Table of Contents

1
Scientific Programming

Julia was initially designed as a language aimed at finding solutions to the problems arising from science and mathematics. Current scripting languages then, and to some extent now, were slow especially when dealing with devectorized ('looping') code and resort to use of compile code (e.g. written in 'C') to achieve acceptable executing times. This leads to the 'two-language' approach where analysis is made using the scripting language, whereas code needs to be compiled into a second language, usually C, in order to achieve enterprise performance.

We have seen that Julia compiles its sources to the appropriate machine code using just-in-time compilation from LLVM and so achieves execution times comparable with those of C and Fortran. It is natural that the applications of Julia in the fields of scientific programming are many and varied and in a single chapter I can do no more than point the reader to some of the more elemental examples. However, in a fashion similar to JuliaStats in statistics, a number of community groupings have been formed and these will be noted as we go along; at present perhaps bewilderingly so.

The Julia base and standard library contains most of Python modules NumPy and some of SciPy. In addition, Julia has a great wealth of packages for scientific programming, both specialist and general purpose.
Among the packages, are those that support finance, astronomy, bioinformatics, machine learning plus many others. The general purpose packages may be native implementations of low-level data structures or wrappers around open source libraries.

In this chapter, I am going to concentrate on a few areas between the specialist and general purpose. It is not an exhaustive list, and now in with version 1.0 there are a wealth of packages (currently over 2000 registered ones) which embrace many diverse disciplines.

This chapter will cover:

- Linear Algebra
- Signal Processing
- Differential Equations

- Calculus
- Optimization
- Stochastic Simulations

Linear Algebra

Linear Algebra is one of the area where the existing code has moved from **`BASE`** to **`STDLIB`**, so any work now needs to be prefaces with **`using LinearAlgebra`**.

It is the branch of mathematics concerning vector spaces and linear mappings between such spaces. It includes the study of lines, planes, and subspaces, but is also concerned with properties common to all vector spaces and encompasses matrices and multidimensional arrays.

Julia has excellent support for the latter, mapping as it does, higher order arrays to 1-D vectors, reinterpreted via a set of shape parameters. This means that and array, regardless of its arity can be indexed as a 1-D vector provided that that index does not exceed the total bounds of the array. It is worth noting also that for matrices the apparent order is via columns and for 3-D arrays from planes, then columns etc.

The normal rules for manipulating arrays are available in Julia and we will not spent too much time on them

Arrays can be defined by the [] convention; items on a row being separated by whitespace and individual rows by semicolons.

```
<class="mce-root packt_tip">
```

There are alternative methods to define arrays of a large number items, which we will meet in examples later.

The element type is determined by the highest common type, so in the array below one containing **`Int64`** values is created; if but one of these were specified as (say) 1.0, then a **`Float64`** would be the result.

```
# Define an matrix A and calculate its determinant
julia> using LinearAlgebra
julia> A = [1 -2 2; 1 -1 2; -1 1 1];
julia> det(A)
3.0

# Create a column vector b and perform a matrix division into A
julia> b = [5, 7, 5];
```

```
julia> v = A\b
3-element Array{Float64,1}:
 1.0
 2.0
 4.0

# Display the transpose of v
# Take note of the new type signature of the transpose
julia> transpose(v)
1×3 Transpose{Float64,Array{Float64,1}}:
 1.0 2.0 4.0

# Slicing and dicing of arrays remains the same under version 1
# The use of ':' refers to the entire row (or column) etc.
julia> A1 = A[:, 2:3]
3×2 Array{Int64,2}:
 -2 2
 -1 2
  1 1
```

Leadins

```
# Note again the new, and different, type sig. of the following:
julia> (A1\b)'
1×2 Adjoint{Float64,Array{Float64,1}}:
1.27586 3.9310
julia> A2 = A[1:2,:]; b2 = b[1:2];
julia> (A2\b2)'
1×3 Adjoint{Float64,Array{Float64,1}}:
1.8 2.0 3.6
```

Matrix decompositions

In this section, we are going to consider the process of factorizing a matrix into a product of a number of other matrices. This is termed as matrix decomposition and proves useful in a number of classes of problems. The **** operator hides how the problem is actually solved. Depending on the dimensions of the matrix A (say), different methods are chosen to solve the problem. An intermediate step in the solution is to calculate a factorization of the A matrix. This is a way of expressing A as a product of triangular, unitary, and permutation matrices.

```
<class="packt_infobox">
```

When the matrix is square, it is possible to factorize it into a pair of upper and lower diagonal matrices (U, L) - together with a P permutation matrix such that A = PLU:

```
# Working with the matrix A, defined above
# Its LU decomposition is created by:
julia> Alu = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
  1.0   0.0  0.0
  1.0   1.0  0.0
 -1.0  -1.0  1.0
U factor:
3×3 Array{Float64,2}:
 1.0  -2.0  2.0
 0.0   1.0  0.0
 0.0   0.0  3.0

# We can check that multiplying the L and U components generate the
original matrix
julia> Alu.L*Alu.U  # => A
3×3 Array{Float64,2}
  1.0  -2.0  2.0
  1.0  -1.0  2.0
 -1.0   1.0  1.0
```

Simultaneous Equations

A set of n equations in n unknown quantities will have a unique solution, provided that one of the equations is not a multiple of another. In the latter case, the system is term degenerate since effectively we only have n-1 equations.
Clearly n is a positive number, with n >= 2, since n = 1 is trivial.

The solution of such equations is obtained by matrix methods such as those above.

Consider the case of the following system of equations:

```
 x - 2y + 2z = 5
 x - y  + 2z = 7
-x + y  + z  = 5
```

In matrix notation, we write this as: Av = b, where v is the [x y z] vector of unknowns, A is a matrix of coefficients, and b is a vector of constants on the right-hand side.

```
i.e.  A = [1 -2 2; 1 -1 2; -1 1 1] and b = [5, 7, 5]
```

These are the very two arrays which we used in the first section and observed that they are non-degenerate, i.e. that the matrix A has a computable determinate and so the system of equations has a solution.

The solution is derived by multiplying each side of the equation Av = b by the inverse of A giving: v = inv(A)*b, since inv(A)*A is the identity matrix and this we have already computed already: i.e: (x, y, z) => (1.0. 2.0, 4.0).

We can also use LU factorisation in solving sets of linear, in fact the routines, internally use these very operations in returning their results.

