Contents

1 Root Finding

Solutions to the equation $f(x) = 0$ in one dimension.

1.1 Bisection Method

Given an interval $[a, b]$ containing a root, the interval is interatively halved. The interval contains a root if $f(a)f(b) < 0$.

The Bisection method always converges, so long as a root exists on the inteval given. The bisection method converges linearly, making it one of the slower options.

```
function [p] = bisection(f, a, b, iterations)

    fa = f(a);
    p = a;
    fp = fa;

    for i=1:iterations

        % set p to center of the interval
        p = (a + b) / 2.0;
        fp = f(p);

        % test location of root in the interval
        if fp*fa > 0
            a = p;
            fa = fp;
        else
            b = p;
        end

        fprintf('Iteration %3.0d: p = %4.9f, f(p) = %4.9f \n', i, p, f(p));

    end
end
```

1.2 Fixed Point Iteration

A fixed point is the solution to $f(p) = p$. One is not guaranteed to exist.

Given a starting point p, a function $g(x) = x - f(x)$ is constructed, and its series is iterated.

$$x_i = g(x_{i-1})$$

The solution for p exists where the solution converges. Fixed Point Iteration converges if g is continuous on its range, and its range contains a fixed point.

```
function [p] = fixed_point(f, p, iterations)
    for i=1:iterations

        p = p - f(p);

        fprintf('Iteration %3.0d: p = %4.9f, f(p) = %4.9f \n', i, p, f(p));

    end
```

```
end
```

1.3 Newton's Method

Given a starting point p, and the function f', its series is iterated.

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$$

Fixed Point iteration is then applied. Newton's Method converges quadratically, but requires the derivative of the function and the initial guess to be close to the root.

```
function [p] = newton(f, fp, p, iterations)

    for i=1:iterations

        p = p - f(p)/fp(p);

        fprintf('Iteration %3.0d: p = %4.9f, f(p) = %4.9f \n', i, p, f(p));

    end
end
```

1.4 Aitken's Δ^2 Method

The Δ^k represents the k-order finite-derivative and is defined such that $\Delta p_n = p_{n+1} - p_n$ and $\Delta^k p_n = \Delta(\Delta^{k-1} p_n)$.

Using the first- and second-order finite-derivates, a series can be constructed.

$$\hat{x}_n = x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n}$$

This series can then be applied with fixed point iteration. Aitken's Δ^2 Method generally converges much faster than the original series.

1.5 Horner's Method

Given that function P is a polynomial with n real roots, and polynomial Q with no real roots, function P can be factorized by Q.

$$P(x) = Q(x) \prod_{i=0}^{n} (x - x_i) + a_0$$

Horner's Method is rapid for polynomials and can find all zeros through recursion. Programmatically, the polynomial is represented as a coefficient array.

2 Interpolation and Polynomial Approximation

Equations which interpolate given a set of points.

2.1 Lagrange Polynomials

Given a set of points x, and the output of their unknown function f, an interpolating function P of order n can be constructed.

$$P(x) = \sum_{k=0}^{n} \Big[f(x_k) \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \Big]$$

Lagrange Polynomials are effective for the general case, but prone to round-off error.

```
function [P] = lagrange(x, y, p)

    n = length(y);
    P = zeros(1, length(p));

    for k=1:n
        L = ones(1, length(p));
        for j=[1:k-1 k+1:n]
            L = L .* ((p - x(j)) ./ (x(k) - x(j)));
        end
        P = P + L.*y(k);
    end

    fprintf('Point %.2f, Value %.8f \n', [p; P]);

end
```

2.2 Neville's Method

Given a set of points x, and the output of their unknown function f, an interpolating polynomial can be recursively constructed.

$$Q_{i,j} = \frac{(x - x_{i-j})Q_{i,j-1} - (x - x_i)Q_{i-1,j-1}}{x_i - x_{i-j}}$$

Neville's Method reduces the computations required for interpolants and therefore reduces the round-off error from lagrange.

```
function [P] = neville(x, y, p)

    n = length(x);
    Q = zeros(n, n);
    Q(:,1) = y(:);

    for i=1:n-1
        for j=1:(n-i)
            Q(j,i+1) = ((p-x(j))*Q(j+1,i)+(x(j+i)-p)*Q(j,i))/(x(j+i)-x(j));
        end
    end

    P = Q(1, n);

end
```

2.3 Newton Divided Differences

Given a set of points x and the output of their unkown function f. A divided difference formula is recursively applied.

$$f[x_i, ..., x_{i+k}] = \frac{f[x_{i+1}, ..., x_{i+k}] - f[x_i, ..., x_{i+k-1}]}{x_{i+k} - x_i}$$

These functions are then used to find the interpolating polynomial P_n.

$$P_n(x) = f[x_0] + \sum_{k=1}^{n} \left[f[x_0, ..., x_k] \prod_{i=0}^{k-1} (x - x_i) \right]$$

Divided Differences can have additional points given to it without recalculation of the polynomial.

```
function [P, v] = divided_differences(x, y, p)

    n = length(x);
    D = zeros(n,n);
    D(:,1) = y;
    for j=2:n,
        for k=j:n,
            D(k,j) = (D(k,j-1)-D(k-1,j-1))/(x(k)-x(k-j+1));
        end
    end

    P = diag(D);

    if exist('p', 'var')
        v = P(1);
        for i=1:n-1
            v = v + P(i+1)*prod(p*ones(1,i)-x(1:i));
        end
    end
end
```

2.4 Newton Equally Spaced Differences

Given a set of equally spaced points x and the output of their unkown function f, ordered in increasing order. A forward difference formula is recursively applied.

$$P_n(x) = f(x_0) + \sum_{k=1}^{n} \binom{s}{k} \Delta^k f(x_0)$$

If the equally spaced points are in decreasing order, then backward differences should be used.

$$P_n(x) = f[x_n] + \sum_{k=1}^{n} (-1)^k \binom{-s}{k} \Delta^k f(x_n)$$

The Divided Difference algorithm can then be applied.

