbf{a}^T \mathbf{K}^{-1} \mathbf{q}) \eta = -c + \mathbf{a}^T \mathbf{K}^{-1} \mathbf{r}. \quad (20.15)$$

Let \mathbf{d}_r and \mathbf{d}_q denote the solution of the *symmetric* linear systems

$$\mathbf{K} \mathbf{d}_r = -\mathbf{r}, \quad \mathbf{K} \mathbf{d}_q = \mathbf{q}. \quad (20.16)$$

Then

$$\eta = -\frac{c + \mathbf{a}^T \mathbf{d}_r}{g + \mathbf{a}^T \mathbf{d}_q}, \quad \mathbf{d} = \mathbf{d}_r + \eta \mathbf{d}_q. \quad (20.17)$$

It is seen that two right hand sides, \mathbf{r} and \mathbf{q} , have to be generally solved for at each Newton step. The number reduces to one for $k > 1$, however, if *modified Newton* is used so that \mathbf{K} is held fixed for several steps *and* \mathbf{q} does not vary. The last assumption holds in structural mechanics applications if the loading is conservative and proportional. (The modified Newton method is described in the next Chapter).

§20.6. Termination Tests

At which point should we stop the Newton iteration? There are several convergence criteria that can be applied.

1. *Displacement convergence test.* The change in the last correction \mathbf{d} of the state vector \mathbf{u} , as measured in an appropriate norm, should not exceed a given tolerance ϵ_d . For example, using the 2-norm (Euclidean norm)

$$\|\mathbf{d}\| = \sqrt{\mathbf{d}^T \mathbf{d}} \leq \epsilon_d. \quad (20.18)$$

2. *Residual convergence test.* Since the residual \mathbf{r} measures the departure from equilibrium, another appropriate convergence test is

$$\|\mathbf{r}\| \leq \epsilon_r. \quad (20.19)$$

Some comments are now in order.

1. The two tests may be applied in an “and” or “or” matter as iteration stopping criterion. It is also possible to combine both tests in the form of an “work change” criterion, for example

$$|\mathbf{r}^T \mathbf{d}| \leq \epsilon_d \epsilon_r \quad (20.20)$$

2. Since \mathbf{d} and \mathbf{r} have usually physical dimensions, so do necessarily ϵ_d and ϵ_r . For a general purpose implementation of Newton iteration this dependency on physical units is undesirable and it is more convenient to work out with ratios that render the ϵ_r and ϵ_d dimensionless. For example:

$$\frac{\|\mathbf{r}\|}{\|\mathbf{r}^0\|} \leq \epsilon_r \quad (20.21)$$

where \mathbf{r}^0 is the residual after the predictor step; now ϵ_r can be dimensionless. A similar ratio can be used for the displacement convergence test, but here the reference value should be a *total* or accumulated displacement; for example:

$$\frac{\|\mathbf{d}\|}{\|\mathbf{u}^0\|} \leq \epsilon_d \quad (20.22)$$

3. *Divergence Safeguards.* The Newton iteration is not guaranteed to converge. There should therefore be *divergence detection tests* that will cause the iteration to be interrupted. For example, turning the above ratios around, divergence may be diagnosed if either of the following inequalities occur:

$$\frac{\|\mathbf{r}\|}{\|\mathbf{r}^0\|} \geq g_r, \quad \frac{\|\mathbf{d}\|}{\|\mathbf{u}^0\|} \geq g_d \quad (20.23)$$

where g_r and g_d are “dangerous growth” factors, for example $g_r = g_d = 1000$.

Occasionally the Newton iteration will neither diverge not converge but just “bounce around” (oscillatory behavior). To avoid excessive wheel spinning in such cases it is always a good practice to put a maximum number of iterations per step in the program. Typical limits might be 20 to 50.

§20.7. The Ordinary Newton Method

The Newton iteration discussed in the mathematical literature on solving nonlinear systems assume that λ is held constant. This corresponds to λ control or load control in our terminology. As previously explained, fixing λ makes critical points impassable. However, the resultant method provides good examples to watch the typical behavior of the Newton iteration process.

