Software Packages In Physics 1

Chapter 4

Numerical Computation

Dr. Hassan Al-Juwhari

Department of Physics
University of Jordan
Second Semester 2014 / 2015
h.juwhari@ju.edu.jo

4.1 Numerical Evaluation

The *accuracy* is the effective number of these digits which appear to the right of the decimal point. Note that to achieve full consistency in the treatment of numbers, precision and accuracy often have values that do not correspond to integer numbers of digits.

- N[expr] gives the numerical value of expr.
- `N[`*expr*`, `*n*`]` attempts to give a result with *n*-digit precision.
- `Accuracy[`*x*`]` gives the effective number of digits to the right of the decimal point in the number *x*.
- For exact numbers such as integers, `Accuracy[`*x*`]` is `Infinity`.
- `Precision[`*x*`]` gives the effective number of digits of precision in the number *x*.
- `Precision[`*x*`]` gives a measure of the relative uncertainty in the value of *x*.

`Precision[`*x*`]`	the total number of significant decimal digits in *x*
`Accuracy[`*x*`]`	the number of significant decimal digits to the right of the decimal point in *x*

```
xx = {1.61, 21.618034, 321.618034};
$MachinePrecision - Log[10., Abs[xx]]
{Map[Accuracy, xx], Accuracy /@ xx}
Precision /@ xx
```

```
{15.7478, 14.6198, 13.4472}
```

```
{{15.7478, 14.6198, 13.4472}, {15.7478, 14.6198, 13.4472}}
```

```
{MachinePrecision, MachinePrecision, MachinePrecision}
```

```
{N[√2, 4], Precision[%]}
```

```
{1.414, ∞}
```

```
N[π, 20]
```

```
3.1415926535897932385
```

```
x = N[Pi^10, 30]
{Precision[x], Accuracy[x]}
```

```
93648.047476083020973716690184 9
```

```
{30., 25.0285}
```

4.2 Equation Solving

4.2.1 Solve

- Solve[eqns, vars] attempts to solve an equation or set of equations for the variables vars.
- Solve[*eqns*, *vars*, *elims*] attempts to solve the equations for *vars*, eliminating the variables *elims*.
- Equations are given in the form *lhs* == *rhs*.
- Simultaneous equations can be combined either in a list or with &&.
- Solve gives solutions in terms of rules of the form *x* -> *sol*.

```
Clear[x]
Solve[x^3 + q/3 x + r/27 == 0, x]
```

$$\left\{\left\{x \to \frac{1}{3}\left(-\frac{2^{1/3}\,q}{\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}+\frac{\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}{2^{1/3}}\right)\right\},\right.$$

$$\left\{x \to \frac{\left(1+i\sqrt{3}\right)\,q}{3\,2^{2/3}\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}-\frac{\left(1-i\sqrt{3}\right)\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}{6\,2^{1/3}}\right\},$$

$$\left.\left\{x \to \frac{\left(1-i\sqrt{3}\right)\,q}{3\,2^{2/3}\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}-\frac{\left(1+i\sqrt{3}\right)\left(-r+\sqrt{4\,q^3+r^2}\right)^{1/3}}{6\,2^{1/3}}\right\}\right\}$$

```
eqns = {x == 1 + 2 a y, y == 9 + 2 x};
sol1 = Solve[eqns, {x, y}] (*solve for x and y *)
sol2 = Solve[eqns, x, y]   (*solve for x and elimenate y *)
```

$$\left\{\left\{x \to -\frac{1+18\,a}{-1+4\,a},\ y \to -\frac{11}{-1+4\,a}\right\}\right\}$$

$$\left\{\left\{x \to -\frac{1+18\,a}{-1+4\,a}\right\}\right\}$$

```
Solve[-Log[3] + 1/2 Log[2 + y/3] - 1/2 Log[y/3] == -I π/2, y]
```