Consider as a second example the following equations:

```
 x - 2y + 2z = 5
 x -  y + 2z = 3
-x +  y +  z = 6
```

This has the same matrix of coefficients but a different vector of constants (on the RH side) and so:

```
#=
The equations are:
 x - 2y + 2z = 5
 x - y  + 2z = 3
-x + y  + z = 6
=#
julia> A = [1 -2 2; 1 -1 2; -1 1 1]
julia> b = [5; 3; 6];

# So Ax = b => LUx = b : x = inv(U)*inv(L)*b
julia> (x,y,z)  =  inv(Alu.U)*inv(Alu.L)*b
3-element Array{Float64,1}: -5.0 -2.0  3.0
```

It is worth noting that while it is possible to call specific factorization methods explicitly (such as `lu`), Julia as the `factorize(A)` function that will compute a convenient factorization, including LU, Cholesky, Bunch-Kaufman, and Triangular, based upon the type of the input matrix.

This function is now part of the STDLIB module LinearAlgebra, so a using must be issued before getting help on the function.

```
julia> using LinearAlgebra
julia> ? factorize
Properties of A            Type of Factorization
---------------            ---------------------
Positive-definite          Cholesky (see cholesky)
Dense Symmetric/Hermitian  Bunch-Kaufman
                           (see bunchkaufman)
Sparse Symmetric/Hermitian LDLt (see ldlt)
Triangular                 Triangular
Diagonal                   Diagonal
```

```
Bidiagonal                Bidiagonal
Tridiagonal               LU (see lu)
Symmetric real tridiagonal LDLt (see ldlt)
General square            LU (see lu)
General non-square        QR (see qr)
```

For example if factorize is called on a Hermitian positive-definite matrix, for instance, then factorize will return a Cholesky factorization.

Also, there are the `!` versions of functions such as `lu!()`, `qr!()` etc. that will compute the decompositions *in place* to conserve memory requirements when dealing with a large number of equations. The reader is referred to Julia's online help for more information on these.

Eigenvalues and eigenvectors

An eigenvector or characteristic vector of a square matrix `A` is a non-zero vector `v` that, when the `A` multiplies `v`, gives the same result as when some scalar multiplies `v` the scalar multiplier is usually denoted by λ

That is: `Av = λv` and is called the eigenvalue or characteristic value o`A` corresponding to `v`. Considering our set of three equations (above), this will yield three *[eigvecs]* eigenvectors and the corresponding *[eigvals]* eigenvalues:

```
julia> using LinearAlgebra
julia> A = [1 -2 2; 1 -1 2; -1 1 1];

# Compute the eigenvalues of A
# (These are complex numbers)
julia> U = eigvals(A)
3-element Array{Complex{Float64},1}: -0.2873715369435107 +
1.3499963980036567im -0.2873715369435107 - 1.3499963980036567im
1.5747430738870216 + 0.0im

# ... and the eigenvectors
julia> V = eigvecs(A)
3×3 Array{Complex{Float64},2}:   0.783249+0.0im         0.783249-0.0im
0.237883+0.0im   0.493483-0.303862im    0.493483+0.303862im  0.651777+0.0im
-0.0106833+0.22483im   -0.0106833-0.22483im   0.720138+0.0im

# The eigenvectors are the columns of the V matrix.
julia> A*V[:,1] - U[1]*V[:,1]

#=
That is, all the real and imaginary parts are of the e-16 order,
```

```
so this is in effect a zero matrix of complex numbers.
=#
3-element Array{Complex{Float64},1}: -2.220446049250313e-16 +
2.220446049250313e-16im  1.1102230246251565e-16 + 0.0im
2.7755575615628914e-16 - 1.3877787807814457e-16im
```

Why do we wish to compute eigenvectors?

Lead ins

- They make understanding linear transformations easy, as they are the directions along which a linear transformation acts simply by "stretching/compressing" and/or "flipping". Eigenvalues give the factors by which this compression occurs.
- They provide another way to affect matrix factorization using singular value decomposition using `svdfact()`.
- They are useful in the study of chained matrices, such as the cat and mouse example we saw earlier.
- They arise in a number of studies of a variety of dynamic systems.

We will finish this section by considering a dynamic system given by:

```
x' = ax + by
y' = cx + dy
```

Here, x' and y' are the derivatives of x and y with respect to time and a, b, c, and d are constants.
This kind of system was first used to describe the growth of population of two species that affect one another and are termed the Lotka-Volterra equations.
We may consider that species x is a predator of species y.

- The more of x, the lesser of y will be around to reproduce.
- But if there is less of y then there is less food for x, so lesser of x will reproduce.
- Then if lesser of x are around, this takes pressure off y, which increases in numbers.
- But then there is more food for x, so x increases.

It also arises when you have certain physical phenomena, such a particle in a moving fluid where the velocity vector depends on the position along the fluid. Solving this system directly is complicated and *we will return to it* in the section on differential equations. However, suppose that we could do a transform of variables so that instead of working with x and y, you could work with p and q that depend linearly on x and y. That is, `p = αx + y` and `w = x + δy`, for some constants α, ., and δ.

The system is transformed into something as follows: `p′ = κ p and q′ = λq` That is, you can "decouple" the system, so that now you are dealing with two independent functions. Then solving this problem becomes rather easy: `p = Aexp(κt)` and `q = Bexp(λt)`.

Then, you can use the formulas for z and w to find expressions for x and y. This results precisely to finding two linearly independent eigenvectors for the [a c; b d] matrix where p and q correspond to the eigenvectors and to the eigenvalues. So by taking an expression that *"mixes"* x and y, and *"decoupling"* it into one that acts independently on two different functions, the problem becomes a lot easier.

This can be reduced to a generalized eigenvalue problem by clever use of algebra at the cost of solving a larger system. The orthogonality of the eigenvectors provides a decoupling of the differential equations, so that the system can be represented as linear summation of the eigenvectors. The eigenvalue problem of complex structures is often solved using finite element analysis, but it neatly generalizes the solution to scalar-valued vibration problems.

Special Matrices

The structure of matrices is very important in linear algebra. In Julia, these structures are made explicit through composite types such as Diagonal, Triangular, Symmetric,Hermitian, Tridiagonal, and SymTridiagonal; specialized methods are written for the special matrix types to take advantage of their structure.

So diag(A) is the diagonal vector of the A but Diagonal(diag(A)) is a special matrix:-

```
julia> Diagonal(diag(A))
3×3 Diagonal{Int64,Array{Int64,1}}: 1   ·  ··  −1  ··   ·  1
```

A symmetric eigenvalue problem

Whether or not Julia is able to detect if a matrix is symmetric/Hermitian, it can have a big influence on how fast an eigenvalue problem is solved. Sometimes it is known that a matrix is symmetric or Hermitian, but due to floating point errors this is not detected by the eigvals function.

In following example, B1 and B2 are almost identical, if however Julia is not told that B2 issymmetric, the elapsed time for the computation is very different.

```
julia> n = 2000;
julia> B = randn(n,n);
julia> B1 = B + B';
julia> B2 = copy(B1);
```

```
julia> B2[1,2] += 1eps();
julia> B2[2,1] += 2eps();

julia> issymmetric(B1)'
true
julia> issymmetric(B2)'
false
```

The`B1` matrix is symmetric whereas `B2` is ***not*** because of the small error added to the cells (1,2) and (2,1).