If λ is kept constant the incremental step constraint is $\Delta\lambda_n = \ell_n$, which has derivatives $\mathbf{0}$ and 1 with respect to \mathbf{u} and λ , respectively. System (20.13) simplifies to

$$\begin{bmatrix} \mathbf{K} & -\mathbf{q} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{d} \\ \eta \end{bmatrix} = - \begin{bmatrix} \mathbf{r} \\ 0 \end{bmatrix}, \quad (20.24)$$

where $c = 0$ because the constraint is satisfied exactly. Since $\eta = 0$, the “bordering” disappears and the Newton iteration reduces to

$$\mathbf{u}^{k+1} = \mathbf{u}^k - (\mathbf{K}^k)^{-1} \mathbf{r}^k, \quad \lambda^{k+1} = \lambda^k = \lambda_n + \ell_n \quad (\text{kept fixed}) \quad (20.25)$$

Note that the incremental load vector \mathbf{q} disappeared entirely. This is the method found in standard numerical analysis texts. This version is used in the examples that follow.

Example 20.1. The computation of the square root $+\sqrt{a}$ of a scalar number $a > 0$ by Newton iteration is set up as follows. The square root satisfies the equation $r(x) = f(x) = x^2 - a = 0$. Starting from an initial value $x_0 > 0$, the Newton iteration computes

$$x^{k+1} = x^k - f(x^k)/f'(x^k) = x^k - ((x^k)^2 - a)/(2x^k). \quad (20.26)$$

where prime denotes derivative with respect to x . The results for $a = 3$ and $x_0 = 1$ are illustrated by the Mathematica program below.

```
f[x_, a_] := x^2 - a; Df[x_] := 2*x;
a=3.; xk=1.; Print["x0=", xk];
For[k=0, k<6, k++, xkp1=xk-f[xk, a]/Df[xk];
  Print["x", k+1, "=", xkp1//InputForm]; xk=xkp1];

x0=1.
x1=2.
x2=1.75
x3=1.732142857142857143
x4=1.732050810014727541
x5=1.732050807568877295
x6=1.732050807568877294
```

After six cycles the iteration yields 16 places of accuracy for $\sqrt{3}$. Note that the number of exact digits roughly doubles from $k = 2$ onwards. This is typical of the Newton iteration once it “locks in” a root because the process has asymptotically quadratic convergence.

The numerical process shown above is actually that used by mathematical software libraries of languages like Fortran or C for the computation of the square root function. However the initial value is determined by a scaled rational interpolant that gives 2-4 digits of accuracy for x_0 ; as a result only 2 or 3 cycles are needed to achieve double-precision accuracy for most inputs.

Example 20.2. This example is more typical of a structural application. Consider the residual equation for a 2-DOF system

$$\mathbf{r}(\mathbf{u}, \lambda) = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} u_1 + 2u_1^3 - u_2^2 - 2\lambda \\ 3u_2 - 2u_1u_2 - \lambda \end{bmatrix} \quad (20.27)$$

The tangent stiffness matrix is

$$\mathbf{K} = \frac{\partial \mathbf{r}}{\partial \mathbf{u}} = \begin{bmatrix} 1 + 6u_1^2 & -2u_2 \\ -2u_2 & 3 - 2u_1 \end{bmatrix} \quad (20.28)$$

For $\lambda = 1$ the residual equations have three real roots which to six digits of accuracy are

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} 1.18400 \\ 1.58227 \end{bmatrix}, \quad \mathbf{u}_3 = \begin{bmatrix} 1.66726 \\ -2.98938 \end{bmatrix}. \quad (20.29)$$