$$\left\{\left\{y \to -\frac{3}{5}\right\}\right\}$$

4.2.2 NSolve

■ NSolve[*lhs*==*rhs*, *var*] gives a list of numerical approximations to the roots of a polynomial equation.
■ NSolve[{*eqn*$_1$, *eqn*$_2$, … }, {*var*$_1$, *var*$_2$, … }] solves a system of polynomial equations.

```
poly = 3 + 3 x - 7 x^2 - x^3 + 2 x^4 + 3 x^7 - 3 x^8 - x^9 + x^10;
NSolve[poly == 0, x]
```

```
{{x → -1.73205}, {x → -0.868688 - 0.585282 i}, {x → -0.868688 + 0.585282 i},
 {x → -0.496292}, {x → 0.0763556 - 1.14095 i}, {x → 0.0763556 + 1.14095 i},
 {x → 1.}, {x → 1.04048 - 0.56735 i}, {x → 1.04048 + 0.56735 i}, {x → 1.73205}}
```

```
highdegree = x^80 + 4 x^40 + 2 x^2 - 3;
sol = NSolve[highdegree == 0, x];
highdegree /. sol // Chop[#, 10^-2] &
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -0.173841 + 0.0985036 i, -0.173841 - 0.0985036 i, 0.312804, 0.312804, -0.565497 - 1.37761 i,
 -0.565497 + 1.37761 i, -1.55504 - 0.730422 i, -1.55504 + 0.730422 i, 1.30952 + 3.90669 i,
 1.30952 - 3.90669 i, -0.731754, 0, 22.2044 - 21.3828 i, 22.2044 + 21.3828 i,
 -85.0346 - 1.88577 i, -85.0346 + 1.88577 i, 25.8604, 55.4067 + 77.6349 i, 55.4067 - 77.6349 i}
```

```
Clear[eq1, eq2, x, y]
eq1 = x^5 + 4 x^3 + 3 x^2 + 2 x == 10;
NSolve[eq1, x]
```

```
{{x → -1. - 1. i}, {x → -1. + 1. i}, {x → 0.5 - 2.17945 i}, {x → 0.5 + 2.17945 i}, {x → 1.}}
```

```
Clear[eq1, eq2, x, y]
eq1 = x^2 - x y - y^2 == 1;
eq2 = x^3 + 3 x y - y^3 == 9;
NSolve[{eq1, eq2}, {x, y}]
```

```
{{x → 1.76268, y → -2.57952}, {x → -0.43195 - 0.816763 i, y → 0.552813 + 1.71762 i},
 {x → -0.43195 + 0.816763 i, y → 0.552813 - 1.71762 i},
 {x → 1.74511, y → 0.802781}, {x → -2.07194 - 1.87447 i, y → -1.16444 - 1.26905 i},
 {x → -2.07194 + 1.87447 i, y → -1.16444 + 1.26905 i}}
```

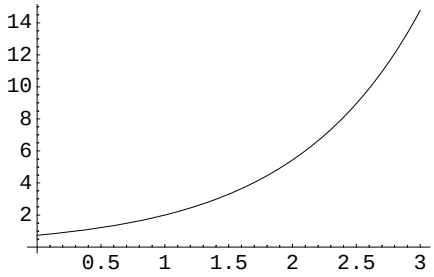
4.2.3 NDSolve

■ NDSolve[*eqns*, *y*, {*x*, x_{min}, x_{max}}] finds a numerical solution to the ordinary differential equations *eqns* for the function *y* with the independent variable *x* in the range x_{min} to x_{max}.
■ NDSolve[*eqns*, *y*, {*x*, x_{min}, x_{max}}, {*t*, t_{min}, t_{max}}] finds a numerical solution to the partial differential equations *eqns*.
■ NDSolve[*eqns*, {y_1, y_2, … }, {*x*, x_{min}, x_{max}}] finds numerical solutions for the functions y_i.
■ NDSolve[*eqns*, *y*[*x*], {*x*, x_{min}, x_{max}}] gives solutions for *y*[*x*] rather than for the function *y* itself.

```
solution = NDSolve[{y'[x] == y[x], y[1] == 2}, y, {x, 0, 3}]
y[1.5] /. solution
Plot[y[x] /. solution, {x, 0, 3}, ImageSize → {200, 150}];
```

```
{{y → InterpolatingFunction[{{0., 3.}}, <>]}}
```

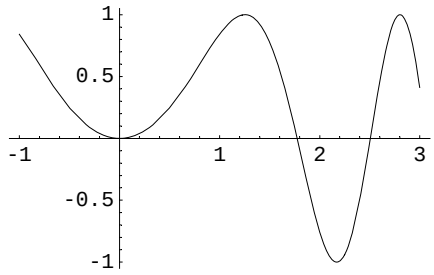
```
{3.29744}
```



4.2.4 FindRoot

■ `FindRoot[`*lhs*`==`*rhs*`, {`*x*`, `x_0`}]` searches for a numerical solution to the equation *lhs*==*rhs*, starting with $x=x_0$.
■ `FindRoot[{`eqn_1`, `eqn_2`, … }, {{`*x*`, `x_0`}, {`*y*`, `y_0`}, … }]` searches for a numerical solution to the simultaneous equations eqn_i.
■ `FindRoot[`*lhs*`==`*rhs*`, {`*x*`, `x_0`, `x_1`}]` searches for a solution using x_0 and x_1 as the first two values of *x*, avoiding the use of derivatives.