So if we look at the timings:-

```
julia> @time eigvals(B1);
1.721057 seconds (902.89 k allocations: 74.717 MiB, 4.29% gc time

julia> @time eigvals(B1);
15.516652 seconds (18 allocations: 31.099 MiB, 0.08% gc time)

# However, if we symmetrize B2 and rerun the calculation:
julia> @time eigvals(Symmetric(B2));
2.804628 seconds (8.06 k allocations: 31.652 MiB, 0.46% gc time)
```

Signal Processing

Signal processing is the art of analyzing and manipulating signals arising in many fields of engineering. It deals with operations on or analysis of analog as well as digitized signals, representing time-varying, or spatially-varying physical quantities.

Julia has the functionality for processing signals built into the standard library along with a growing set of packages and the speed of Julia makes it especially well-suited to such analysis.

We can differentiate between `1D` signals, such as audio signals, ECG, variations in pressure and temperature and so on, and `2D` resulting in imagery from video and satellite data streams. In this section, I will mainly focus on the former but the techniques carry over in a straightforward fashion to the `2D` cases.

Frequency analysis

A signal is simply a measurable quantity that varies in time and/or space. The key insight of signal processing is that a signal in time can be represented by a linear combination of

sinusoids at different frequencies.

There exists an operation called the Fourier transform, which takes a x(t) function of time that is called the time-domain signal and computes the weights of its sinusoidal components. These weights are represented as a X(f) function of frequency called the frequency-domain signal.

The Fourier transform takes a continuous function and produces another continuous function as a function of the frequencies of which it is composed. In digital signal processing, since we operate on signals in discrete time, we use the discrete Fourier transform (DFT).

This takes a set of N samples in time and produces weights at N different frequencies. Julia's signal processing library, like most common signal processing software packages, computes DFTs by using a class of algorithms known as fast Fourier transforms (FFT)

Smoothing and Filtering

First we will generate a signal from 3 sinusoids and visualise it

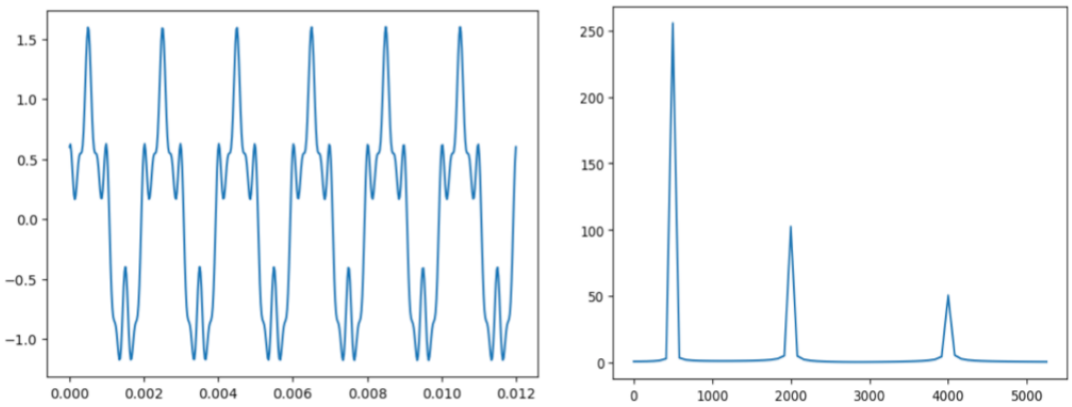
```
julia> using PyPlot

julia> fq = 500.0;
julia> N = 512;
julia> T = 6 / fq;
julia> t = collect(range(0, stop=T, length=N));

julia> x1 = sin.(2π * fq * t);
julia> x2 = cos.(8π * fq * t);
julia> x3 = cos.(16π * fq * t);
julia> x = x1 + 0.4*x2 + 0.2*x3;
```

Now use the `rfft` function (the real FFT function), since our input signal is composed entirely of real numbers — as opposed to complex numbers. This allows us to optimize by only computing the weights for frequencies from 1 to N/2+1.
The higher frequencies are simply a mirror image of the lower ones, so they do not contain any extra information. We will need to use the absolute magnitude (modulus) of the output of rfft because the outputs are complex numbers. Right now, we only care about the magnitude, and not the phase of the frequency domain signal.

```
julia> X = rfft(x);
julia> sr = N / T;
julia> fd = linspace(0, sr / 2, int(N / 2) + 1);
julia> plot(fd, abs(X)[1:N/8])
```

This transforms the time domain representation of the signal (amplitude versus time) into one in the frequency domain (magnitude versus frequency).

The following figure shows the two representations:



Now we can add some high frequency noise to the signal using:

```
julia> ns = 0.1*randn(length(x));
julia> xn = x + ns;
```

Then use a convolution procedure in the time domain to attempt to remove it. In essence, this is a moving average smoothing technique.

We define a 16-element window and use a uniform distribution, although it might be sensible to use a Gaussian or parabolic one that would weigh the nearest neighbors more appropriately.

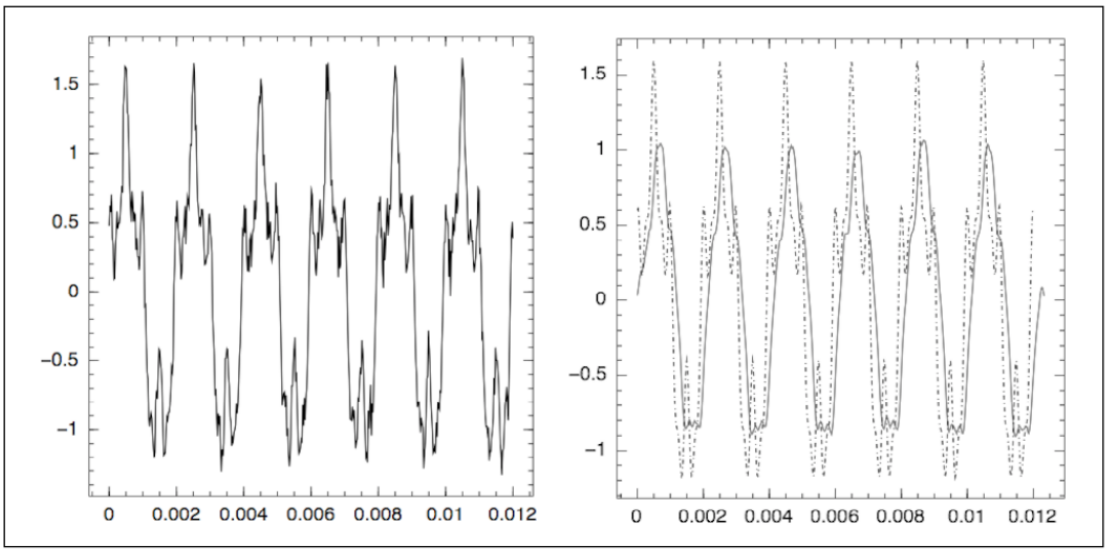
```
M = 16;
xm = ones(Float64, M) / M;
```

It is important that the sum of the weights is 1.