The following Mathematica programs starts from the initial values $u_1^0 = u_2^0 = 0.8$ and quickly finds the nearest root $u_1 = u_2 = 1$, delivering 16 digits of accuracy after 5 cycles:

```
ClearAll[r,u1,u2,lambda];
lambda=1;
r[u1_,u2_] := {{u1+2*u1^3-u2^2-2*lambda},{3*u2-2*u1*u2-lambda}};
Kt[u1_,u2_] := {{1+6*u1^2,-2*u2},{-2*u2,3-2*u1}}
uk={{0.8},{0.8}};
Print["Starting v0=",uk//InputForm];
For [k=0, k<5, k++,
  {u1,u2}={uk[[1,1]],uk[[2,1]]};
  ukp1 = uk - Inverse[Kt[u1,u2]].r[u1,u2];
  Print["Cycle k= ",k," u",k+1,"=",ukp1//InputForm];
  uk=ukp1 ];
{u1,u2}={uk[[1,1]],uk[[2,1]]};
Print["Final residual=",r[u1,u2]];

Starting v0={{0.8}, {0.8}}
Cycle k= 0 u1={{1.025426944971537002}, {0.9719165085388994309}}
Cycle k= 1 u2={{1.001827210881738689}, {1.005246766090385063}}
Cycle k= 2 u3={{0.999984431106495672}, {0.9999493398104184104}}
Cycle k= 3 u4={{0.999999985779087286}, {0.9999999955786530219}}
Cycle k= 4 u5={{0.999999999999999893}, {0.999999999999999659}}
-18 -17
Final residual={{-7.48099 10 }, {-1.25767 10 }}}
```

But changing the initial values to $u_0 = 0.8$ and $u_1 = 1.1$, which is even closer to the $u_1 = u_2 = 1$ root, the process converges to the second root in (20.29), reaching 16 digits of accuracy after 8 cycles:

```
ClearAll[r,u1,u2,lambda];
lambda=1;
r[u1_,u2_] := {{u1+2*u1^3-u2^2-2*lambda},{3*u2-2*u1*u2-lambda}};
Kt[u1_,u2_] := {{1+6*u1^2,-2*u2},{-2*u2,3-2*u1}}
uk={{0.8},{1.1}};
Print["Starting u0=",uk//InputForm];
For [k=0, k<5, k++,
```

```

{u1,u2}={uk[[1,1]],uk[[2,1]]};
ukp1 = uk - Inverse[Kt[u1,u2]].r[u1,u2];
Print["Cycle k= ",k," u",k+1,"=",ukp1//InputForm];
uk=ukp1 ];
{u1,u2}={uk[[1,1]],uk[[2,1]]};
Print["Final residual=",r[u1,u2]];

Starting u0={{0.8}, {1.1}}
Cycle k= 0 u1={{1.1886363636363636}, {1.325}}
Cycle k= 1 u2={{1.215044904251438747}, {1.718220285975100653}}
Cycle k= 2 u3={{1.189744133901914791}, {1.602104015495415557}}
Cycle k= 3 u4={{1.184187697535010493}, {1.582880662857412937}}
Cycle k= 4 u5={{1.183998614947570215}, {1.582271181104649434}}
Cycle k= 5 u6={{1.183998417328768284}, {1.582270556284133866}}
Cycle k= 6 u7={{1.183998417328558547}, {1.582270556283474425}}
Cycle k= 7 u8={{1.183998417328558548}, {1.582270556283474426}}
-19
Final residual={{0.}, {-8.67362 10  }}

```

This illustrates the “finicky” nature of Newton iteration. It can do (and often does) the unexpected, such as diverging or converging to the “wrong” root. In fact the a whole subset of fractal or chaotic mathematics is devoted to the understanding of “domains of attraction” of roots. Because of this capricious behavior, in practical use of the Newton corrector numerous safeguards are implemented to avoid surprises. But the whole subject is too lengthy for coverage in an introductory treatment.

Homework Exercises for Chapter 20
Conventional Newton Methods

EXERCISE 20.1 If the case of load control, the incremental stepsize constraint reduces to

$$c = \Delta\lambda_n - \ell = 0 \quad (\text{E20.1})$$

where ℓ is prescribed. After the predictor step, $c = 0$. Show that the Newton method reduces to a standard form that requires only the solution of one of the auxiliary systems (20.16).