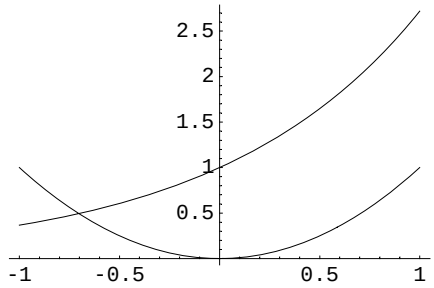
```
Plot[Sin[x^2], {x, -1, 3}, ImageSize → {200, 150}];
FindRoot[Sin[x^2] == 0, {x, 2}]
Sqrt[Pi] // N
```




```
{x → 1.77245}
```

```
1.77245
```

```
Plot[{Exp[x], x^2}, {x, -1, 1}, ImageSize → {200, 150}];
FindRoot[Exp[x] == x^2, {x, -0.5}]
```




```
{x → -0.703467}
```

4.3 Sums And Products

4.3.1 Sum

- Sum[f, {i, i_{max}}] evaluates the sum $\sum_{i=1}^{i_{max}} f$.
- Sum[f, {i, i_{min}, i_{max}}] starts with $i = i_{min}$. Sum[f, {i, i_{min}, i_{max}, di}] uses steps di.
- Sum[f, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, …] evaluates the multiple sum $\sum_{i=i_{min}}^{i_{max}} \sum_{j=j_{min}}^{j_{max}} \ldots f$.
- Sum[f, {i, i_{min}, i_{max}}] can be entered as $\sum_{i=i_{min}}^{i_{max}} f$.

```
{Sum[2 k - 1, {k, 1, 12}], ∑_{k=1}^{12} (2 k - 1)}
```

```
{144, 144}
```

```
Clear[sinx, cosx, x]
Sum[ (-1)^(n + 2) x^(2 n) / (2 n)!, {n, 0, ∞}] // FullSimplify
```

```
Cos[x]
```

```
cosx = Sum[ (-1)^(n + 2) x^(2 n) / (2 n)!, {n, 0, 4}]
Sum[(-1)^(n / 2) x^n / n!, {n, 0, 8, 2}]
cosx /. x → Pi / 6 // N
With[{x = Pi / 6}, Evaluate[cosx]] // N
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320}$$

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320}$$

```
0.866025
```

```
0.866025
```

```
Clear[sinx]
sinx = Sum[(-1)^(n + 2) x^(2 n + 1) / (2 n + 1)!, {n, 0, 4}]
Sum[(-1)^((n + 1) / 2 + 1) x^n / n!, {n, 1, 9, 2}]
{With[{x = Pi / 6}, Evaluate[sinx]] // N, sinx /. x → Pi / 6 // N}
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880}$$

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880}$$

```
{0.5, 0.5}
```

```
Sum[Sum[Sin[x^m + y^n], {n, 1, m}], {m, 1, 3}]
```

$$\sum_{m=1}^{3}\sum_{n=1}^{m}\text{Sin}[x^m + y^n]$$

$$\text{Sin}[x + y] + \text{Sin}[x^2 + y] + \text{Sin}[x^3 + y] + \text{Sin}[x^2 + y^2] + \text{Sin}[x^3 + y^2] + \text{Sin}[x^3 + y^3]$$

$$\text{Sin}[x + y] + \text{Sin}[x^2 + y] + \text{Sin}[x^3 + y] + \text{Sin}[x^2 + y^2] + \text{Sin}[x^3 + y^2] + \text{Sin}[x^3 + y^3]$$

```
With[{n = 25}, Sum[(2 n)! / ((n + k)! (n - k)!), {k, 0, n}]]
Sum[(2 n)! / ((n + k)! (n - k)!), {k, 0, n}] /. n → 25
```

```
626155256640188
```

```
626155256640188
```

4.3.2 Product

- `Product[f, {i, imax}]` evaluates the product $\prod_{i=1}^{i_{max}} f$.
- `Product[f, {i, imin, imax}]` starts with $i = i_{min}$. `Product[f, {i, imin, imax, di}]` uses steps *di*.
- `Product[f, {i, imin, imax}, {j, jmin, jmax}, ... ]` evaluates the multiple product $\prod_{i=i_{min}}^{i_{max}} \prod_{j=j_{min}}^{j_{max}} \ldots f$.
- $\prod$ can be entered as `:prod:` or `\[Product]`.
- `Product[f, {i, imin, imax}]` can be entered as $\prod_{i=i_{min}}^{i_{max}} f$.

```
{∏_{k=1}^{∞} (1 + 1 / k^2), Product[(1 + 1 / k^2), {k, 1, ∞}]}
```