The Julia standard library has a built-in convolution function and applying xm to xn:

```
julia> xf = conv(xn, xm);
julia> t = [0:length(xf) - 1] / sr
```

The following figure shows the noisy signal together with the filtered one:-



The main carrier wave is recovered and the noise eliminated, but given the size of the window chosen the convolution has a drastic effect on the higher frequency components.

Digital signal filters

Moving average filters (convolution) work well for removing noise, if the frequency of the noise is much higher than the principal components of a signal.

A common requirement in RF communications is to retain parts of the signal but to filter out the others. The simplest filter of this sort is a low-pass filter. This is a filter that allows sinusoids below a critical frequency to go through unchanged, while attenuating the signals above the critical frequency. Clearly, this is a case where the processing is done in the frequency domain.

Filters can also be constructed to retain sinusoids above the critical frequency (high pass), or within a specific frequency range (medium band). Julia provides a set of signal processing packages as the DSP group and we will apply some of the routines to filter out the noise on the signal we created in the previous section.

```
julia> using DSP
julia> responsetype = Lowpass(0.2);
```

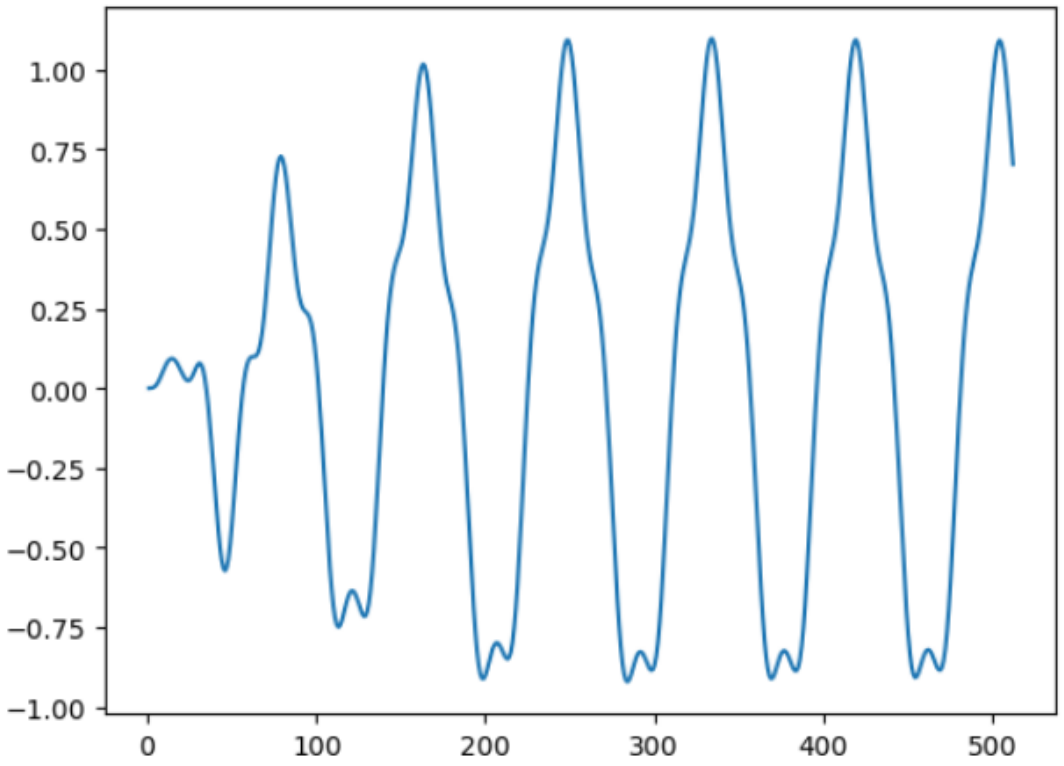
```
julia> prototype = Elliptic(4, 0.5, 30);
julia> tf = convert(TFFilter, digitalfilter(responsetype, prototype));
julia> numerator_coefs = coefb(tf);
julia> denominator_coefs = coefa(tf);
```

This constructs a fourth order elliptic low-pass filter with normalized cut-off frequency 0.2, 0.5 dB of passband ripple, and 30 dB attenuation in the stopband. Then the coefficients of the numerator and denominator of the transfer function will be:

```
julia> responsetype = Bandpass(10, 40; fs=1000);
julia> prototype = Butterworth(4);
julia> xb = filt(digitalfilter(responsetype, prototype), x);
julia> plot(xb)
```

This code filters the data in the x signal, sampled at 1000 Hz, with a fourth order Butterworth bandpass filter between 10 and 40 Hz.

The resultant signal is displayed as follows:



While being cleaner than convolution, this still affects the high frequencies. Also, while the band pass filter is infinite in extent, the one constructed is truncated and this means that the initial portion of the signal is modulated.

Image Processing

Frequency-based methods can be applied to 2D signals, such as those from video and satellite data streams. High frequency noise in imagery is termed "speckle". Essentially, due to the orthogonality of the FFT, processing involves applying a series of row-wise and column-wise FFTs independently of each other.

The DSP package has routines to deal with both 1D and 2D cases. Also, the convolution techniques we looked at in the section on Frequency analysis are often employed in enhancing or transforming images and we will finish by looking at a simple example using a 3x3 convolution kernel.

The kernel needs to be zero-summed, otherwise histogram range of the image is altered. We will look at the lena image that is provided as a 512x512 PGM image:

```
julia> img = open("Files/lena.pgm");
julia> magic = chomp(readline(img));
julia> params = chomp(readline(img));
julia> pm = split(params)

# Remember the GSD
julia> try
  global wd = parse(Int64,pm[1]);
  global ht = parse(Int64,pm[2]);
catch
  error("Can't figure out the image dimensions")
end

# Version 1.0 way of defining a byte array
# readbytes!() will read in place
julia> data = Array{UInt8,2}(undef,wd,ht)
julia> readbytes!(img, data, wd*ht);
julia> data = reshape(data,wd,ht);
julia> close(img);
```

The preceding code reads the PGM image and stores the imagery as a byte array in data, reshaping it to be wd by ht.