EXERCISE 20.2 What algebraic constraint $c = 0$ would make the augmented tangent stiffness symmetric?

EXERCISE 20.3 Enlarge the program that solves Exercise 18.2 by using conventional Newton as corrector, the incremental stepsize constraint

$$c = |\Delta s_n| - \ell_n = 0, \quad (\text{E20.2})$$

(where $\ell = \ell_n$ is kept fixed throughout) and the termination tolerance $|\mathbf{r}^k / \mathbf{r}^0| \leq 10^{-3}$ or 20 iterations, whichever occurs first. Suggested values for ℓ is 0.01, but the results should be insensitive to that choice.

EXERCISE 20.4 (Advanced) Obtain the expression of the augmented stiffness if the orthogonal trajectory accession method is used and write out the Newton algorithm for this case.

Homework Assignments for Chapter 20
Solutions

EXERCISE 20.1 The constraint $c = \Delta\lambda_n - \ell = 0$ does not depend on \mathbf{u} , thus

$$g = \frac{\partial c}{\partial \lambda} = 1, \quad \mathbf{a}^T = \frac{\partial c}{\partial \mathbf{u}} = \mathbf{0} \quad (\text{E20.3})$$

The Newton correction system becomes, with $c = 0$,

$$\begin{bmatrix} \mathbf{K} & -\mathbf{q} \\ \mathbf{a}^T & g \end{bmatrix} \begin{bmatrix} \mathbf{d} \\ \eta \end{bmatrix} = \begin{bmatrix} \mathbf{K} & -\mathbf{q} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{d} \\ \eta \end{bmatrix} = \begin{bmatrix} -\mathbf{r} \\ 0 \end{bmatrix} \quad (\text{E20.4})$$

The second equation gives $\eta = 0$ (meaning there is no λ correction) and the first one becomes

$$\mathbf{Kd} = -\mathbf{r} \quad (\text{E20.5})$$

which is the conventional Newton method.

EXERCISE 20.2 The augmented stiffness is symmetric if

$$\mathbf{a}^T = \frac{\partial c}{\partial \mathbf{u}} = -\mathbf{q} \quad (\text{E20.6})$$

where the incremental load vector \mathbf{q} is defined by the statement of the problem. If \mathbf{q} does not depend on \mathbf{u} the simplest choice that satisfies the differential relation $\partial c / \partial \mathbf{u} = -\mathbf{q}$ is

$$c = -(\mathbf{q}^T \Delta \mathbf{u} - \ell_n) = 0 \quad (\text{E20.7})$$

where ℓ_n is a prescribed stepsize. Note that $\Delta \mathbf{u} = \mathbf{u} - \mathbf{u}_n$ is used so that setting $\ell_n = 0$ gives the past solution $\mathbf{u} = \mathbf{u}_n$. Physically this is an external work constraint because $\mathbf{q}^T \mathbf{u}$ has meaning of external energy.

If \mathbf{q} depends on \mathbf{u} , $c(\mathbf{u}, \lambda) = 0$ cannot be generally expressed in closed form. It is always possible, however, to set $\mathbf{a}^T = \mathbf{q}$ directly.

Cell 20.1 The Incremental Iterative Step Driver

```

IncIterSolution[method_,sprop_,force_,sol_,solpar_]:=
Module[ {integ,ics,n=sol[[1]],lambda=sol[[2]],u=sol[[3]],
  nmax=solpar[[1]],lambdamax=solpar[[2]],umax=solpar[[3]],
  solpred,solnext,status,sol0},
If [n>nmax, Return[{Null,
  "IncSolution: Max steps reached"}]];
If [Abs[lambda]>Abs[lambdamax], Return[{Null,
  "IncSolution: lambda exceeds bound"}]];
If [Abs[u[[1]]]>Abs[umax[[1]]], Return[{Null,
  "IncSolution: uX exceeds bound"}]];
If [Abs[u[[2]]]>Abs[umax[[2]]], Return[{Null,
  "IncSolution: uY exceeds bound"}]];
{integ,ics}=method; sol0=sol;
If [integ=="FE",
  {solpred,status}=FEstep [ics,sprop,force,sol,solpar]];
If [integ=="MR",
  {solpred,status}=MRstep [ics,sprop,force,sol,solpar]];
If [integ=="RK4",
  {solpred,status}=RK4step[ics,sprop,force,sol,solpar]];

{solnext,status}=NewtonIteration[ics,sprop,force,solpred,
  solpar,sol0];
Return[{solnext,status}]];