$$\left\{\frac{\text{Sinh}[\pi]}{\pi}, \frac{\text{Sinh}[\pi]}{\pi}\right\}$$

4.3.3 NSum

- `NSum[f, {i, imin, imax}]` gives a numerical approximation to the sum $\sum_{i=i_{min}}^{i_{max}} f$.
- `NSum[f, {i, imin, imax, di}]` uses a step *di* in the sum.
- `NSum` can be used for sums with both finite and infinite limits.
- `NSum[f, {i, ... }, {j, ... }, ... ]` can be used to evaluate multidimensional sums.

```
{∑_{i=1}^{∞} 1/i^2, Sum[1 / i^2, {i, 1, ∞}], NSum[1 / i^2, {i, 1, ∞}]}
```

$$\left\{\frac{\pi^2}{6},\ \frac{\pi^2}{6},\ 1.64493\right\}$$

4.3.4 NProduct

- NProduct[*f*, {*i*, i_{min}, i_{max}}] gives a numerical approximation to the product $\prod_{i=i_{min}}^{i_{max}} f$.
- NProduct[*f*, {*i*, i_{min}, i_{max}, *di*}] uses a step *di* in the product.

```
{Product[1 + 1/n^2, {n, 1, ∞}], NProduct[1 + 1/n^2, {n, 1, ∞}]}
```

$$\left\{\frac{\text{Sinh}[\pi]}{\pi},\ 3.67608 + 0.\,i\right\}$$

4.4 Integration

4.4.1 Integrate

- Integrate[*f*, *x*] gives the indefinite integral $\int f\, dx$.
- Integrate[*f*, {*x*, x_{min}, x_{max}}] gives the definite integral $\int_{x_{min}}^{x_{max}} f\, dx$.
- Integrate[*f*, {*x*, x_{min}, x_{max}}, {*y*, y_{min}, y_{max}}] gives the multiple integral $\int_{x_{min}}^{x_{max}} dx \int_{y_{min}}^{y_{max}} dy\, f$.
- Integrate[*f*, *x*] can be entered as $\int f\, dx$.

```
{∫ x^n dx, Integrate[x^n, x]}
```

$$\left\{\frac{x^{1+n}}{1+n},\ \frac{x^{1+n}}{1+n}\right\}$$

```
{∫ Exp[a x^2] dx, Integrate[Exp[a x^2], x]}
```

$$\left\{\frac{\sqrt{\pi}\ \text{Erfi}\left[\sqrt{a}\ x\right]}{2\sqrt{a}},\ \frac{\sqrt{\pi}\ \text{Erfi}\left[\sqrt{a}\ x\right]}{2\sqrt{a}}\right\}$$

```
Integrate[x^n, {x, 0, 1}, Assumptions → (n > 2)]
```

$$\frac{1}{1+n}$$

```
Assuming[x ∈ Reals, ∫ 1/x^(1/3) dx]
Integrate[1 / x^(1 / 3), x, Assumptions → (x ∈ Reals)]
```

$$\frac{3\,x^{2/3}}{2}$$

$$\frac{3\,x^{2/3}}{2}$$

4.4.2 NIntegrate

■ `NIntegrate[`*f*`, {`*x*`, `*x*$_{min}$`, `*x*$_{max}$`}]` gives a numerical approximation to the integral $\int_{x_{min}}^{x_{max}} f\, dx$.

```
{∫_0^1 Sec[Tan[x]] dx, NIntegrate[Sec[Tan[x]], {x, 0, 1}]}
```

```
{∫_0^1 Sec[Tan[x]] dx, 2.19614}
```

```
Timing[∫_0^∞ Cos[3 x]/Cosh[2 x] dx]
Timing[Integrate[Cos[3 x]/Cosh[2 x], {x, 0, ∞}]] // N
Timing[NIntegrate[Cos[3 x]/Cosh[2 x], {x, 0, ∞}]]
```

```
{0.078 Second, 1/4 π Sech[3 π/4]}
```

```
{0.078 Second, 0.147555}
```

```
{0. Second, 0.147555}
```