Now we define the two 3x3 Gx and Gy kernels as:

```
julia> Gx = [1 2 1; 0 0 0; -1 -2 -1];
```

```
julia> Gy = [1 0 -1; 2 0 -2; 1 0 -1];
```

The following loops over blocks of the original image applying Gx and Gy, constructs the modulus of each convolution, and outputs the transformed image as dout, again as a PGM image.

```
#=
We need to be a little careful that the imagery is still preserved as a
byte array:
=#
julia> dout = copy(data);
julia> for i = 2:wd-1
 for j = 2:ht-1
 temp = data[i-1:i+1, j-1:j+1];
 x = sum(Gx.*temp)
 y = sum(Gy.*temp)
 p = Int64(floor(sqrt(x*x + y*y)))
 dout[i,j] = (p < 256) ? UInt8(p) : 0xff
 end
end

# ... and output the result
julia> out = open("lenaX.pgm","w");
julia> println(out,magic);
julia> println(out,params);
julia> write(out,dout);
julia> close(out);

# This only works if you have Imagemagick (or similar) installed
run(`display lenaX.pgm`);
```

Lead in

Differential Equations

Differential equations are those that have terms that involve the rates of change of variates as well as the variates themselves. They arise naturally in a number of fields, notably dynamics. When the changes are with respect to one dependent variable, most often the systems are called *ordinary* differential equations. If more than a single dependent variable is involved, then they are termed *partial* differential equations.

Julia has a number of packages that aid the calculation of differentials of functions and to solve systems of differential equations and these are grouped together under the community group `JuliaDiffEq` and are now encapsulated as a suite for numerically solving differential equations covered by an envelope package DifferentialEquations.jl, whose purpose is to supply efficient Julia implementations of solvers for wide variety differential equations.

Equations covered by this package include:

- Discrete equations (function maps, discrete stochastic simulations)
- Ordinary differential equations (ODEs)
- Split and Partitioned ODEs (Symplectic integrators, IMEX Methods)
- Stochastic ordinary differential equations (SODEs or SDEs)
- Random differential equations (RODEs or RDEs)

- Differential algebraic equations (DAEs)
- Delay differential equations (DDEs)
- Mixed discrete and continuous equations (Hybrid Equations, Jump Diffusions
- (Stochastic) partial differential equations ((S)PDEs) with both finite difference and finite element methods

Ordinary differential equations

To look first at the simplest of the above (ODEs) we will use the Lotka-Volterra species equations referred to earlier but to make the model a little more interesting will add an intermediate species, which is eaten by the predator and itself preys on the third species.

Some examples might be considered as ecosystems some examples might be three-species ecosystems, such as *owl-snake-mouse*, *foxes-rabbits-vegetation*, and falcon-sparrow-worm.

It is implicit in these models that the predators only eat their prey, such as pandas eating bamboo and koalas eating eucalyptus leaves.

We can write this system as a coupled set of three first-order differential equations

```
 x' = a*x – b*x*y
 y' = –c*y + d*x*y – e*y*z
 z' = –f*z + g*y*z
where x, y and z are the three species
and we require a,b,c,d,e,f g > 0
```

In the preceding equations `a, b, c, d` are in the 2-species Lotka-Volterra equations, `e` represents the effect of predation on species y by species z, `f` represents the natural death rate of the predator z in the absence of prey, and `g` represents the efficiency and propagation rate of the predator z in the presence of prey.

I(f we pass the parameters as a vector `p` and represent the species as a second vector `u` , this translates to the following set of linear equations, defined as a Julia function:

```
function ff(d,u,p,t)
  u[1] =  p[1]*u[1] – p[2]*x[1]*x[2]
  u[2] = –p[3]*u[2] + p[4]*u[1]*u[2] – p[5]*u[2]*u[3]
  u[3] = –p[6]*u[3] + p[7]*u[2]*u[3]
end
```

The vector `d` is the derivatives of `u` and `t` an array corresponding to the time span

Given this function this can be solved once the initial conditions are set and the time range fixed, using the package OrdinaryDiffEq (*itself part of DifferentialEquations.jl*)

```
julia> u0 = [0.5; 1.0; 2.0];  # Setup the initial conditions
julia> tspan = (0.0,10.0);    # and the time range

# Define the ODE problem, this merely sets up the problem ...
# which itself is solved by calling the solve routine

julia> prob = ODEProblem(ff,u0,tspan)
ODEProblem with uType Array{Float64,1} and tType Float64.
In-place: truetimespan: (0.0, 10.0)u0: [0.5, 1.0, 2.0]
```

OrdinaryDiffEq.jl contains some good *"go-to"* choices for ODEs

- AutoTsit5(Rosenbrock23()) handles both stiff and non-stiff equations.
  This is a good algorithm to use if you know nothing about the equation.
- BS3() for fast low accuracy non-stiff.
- Tsit5() for standard non-stiff
  This is the first algorithm to try in most cases.
- Vern7() for high accuracy non-stiff.
- Rodas4() for stiff equations with Julia-defined types, events, etc.
- radau() for really high accuracy stiff equations
  (requires additionally installing ODEInterfaceDiffEq.jl)

So we need to specify which solver to use and the simplest choice would appear to be `Tsit5()`

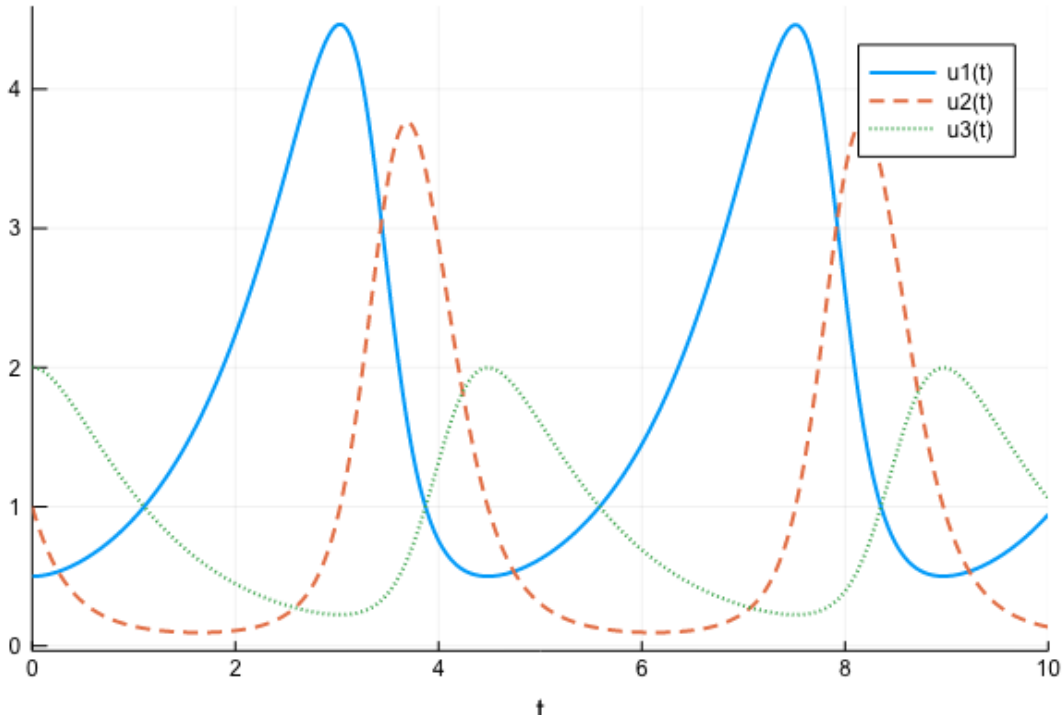
```
u = solve(prob, Tsit5());
```

For those bored with using PyPlot for displaying graphs, I will use the Plots API, together with the GR backend (*its default*); these plus other visualisation examples will be discussed in the next chapter.