```

EXERCISE 20.3 This exercise is solved with the Mathematica modules listed in Cells 20.1 through 20.4, which collectively augment the Notebook described in Chapter 18. The augmented Notebook, called `IncIterSolTwoBarArch`, is posted on the Web.

The incremental-iterative solution over one increment is done by module `IncIterSolution`, which is listed in Cell 20.1. The logic is quite similar to `IncSolution` until one gets near the end. The module performs the same termination tests as `IncSolution`, saves the previous solution in `sol0` and then does the prediction step with `FEStep`, `MRStep` or `RK4Step`, according to the integration method specified (although in practice, the only predictor that makes sense when followed by a corrective phase is Forward Euler). Then it calls `NewtonIteration` to execute the corrective process.

The `NewtonIteration` module is listed in Cell 20.2. This module executes the Conventional Newton process by cycling over $k = 1, 2, \dots$ up to a maximum number of iterations `mNiter`. Within the loop it calls `LCNewton`, `DCNewton` or `ACNewton` according to the increment control strategy specified. On an error free return it performs convergence and divergence tests using the norm of the residual vector.

Modules `LCNewton`, `DCNewton` and `ACNewton` are listed in Cell 20.3. The only difference in the logic is in the handling of different increment control constraints. All three modules begin by calling `IncVelocity` and `ResVelocity` to solve $\mathbf{K}\mathbf{v} = \mathbf{q}$ (\mathbf{v} and \mathbf{d}_q are the same vector) and $\mathbf{K}\mathbf{d}_r = -\mathbf{r}$, respectively. Module `IncVelocity` is the same as in the Chapter 18 Notebook whereas `ResVelocity` is a new module listed in Cell 20.4.

`LCNewton` simply adds \mathbf{d}_r to \mathbf{u}^k and exits since under load control λ does not change. The other two modules: `DCNewton` and `ACNewton` compute the constraint violation c^k , obtain the variation η of the control parameter,

Cell 20.2 The Conventional Newton Iteration Driver

```

NewtonIteration[ics_, sprop_, force_, sol_, solpar_, sol0_] := Module[
  {n, lambdan, un, vn, qn, Kdetn, elln, an, kappan, kappa0, remn,
   rem=" ", state, mNiter, epsconv, facdiv, r0, r0norm, rk, rknorm,
   statenext, converged, status, lambda0, u0, v0},
  {n, lambdan, un, vn, qn, Kdetn, elln, an, kappan, kappa0, remn} = sol;
  mNiter = 2; epsconv = 0.00001; facdiv = 10000.;
  p = FormInternalForceOfTwoBarArch[sprop, un]; p = N[p];
  rn = p - lambdan * force; rn = N[rn];
  rnnorm = Sqrt[rn.rn];
  state = {lambdan, un, rn}; lambda0 = sol0[[2]]; u0 = sol0[[3]]; v0 = sol0[[4]];
  If [rnnorm == 0, Return[sol, status]]; converged = False;
  For [k = 0, k <= mNiter, k++,
    If [ics == "LC",
      {statenext, status} = LCNewtonCycle[sprop, force, state]];
    If [ics == "DC",
      {statenext, status} = DCNewtonCycle[sprop, force, state,
        {lambda0, u0, elln}]];
    If [ics == "AC",
      {statenext, status} = ACNewtonCycle[sprop, force, state,
        {lambda0, u0, v0, elln}]];
    If [status != " ", Break[]];
    {lambda, u, r} = statenext; rnorm = Sqrt[r.r];
    If [rnorm/rnnorm < epsconv, converged = True; Break[]];
    If [rnorm/rnnorm > facdiv, status = "Newton iteration diverges";
      Return[Null, status]];
    state = statenext;
  ];
  If [!converged, status = "Newton didnt converge in max iters"];
  If [converged, Print["Newton converged after ", k, " cycles"]];
  {lambda, u, r} = state;
  solnext = {n, lambda, u, vn, qn, Kdetn, elln, an, kappan, kappa0, remn};
  Return[{solnext, status}]];