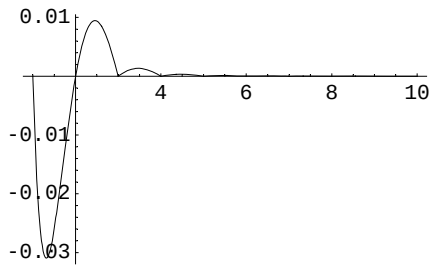
4.4.3 InterpolatingFunction

■ `InterpolatingFunction[`*domain*`, `*table*`]` represents an approximate function whose values are found by interpolation.
■ `InterpolatingFunction[ … ][`*x*`]` finds the value of an approximate function with a particular argument *x*.

```
t1 = Table[{x, 1/x}, {x, 1, 10}]
p1 = Interpolation[t1]
Plot[1/x - p1[x], {x, 1, 10}, PlotRange → All, ImageSize → {200, 150}];
```

```
{{1, 1}, {2, 1/2}, {3, 1/3}, {4, 1/4}, {5, 1/5}, {6, 1/6}, {7, 1/7}, {8, 1/8}, {9, 1/9}, {10, 1/10}}
```

```
InterpolatingFunction[{{1, 10}}, <>]
```



4.5 Optimization

4.5.1 NMaximize

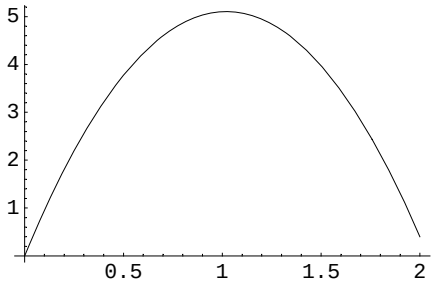
■ NMaximize[*f*, {*x*, *y*, … }] maximizes *f* numerically with respect to *x*, *y*, … .
■ NMaximize[{*f*, *cons*}, {*x*, *y*, … }] maximizes *f* numerically subject to the constraints *cons*.
This gives the maximum value, and where it occurs.

```
NMaximize[x/(1 + Exp[x]), x]
```

```
{0.278465, {x → 1.27846}}
```

```
Clear["Global`*"]
y = v0 t - 0.5 g t^2 /. {v0 → 10., g → 9.8};
NMaximize[y, t]
Plot[y, {t, 0, 2}, ImageSize → {200, 150}];
```

```
{5.10204, {t → 1.02041}}
```



4.5.2 NMinimize

■ NMinimize returns a list of the form {f_{min}, {x -> x_{min}, y -> y_{min}, … }}.
This minimizes the function within the unit circle.

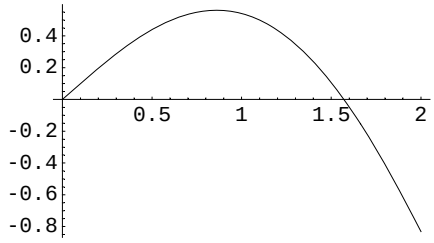
```
Clear["Global`*"]
NMinimize[{Cos[x] - Exp[x y], x^2 + y^2 < 1}, {x, y}]
```

```
{-0.919441, {x → 0.795976, y → 0.605329}}
```

4.5.3 FindMaximum

■ FindMaximum[*f*, {*x*, x_0}] searches for a local maximum in *f*, starting from the point $x = x_0$.
■ FindMaximum[*f*, {{*x*, x_0}, {*y*, y_0}, … }] searches for a local maximum in a function of several variables.

```
Plot[x Cos[x], {x, 0, 2.}, ImageSize → {200, 150}];
{FindMaximum[x Cos[x], {x, 2}], NMaximize[x Cos[x], x]}
```

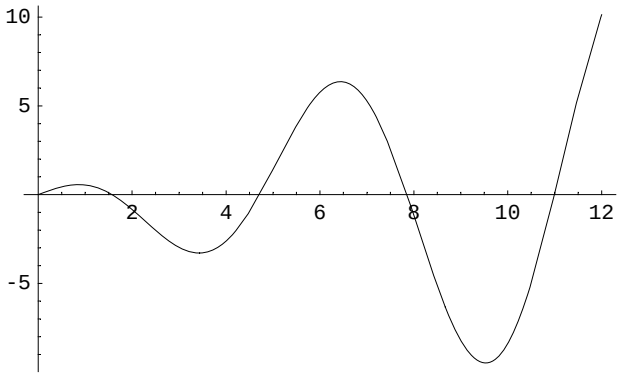



```
{{0.561096, {x → 0.860334}}, {0.561096, {x → 0.860334}}}
```

4.5.4 FindMinimum

■ `FindMinimum[`*f*`, {`*x*`, `x_0`}]` searches for a local minimum in *f*, starting from the point $x = x_0$.
■ `FindMinimum[`*f*`, {{`*x*`, `x_0`}, {`*y*`, `y_0`}, … }]` searches for a local minimum in a function of several variables.

```
Plot[x Cos[x], {x, 0, 12.}];
FindMinimum[x Cos[x], {x, 10}]
```




```
{-9.47729, {x → 9.52933}}
```