2.5 Hermite's Method

Hermite Polynomials H agree on the value of the function and its derivative at the points given.

Given a set of points x and the values of their function f and its derivative f'. Using divided differences and creating virtual nodes z to represent the derivative values, the function can be interpolated.

```
function [P, v] = hermite(x, y, yp, p)

    n = length(x);
    z = zeros(2*n,1);
    Q = zeros(2*n,2*n);

    for i=1:n
        z(2*i-1) = x(i);
        z(2*i) = x(i);
        Q(2*i-1,1) = y(i);
        Q(2*i,1) = y(i);
        Q(2*i,2) = yp(i);
        if i~=1
            Q(2*i-1,2) = (Q(2*i-1,1)-Q(2*i-2,1)) / (z(2*i-1)-z(2*i-2));
        end
    end

    for i=2:2*n-1
        for j=2:i
            Q(i+1,j+1) = (Q(i+1,j)-Q(i,j))/(z(i+1)-z(i-j+1));
        end
    end

    P = diag(Q);

    if exist('p', 'var')
        v = P(1);
        for i=1:2*n-1
            v = v + P(i+1)*prod(p*ones(i,1)-z(1:i));
        end
    end
end
```

2.6 Natural Cubic Splines

Splines break the domain into piecewise portions, using a different polynomial for each portion. They must be continuous to the second derivative across pieces.

Given a set of points x and the values of their function f. A spline S_j is constructed.

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

A natural spline will be linear at its bounds a, b. That is, $S''(a) = 0$ and $S'(b) = 0$. A system of equations is constructed and solved.

```
function [S] = natural_cubic_splines (x, y, p)

    n = length(x)-1;
    m = n - 1;
    A = y;
    h = zeros(1, n);
    alpha = h;

    for i=1:n
```

```
        h(i) = x(i+1)-x(i);
    end

    for i=1:m
        alpha(i+1) = 3.0*(A(i+2)*h(i)-A(i+1)*(x(i+2)-x(i))+A(i)*h(i+1))/(h(i
            +1)*h(i));
    end

    l = zeros(1,n+1);
    mu = l;
    z = l;
    l(1) = 1;
    mu(1) = 0;
    z(1) = 0;

    for i=1:m
        l(i+1) = 2*(x(i+2)-x(i))-h(i)*mu(i);
        mu(i+1) = h(i+1)/l(i+1);
        z(i+1) = (alpha(i+1)-h(i)*z(i))/l(i+1);
    end

    l(end) = 1;
    z(end) = 0;

    B = zeros(1,n+1);
    C = zeros(1,n+1);
    D = zeros(1,n+1);
    C(end) = z(end);

    for i = 0:m
        j = m-i;
        C(j+1) = z(j+1)-mu(j+1)*C(j+2);
        B(j+1) = (A(j+2)-A(j+1))/h(j+1)-h(j+1)*(C(j+2)+2.0*C(j+1))/3.0;
        D(j+1) = (C(j+2)-C(j+1))/(3.0*h(j+1));
    end

    S = [A; B; C; D]';
end
```

2.7 Clamped Cubic Splines

Given a set of points x, the values of their function f, and the values of their first derivative at the endpoints. A spline S_j is constructed. A spline is constructed with the additional endpoint constraint.

```
function [S] = clamped_cubic_splines (x, y, yp)

    n = length(x)-1;
    m = n - 1;
    A = y;
    h = zeros(1, m+1);

    for i=1:n
        h(i) = x(i+1)-x(i);
    end
```

```

    alpha = zeros(1, n+1);
    alpha(1) = 3.0*(A(2)-A(1))/h(1) -3.0*yp(1);
    alpha(end) = 3.0*yp(2) -3.0*(A(n+1)-A(n))/h(n);

    for i=1:m
        alpha(i+1) = 3.0*(A(i+2)*h(i)-A(i+1)*(x(i+2)-x(i))+A(i)*h(i+1))/(h(i
            +1)*h(i));
    end

    l = zeros(1,n+1);
    mu = zeros(1,n+1);
    z = zeros(1,n+1);
    l(1) = 1;
    mu(1) = 0;
    z(1) = 0;

    for i=1:m
        l(i+1) = 2*(x(i+2)-x(i))-h(i)*mu(i);
        mu(i+1) = h(i+1)/l(i+1);
        z(i+1) = (alpha(i+1)-h(i)*z(i))/l(i+1);
    end

    l(n+1) = h(n)*(2-mu(n));
    z(n+1) = (alpha(n+1)-h(n)*z(n))/l(n+1);

    B = zeros(1,n+1);
    C = zeros(1,n+1);
    D = zeros(1,n+1);
    C(n+1) = z(n+1);

    for i = 1:n
        j = n-i;
        C(j+1) = z(j+1)-mu(j+1)*C(j+2);
        B(j+1) = (A(j+2)-A(j+1))/h(j+1)-h(j+1)*(C(j+2)+2.0*C(j+1))/3.0;
        D(j+1) = (C(j+2)-C(j+1))/(3.0*h(j+1));
    end

    S = [A; B; C; D]';
end
```