Because there is a name clash with the routine `plot`, defined in both PyPlot and GR, it is necessary to fully qualify the function to call the appropriate routine

```
# Plot API will plot the array
julia> using Plots
julia> styles = [:solid; :dash; :dot]
julia> N = length(styles)
julia> styles = reshape(styles, 1, N)
# styles is now a 1xN Vector
```

```
julia> Plots.plot(u, line = (2,styles))
```



This model assumes the z species does not directly eat x.

This might not be true, for example, for the owl-snake-mouse ecosystem, were owls also eat mice; so in this case we would add an additional term: `x' = a*x - b*x*y - h*xz`
Redoing the model as: `d[1] = p[1]*u[1] - p[2]*u[1]*u[2] - p[8]*u[1]*u[3]`
There is an addition constraint that the sum of p5 and p8 must be 1.0, to save the first species from over-eating.
The solution is given in the Jupyter workbook accompanying the chapter and we see the apparently the peak populations of the three species are little altered by the extra term save that the periodicity is almost doubled.

Non-linear differential equations

Non-linear ODEs differ from their linear counterparts in a number of ways. They may contain functions, such as sine, log, and so on, of the independent variable and/or higher powers of the dependent variable and its derivatives. A classic example is the double pendulum that comprises of one pendulum with another pendulum attached to its end. It is

a simple physical system that exhibits rich dynamic behavior with strong sensitivity to initial conditions that under some instances can result in producing chaotic behaviors.

The example we are going to look at is somewhat simpler, a non-linear system arising from chemistry. We will consider the temporal development of a chemical reaction. The reaction will generate heat by the $e^{-E/RT}$ Arrhenius function and lose heat at the boundary proportional to the (T - T0) temperature gradient. Assuming the reaction is gaseous, then we can ignore heat transfer across the vessel. So the change in temperature will be given by: It is possible to write $e^{-E/RT} \sim e^{-E/RT^0 (T - T^0)}$, which means at the low temperature, behaviour is proportional to the exponential value of the temperature difference $\theta = T - T_0$.

$$\frac{d\theta}{dt} = e^{-a\theta} - b\theta$$

Although the ODE package is capable of handling non-linear systems, I will look at a solution that utilizes the alternative Sundials package.

Sundials is a wrapper around a C library, which should be installed when adding the package in Julia; it provides:

- CVODES: for integration and sensitivity analysis of ODEs.
  CVODES treats stiff and non-stiff ODE systems such as y' = f(t,y,p), $y(t_0) = y_0(p)$ where p is a set of parameters.
- IDAS: for integration and sensitivity analysis of DAEs. IDAS treats DAE systems of the form: F(t,y,y',p) = 0, $y(t_0) = y_0(p)$, $y'(t_0) = y_0'(p)$.
- KINSOL; For solution of non-linear algebraic systems.
  KINSOL treats *fixed point* non-linear systems, i.e. of the form F(u) = 0.

By redefining the time scales, it is possible to simplify the equations and write the equations in terms of temperature difference x1 and reactant concentration x2 as:

```
x1 = x2^n * exp(x1) - a*x1
x2 = -b*x2^n * exp(x1)
```

Note we have added an addition term to allow for the fact that the fuel is limited and so the temperature will eventually return even after a *catastrophic* explosion

```
julia> function exotherm(t, x, dx; n=1, a=1, b=1)
 p = x[2]^n * exp(x[1])
 dx[1] = p - a*x[1]
```

```
 dx[2] = -b*p
 return(dx)
end

julia> using Sundials
julia> t = collect(range(0.0; stop=5.0,length=1001));
julia> fexo(t,x,dx) = exotherm(t, x, dx, a=0.6, b=0.1);

julia> x1 = Sundials.cvode(fexo, [0.0, 1.0], t)
1001x2 Array{Float64,2}: 0.0         1.0         0.00500386  0.999499
0.0100153   0.998995    0.0150347   0.99849     0.0200619   0.997982
0.025097    0.997471
 ........    ........
 ........    ........
 ........    ........
 1.05301     0.000137291 1.04986     0.000137722

julia> PyPlot.plot(x1[:.1])
# Need to qualify the plot call is we have used the Plots API above
```

The solutions are stable for b = 1.8, but starts to increase for lower values of a and are only pulled back by depletion of the fuel. If we start solving this for a simple case without any fuel depletion: `x1 = exp(x1) – b*x1` and solve this equations easily using the Roots package and the `find_zero()` routine

```
julia> using Roots, Printf
julia> f(x,a) = exp(x) - b*x;

# Simple calculus gives that is no solution for b <= exp(1)2.7182
julia> for p = 2.8:-0.02:2.6
try
 ff(x) = f(x,p)
 @printf "%.2f : %.5f\n" p find_zero(ff,1.0)
 catch
 error("No convergence for parameter value: $p")
 end
end

2.80 : 0.77594
2.78 : 0.80279
2.76 : 0.83547
2.74 : 0.87909
2.72 : 0.96487

No convergence for parameter value: 2.7
Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] top-level scope at printf.jl:145
```

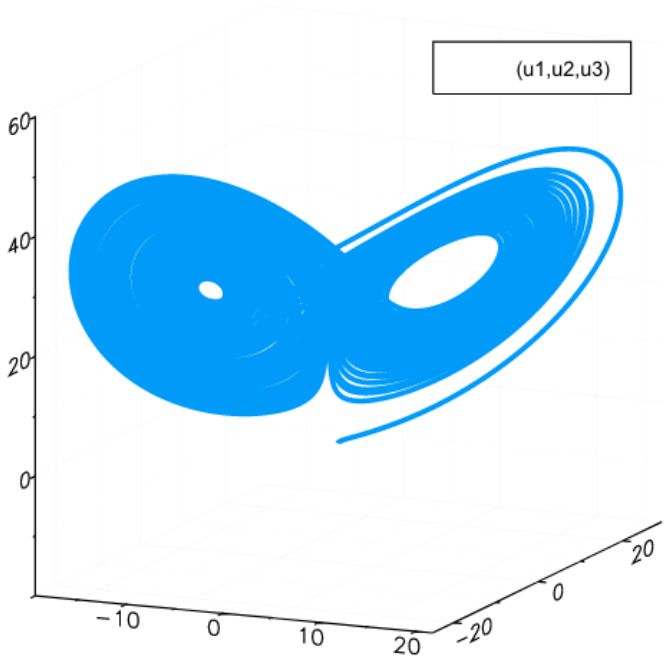
```
[3] top-level scope at In[30]:8
```

A Touch of Chaos

To illustrate the power of ODE solvers we will conclude this section by modelling the strange attractor chaotic system first derived by Edward Lorentz and who coined the term the Butterfly effect.