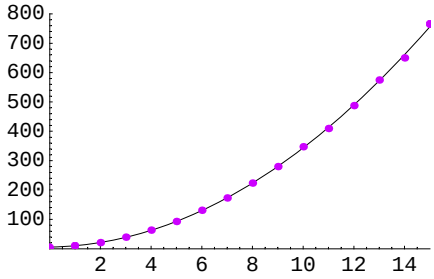
4.5.5 FindFit

■ `FindFit[`*data*`, `*expr*`, `*pars*`, `*vars*`]` finds numerical values of the parameters *pars* that make *expr* give a best fit to *data* as a function of *vars*.
■ The data can have the form `{{`x_1`, `y_1`, … , `f_1`}, {`x_2`, `y_2`, … , `f_2`}, … }`, where the number of coordinates *x*, *y*, … is equal to the number of variables in the list *vars*.
■ The data can also be of the form `{`f_1`, `f_2`, … }`, with a single coordinate assumed to take values `1`, `2`, … .
■ `FindFit[`*data*`, `*expr*`, {{`par_1`, `p_1`}, {`par_2`, `p_2`}, … }, `*vars*`]` starts the search for a fit with `{`par_1 `->` p_1`, `par_2 `->` p_2`, … }`.

```
points1 = Table[{x, 4.5 + 2 x + 3.2 x^2 + Random[Real, {-0.05 x^2, 0.05 x^2}]}, {x, 0, 15}]
f2 = c + d x + e x^2;
f1 = FindFit[points1, f2, {c, d, e}, x]
p1 = Plot[f2 /. f1, {x, -1, 15}, AxesOrigin → {0, 0},
   PlotRange → {{0, 15}, {0, 800}}, DisplayFunction → Identity];
p2 = ListPlot[points1, PlotStyle → {PointSize[0.02], Hue[0.8]},
   DisplayFunction → Identity];
Show[{p1, p2}, DisplayFunction → $DisplayFunction, ImageSize → {200, 150}];
```

```
{{0, 4.5}, {1, 9.72719}, {2, 21.4737}, {3, 39.3622}, {4, 63.9382}, {5, 93.4226},
 {6, 131.029}, {7, 173.059}, {8, 223.948}, {9, 280.355}, {10, 347.388},
 {11, 409.656}, {12, 487.283}, {13, 574.673}, {14, 650.047}, {15, 765.584}}
```

```
{c → 5.79878, d → 1.17288, e → 3.25961}
```



4.6 Data Manipulation

4.6.1 Mean

- `Mean[`*list*`]` gives the statistical mean of the elements in *list*.
- `Mean[{{`x_1`, `y_1`, … }, {`x_2`, `y_2`, … }, … }]` gives `{Mean[{`x_1`, `x_2`, … }], Mean[{`y_1`, `y_2`, … }]}`.

```
m = {-1.849637, 4.11, -9.113, -0.4,
   3.40442119, 3.33, 9.193399786, -2.07, 5.32343, 5.9607825};
{Mean[m], Total[m] / Length[m]}
```

```
{1.78894, 1.78894}
```

4.6.2 Variance

- `Variance[`*list*`]` gives the statistical variance of the elements in *list*.
- `Variance[`*list*`]` is equivalent to `Total[(`*list*`-Mean[`*list*`])^2]/(Length[`*list*`]-1)`.

```
{Variance[m], Total[(m - Mean[m])^2] / (Length[m] - 1)}
```

```
{27.4299, 27.4299}
```

4.6.3 StansardDeviation

■ `StandardDeviation[`*list*`]` gives the standard deviation of the elements in *list*.
■ `StandardDeviation[`*list*`]` is equivalent to `Sqrt[Variance[`*list*`]]`.
■ `StandardDeviation[{{`x_1`, `y_1`, … }, {`x_2`, `y_2`, … }, … }]` gives `{StandardDeviation[{`x_1`, `x_2`, … }], StandardDeviation[{`y_1`, `y_2`, … }]}`.

```
{StandardDeviation[m], Sqrt[Variance[m]]}
```