```

combines the two solutions \mathbf{d}_q and \mathbf{d}_r to form \mathbf{d}^k , correct \mathbf{u} and λ by \mathbf{d} and η , respectively, and exit.

Cell 20.3 The Newton Iteration Cycle Modules

```

LCNewtonCycle[sprop_,force_,{lambdak_,uk_,rk_}]:=Module[
  {v,q,status,dr,r,lambda,u},
  {v,q,status}= IncVelocity[sprop,force,{lambdak,uk}];
  {dr,r,status}=ResVelocity[sprop,force,{lambdak,uk}];
  If [status!=" ",Return[Null,status]];
  lambda=lambdak; u=uk+dr;
  r=FormInternalForceOfTwoBarArch[sprop,u]-lambda*force; r=N[r];
  Return[{{lambda,u,r}," "}}];

DCNewtonCycle[sprop_,force_,{lambdak_,uk_,rk_},
  {lambda0_,u0_,v0_,ell_}]:=Module[ {v,q,status,dr,r,lambda,u,
  eta,ck},
  {v,q,status}= IncVelocity[sprop,force,{lambdak,uk}];
  {dr,r,status}=ResVelocity[sprop,force,{lambdak,uk}];
  If [status!=" ",Return[Null,status]];
  ck=Sqrt[(uk-u0).(uk-u0)]*Sign[q.v0]-ell;
  eta=-(ck+(uk-u0).dr)/((uk-u0).v);
  lambda=lambdak+eta; u=uk+dr+eta*v;
  r=FormInternalForceOfTwoBarArch[sprop,u]-lambda*force; r=N[r];
  Return[{{lambda,u,r}," "}}];

ACNewtonCycle[sprop_,force_,{lambdak_,uk_,rk_},
  {lambda0_,u0_,v0_,ell_}]:=Module[ {v,q,status,dr,r,lambda,u,
  eta,ck,f0},
  {v,q,status}= IncVelocity[sprop,force,{lambdak,uk}];
  {dr,r,status}=ResVelocity[sprop,force,{lambdak,uk}];
  If [status!=" ",Return[Null,status]];
  f0=Sqrt[1+v0.v0]*Sign[q.v0];
  ck=(v0.(uk-u0)+(lambdak-lambda0))/f0-ell;
  eta=-(ck+(v0/f0).dr)/((1+v0.v)/f0);
  lambda=lambdak+eta; u=uk+dr+eta*v;
  r=FormInternalForceOfTwoBarArch[sprop,u]-lambda*force; r=N[r];
  Return[{{lambda,u,r}," "}}];

```

Cell 20.4 The Residual Velocity Computation Module

```

ResVelocity[sprop_,force_,{lambda_,u_}] :=
Module[{K,r,p,nudge,un=u,dr,OK,ueps=10.^(-10)},
nudge=0; SeedRandom[7654321];
(*Print["ResVelocity, lambda=",lambda," un=",un];*)
While [nudge<=10,
K=FormTanStiffnessOfTwoBarArch[sprop,un]; K=N[K];
p=FormInternalForceOfTwoBarArch[sprop,un]; p=N[p];
r=p-lambda*force; r=N[r];
{dr,OK}=SolveTwoLinearEqs[K,-r];
If [OK, Break[]];
nudge++; un+=Table[Random[]*ueps,{2} ]];
If [OK, Return [{N[dr],N[r]," "}]];
Return [Null,Null,"ResVelocity: Cant escape singularity"]];

```