The following code defines the problem, solves it, using a CVODE_Adams (from the Sundials package) and displays the solution, via the Plots API to show a 3-D visualization, all in 10 lines of code.

```
# The parameter (28.0) is chosen so the equation are chaotic.
julia> function lorenz(du,u,p,t)
  du[1] = 10.0(u[2]-u[1])
  du[2] = u[1]*(28.0-u[3]) - u[2]
  du[3] = u[1]*u[2] - (8/3)*u[3]
end
julia> u0 = [1.0;0.0;0.0];
julia> tspan = (0.0,100.0);
julia> prob = ODEProblem(lorenz,u0,tspan);
julia> sol = solve(prob,CVODE_Adams());
julia> Plots.plot(sol,vars=(1,2,3))
```



The Differential Equation Framework

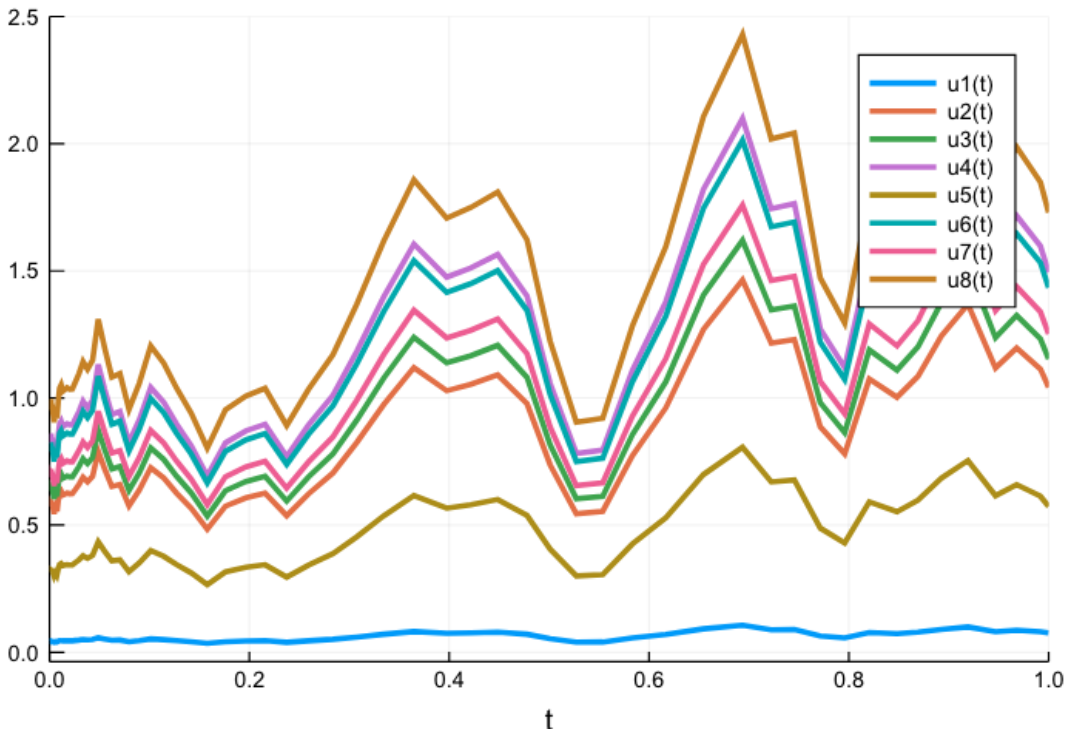
The differential equation framework can be used to solve more and just ODE's.

The example chosen here is a stochastic differential equation (SDE) corresponding to the Wiener process, which describes Brownian motion and arises in many financial models.

An SDE is a differential equation in which one or more of the terms is a random process and so its variation is *modelled* by using a statistical distribution, often, but not always `Gaussian`

```
julia> using DifferentialEquations
julia> f(du,u,p,t) = (du .= u)
julia> g(du,u,p,t) = (du .= u)

# Create 8 initial values for u, as a 4x2 matrix
julia> u0 = rand(4,2)
4×2 Array{Float64,2}: 0.347156  0.302328 0.102206  0.717981 0.990563
0.842297 0.962188  0.983732

# The simplest Weiner process
```

```
julia> W = WienerProcess(0.0,0.0,0.0);

# Define the problem using W and solve the SDE
julia> prob = SDEProblem(f,g,u0,(0.0,1.0),noise=W)
julia> sol = solve(prob,SRIW1());

# Plot the solutions using the Plots API and GR
# using Plots; gr()
julia> Plots.plot(sol)
```



The workbook also contains an example of modelling **jump** equations.

These are differential systems with sudden discontinuous breaks imposed to model sudden raises and falls (*usually falls*) in the stock market.

Calculus

The Calculus package provides tools for working with the basic calculus operations of

differentiation and integration. It can be uses to produce approximate derivatives by several forms of finite differencing or to produce exact derivative using symbolic differentiation.

Differentiation

There are a few basic approaches to using the package are :

- finite-differencing to evaluate the derivative at a specific point
- higher-order functions to create new functions that evaluate derivatives
- symbolic differentiation to produce exact derivatives for simple functions

```
julia> using Calculus
julia> f(x)=sin(x)*cos(x);
julia> derivative(f,1.0)
-0.4161468365471423

# Check since d(f) => cos*cos - sin*sin
julia> cos(1.0)^2 - sin(1.0)^2
-0.4161468365471423

# Possible to curry the function
julia> df = derivative(f)
julia> df(1.0)
-0.4161468365471423

# Also defined is the 2nd derivative
julia> d2f = second_derivative(f)
julia> d2f(1.0)
-1.8185953905296441
```

`Calculus.jl` has some useful 2D functions, i.e. which maps R^2 => R

```
#=
The argument is a N-vector
We need to be careful of name clashes by qualifying the  functions.
=#
julia> h(x) = (1+x[1])*exp(x[1])*sin(x[2])*cos(x[2]);
julia> gd=Calculus.gradient(h);
julia> gd([1.0,1.0])
2-element Array{Float64,1}:  3.7075900080760276 -2.262408767426671

julia> hs = Calculus.hessian(h);
julia> hs([1.0,1.0]
2×2 Array{Float64,2}:  4.94345  -3.39361 -3.39361  -9.88691
```