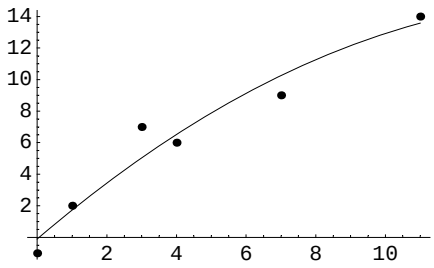
```
{5.23736, 5.23736}
```

4.6.4 Fit

■ `Fit[`*data*`, `*funs*`, `*vars*`]` finds a least-squares fit to a list of data as a linear combination of the functions *funs* of variables *vars*.
■ The data can have the form `{{`x_1`, `y_1`, … , `f_1`}, {`x_2`, `y_2`, … , `f_2`}, … }`, where the number of coordinates *x*, *y*, … is equal to the number of variables in the list *vars*.
■ The data can also be of the form `{`f_1`, `f_2`, … }`, with a single coordinate assumed to take values `1`, `2`, … .
■ The argument *funs* can be any list of functions that depend only on the objects *vars*.
■ `Fit[{`f_1`, `f_2`, … }, {1, x, x^2}, x]` gives a quadratic fit to a sequence of values f_i. The result is of the form a_0 `+` a_1 `x +` a_2 `x^2`, where the a_i are real numbers. The successive values of `x` needed to obtain the f_i are assumed to be `1`, `2`, … .
■ `Fit[{{`x_1`, `f_1`}, {`x_2`, `f_2`}, … }, {1, x, x^2}, x]` does a quadratic fit, assuming a sequence of `x` values x_i.
■ `Fit[{{`x_1`, `y_1`, `f_1`}, … }, {1, x, y}, {x, y}]` finds a fit of the form a_0 `+` a_1 `x +` a_2 `y`.

```
Clear["Global`*"]
data = {{0, -1}, {1, 2}, {3, 7}, {4, 6}, {7, 9}, {11, 14}};
p1 = ListPlot[data, PlotStyle → PointSize[0.02], DisplayFunction → Identity];
f1 = Fit[data, {1, x, x^2}, x]
p2 = Plot[f1, {x, 0, 11}, DisplayFunction → Identity];
Show[{p1, p2}, ImageSize → {200, 150}, DisplayFunction -> $DisplayFunction];
```

```
-0.113262 + 1.89613 x - 0.0592841 x^2
```



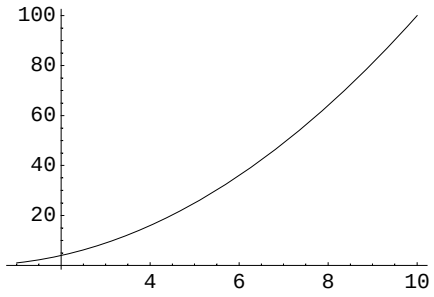
4.6.5 Interpolation

■ `Interpolation[`*data*`]` constructs an `InterpolatingFunction` object which represents an approximate function that interpolates the data.
■ The data can have the forms `{{`x_1`, `f_1`}, {`x_2`, `f_2`}, … }` or `{`f_1`, `f_2`, … }`, where in the second case, the x_i are taken to have values 1, 2, … .

```
data1 = Table[{y, y^2}, {y, 1., 10.}]
sqrt1 = Interpolation[data1]
Plot[sqrt1[x], {x, 1, 10}, PlotRange → All, ImageSize → {200, 150}];
```

```
{{1., 1.}, {2., 4.}, {3., 9.}, {4., 16.}, {5., 25.},
 {6., 36.}, {7., 49.}, {8., 64.}, {9., 81.}, {10., 100.}}
```

```
InterpolatingFunction[{{1., 10.}}, <>]
```



4.6.6 ListInterpolation

■ `ListInterpolation[`*array*`]` constructs an `InterpolatingFunction` object which represents an approximate function that interpolates the array of values given.
■ `ListInterpolation[`*array*`, {{`x_{min}`, `x_{max}`}, {`y_{min}`, `y_{max}`}, … }]` specifies the domain of the grid from which the values in *array* are assumed to come.

`ListInterpolation` is similar to Interpolation, but provides a more convenient interface for data that does not include coordinates and for multidimensional data.
Here is a table of values of a function on a regular three dimensional grid.

```
Clear[data, p1, x, y]
data = Table[Cos[x y], {x, -2., 2.}, {y, -2., 2.}]
p1 = ListInterpolation[data, {{-2, 2}, {-2, 2}}]
{Cos[x y], p1[x, y]} /. {x → 2., y → 2.}
```

```
{{-0.653644, -0.416147, 1., -0.416147, -0.653644},
 {-0.416147, 0.540302, 1., 0.540302, -0.416147},
 {1., 1., 1., 1., 1.}, {-0.416147, 0.540302, 1., 0.540302, -0.416147},
 {-0.653644, -0.416147, 1., -0.416147, -0.653644}}
```

```
InterpolatingFunction[{{-2., 2.}, {-2., 2.}}, <>]
```

```
{-0.653644, -0.653644}
```

4.6.7 FunctionInterpolation

■ `FunctionInterpolation[`*expr*`, {`*x*`, `x_{min}`, `x_{max}`}]` evaluates *expr* with *x* running from x_{min} to x_{max} and constructs an `InterpolatingFunction` object which represents an approximate function corresponding to the result.
■ `FunctionInterpolation[`*expr*`, {`*x*`, `x_{min}`, `x_{max}`}, {`*y*`, `y_{min}`, `y_{max}`}, … ]` constructs an `InterpolatingFunction` object with several arguments.
The function `fint` is an `InterpolatingFunction` object that interpolates over a two-dimensional domain, with `t` varying over the domain of `int` and `x` positive.

```
fint = FunctionInterpolation[t^2 + Sqrt[x],
  {x, 0, 1}, {t, 0, 2}]
{fint[x, t], t^2 + Sqrt[x]} /. {x → .51, t → 1.1}
```

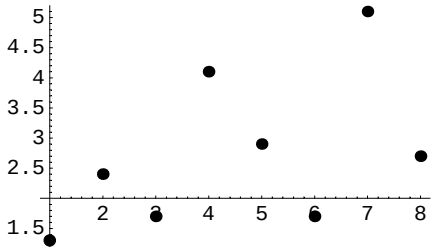
```
InterpolatingFunction[{{0, 1}, {0, 2}}, <>]
```

```
{1.92415, 1.92414}
```

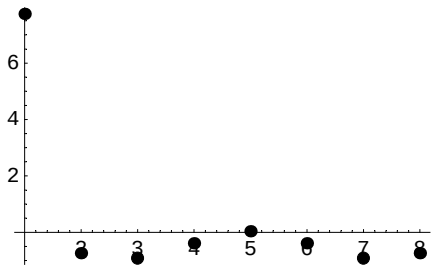
4.6.8 Fourier

■ `Fourier[`*list*`]` finds the discrete Fourier transform of a list of complex numbers.
■ The discrete Fourier transform v_s of a list u_r of length n is by default defined to be $\frac{1}{\sqrt{n}} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$.
■ With the setting `FourierParameters -> {`*a*`, `*b*`}` the discrete Fourier transform computed by `Fourier` is $\frac{1}{n^{(1-a)/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i b (r-1)(s-1)/n}$.

```
Clear[data, p1]
data = {1.3, 2.4, 1.7, 4.1, 2.9, 1.7, 5.1, 2.7};
p1 = ListPlot[data, PlotStyle → PointSize[0.03], ImageSize → {200, 150}];
f1 = Fourier[data]
ListPlot[Re[f1], PlotStyle → PointSize[0.03], ImageSize → {200, 150}];
{InverseFourier[f1], data}
```




```
{7.74282 + 0. i, -0.740685 - 0.677082 i, -0.919239 - 0.954594 i, -0.390685 + 1.72708 i,
 0.0353553 + 0. i, -0.390685 - 1.72708 i, -0.919239 + 0.954594 i, -0.740685 + 0.677082 i}
```




```
{{1.3, 2.4, 1.7, 4.1, 2.9, 1.7, 5.1, 2.7}, {1.3, 2.4, 1.7, 4.1, 2.9, 1.7, 5.1, 2.7}}
```

4.6.9 InverseFourier

■ `Fourier[`*list*`]` finds the discrete Fourier transform of a list of complex numbers.

■ The discrete Fourier transform v_s of a list u_r of length n is by default defined to be $\frac{1}{\sqrt{n}} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$.

■ With the setting `FourierParameters -> {`*a*`, `*b*`}` the discrete Fourier transform computed by `Fourier` is $\frac{1}{n^{(1-a)/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i b (r-1)(s-1)/n}$.

```
{InverseFourier[f1], data}
```

```
{{1.3, 2.4, 1.7, 4.1, 2.9, 1.7, 5.1, 2.7}, {1.3, 2.4, 1.7, 4.1, 2.9, 1.7, 5.1, 2.7}}
```

4.7 Number Representation

Four underlying types of numbers are built into Mathematica.

`Integer`	arbitrary-length exact integer
`Rational`	*integer*`/`*integer* in lowest terms
`Real`	approximate real number, with any specified precision
`Complex`	complex number of the form *number*`+`*number*`I`

`NumberQ[`*x*`]`	test whether *x* is any kind of number
`IntegerQ[`*x*`]`	test whether *x* is an integer
`EvenQ[`*x*`]`	test whether *x* is even
`OddQ[`*x*`]`	test whether *x* is odd
`PrimeQ[`*x*`]`	test whether *x* is a prime integer
`Head[`*x*`]===`*type*	test the type of a number