For scalar functions we can use the ' operator to calculate derivatives as well.

```
julia> f'(1.0)
-0.41614683653632545
```

This operator can be used arbitrarily many times, but note that the approximation worsens with each higher order derivative calculated.

```
julia> f''(1.0)
-1.8185953905296441

julia> f'''(1.0)
1.7473390557101791

julia> f''''(1.0)
5505.591834126032
```

It is possible to output the symbolic version of the derivative using the `differentiate` routine.

```
julia> differentiate("sin(x)*cos(x)", :x)
:((1 * cos(x)) * cos(x) + sin(x) * (1 * -(sin(x))))
```

Although not entirely perfect, this can be somewhat simplified as :

```
julia> simplify(differentiate("sin(x)*cos(x)", :x))
:(cos(x) * cos(x) + sin(x) * -(sin(x)))
```

These techniques work with more than just a single variable:

```
julia> simplify(differentiate("x*exp(-x)*sin(y)", [:x, :y]))
2-element Array{Any,1}:
 :(1*exp(-x)*sin(y) + x*(-1 * exp(-x)) * sin(y) + x*exp(-x)*0) :(0*exp(-
x)*sin(y) + x*0*sin(y) + x*exp(-x)*(1*cos(y)))
```

Here we get a 2-D array corresponding to the partial derivatives and clearly the terms with a zero multiplier can be ignored.

Julia has a community group `JuliaDif` which specifically deals with matters concerning differentiation and for a full list of other packages the reader is referred there.(17, k=5)d

Quadrature

Quadratures are no longer covered in Calculus.jl, an so in this section I will which refer to a tow separate packages: QuadGK.jl and HCubature.jl

To illustrate the use of QuadGK let us use a simple function `sin(x)*(1.0 + cos(x))` and integrate it over the interval : [0.0,1.0]

```
julia> using QuadGK
julia> f(x) = sin(x)*(1.0 + cos(x))
julia> quadgk(f,0.0,1.0)
(0.813734403268646, 2.220446049250313e-16)
```

The function returns a tuple, whose first component is the value of the quadrature and the second an estimate of the error.

The alternate package (HCubature.jl) has been implement written by Steven Johnson *(of PyCall, PyPlot, IJulia, etc.)* and this will compute multidimensional quadratures, however (clearly) it can also be used for 1-D integration providing the same result as QuadGK.

```
julia> using HCubature
julia> hquadrature(f,0.0,1.0)
(0.813734403268646, 2.220446049250313e-16)
```

The power comes when applying to a 2-D (or more) case and the initial conditions provided as vectors.

So for the function $2x*e^{-x}*\sin(x)*\cos*(x)$ we can evaluate the quadrature as:

```
julia> h(u) = 2.0*u[1]*exp(-u[1])*sin(u[2])*cos(u[2])
julia> hcubature(h, [0,0], [1,1]
(0.18710211142604422, 2.598018441709888e-9)
```

Optimization

Mathematical optimisation problems arise in the field of linear programming, machine learning, resource allocation, production planning, and so on.
One well-known allocation problem is that of the travelling salesman who has to make a series of calls, and wishes to compute the optimal route between calls. The problem is not tractable but clearly can be solved exhaustively; however by clustering and tree pruning, the number of tests can be markedly reduced.
The generalised aim is to formulate as the minimisation of some f(x) function for all values of x over a certain interval, subject to a set of $g_i(x)$ restrictions

The problems of local maxima are also included by redefining the domain of x. It is possible to identify three cases:

1. No solution exists.
2. Only a single minimum (or maximum) exists.
   In this case, the problem is said to be convex and is relatively insensitive to the choice of starting value for x.
3. The function f(x) having multiple extreminals.
   For this, the solution returned will depend particularly on the initial value of x.

Approaches to solving mathematical optimization may be purely algebraic or involve the use of derivatives. The former is typically exhaustive with some pruning, the latter is quicker utilizing hill climbing type algorithms.
Optimization is supported in Julia by a community group JuliaOpt and we will briefly introduce three packages: JuMP, Optim, and NLopt.

JuMP

As an example of using JMP we are going to maximise the function 5x + 3y subject to the constraint 3x+5y < 7

```
julia> using JuMP, Clp
julia> m = Model(with_optimizer(Clp.Optimizer))
julia> @variable(m, 0 <= x <= 5 );
julia> @variable(m, 0 <= y <= 10 );
julia> @objective(m, Max, 5x + 3y );
julia> @constraint(m, 2x + 5y <= 7.0 );
```

We define a model and associate it with a specific optimisation method with the `with_optimizer` routine; it is new to v0.19 of the JMP package and at the time of writing is only available in the master branch .

For the optimiser selected the `Clp` must be imported, in addition to `JuMP`

The use of the macros @variable, @objective, and @constraint define the starting domain (x,y) for model m , and the objective function and constraints in a clear manner. Again these have changed recently from @defVar, @setObjective and @addConstraint respectively.

Solving this is (now) a matter of calling the `optimize!()` routine in JuMP

```
julia> JuMP.optimize!(m)
Coin0506I Presolve 1 (0) rows, 2 (0) columns and 2 (0) elementsClp0006I 0
Obj -0 Dual inf 15.5 (2)Clp0006I 1  Obj 17.5Clp0000I Optimal - objective
```

```
value 17.5Clp0032I Optimal objective 17.5 - 1 iterations time 0.002
```

The output is diagnostic from `Clp` but does contain the optimal solution.

This is also available in the model `m` (which `optimise!()` has modified), together with the values of the variable at the optimal solution

```
julia> println("x = ", JuMP.value(x), " and  y = ", JuMP.value(y))
x = 3.5000000000000004 and y = 0.0

julia> JuMP.objective_value(m)
17.500000000000004
```

Knapsack Problem

The solution to the travelling salesman problem is given in the JuMP (GitHub) source. However, this requires the Gurobi solver, so as an alternative we will look at a solution to the knapsack problem.

This is a problem in combinatorial optimization; given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit, and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack, and must fill it with the most valuable items. The knapsack problem has been studied for more than a century, with early works dating as far back as 1897, and nowadays often arises in resource allocation where there are financial constraints; it is studied in fields such as combinatorics, computer science, complexity theory, cryptography, and applied mathematicsusing JuMP, LinearAlgebra, Printf

```
julia> using JuMP, LinearAlgebra, Printf
julia> N = 6;
julia> m = Model()
julia> @variable(m, x[1:N], Bin); # Define array to hold the results
julia> profit = [ 5, 3, 2, 7, 4, 4 ]; # Profit vector of size N
julia> weight = [ 2, 8, 4, 2, 5, 6 ]; # Weights vector of size
julia> maxcap = 15;

# Add the objective and the constraint(s)
julia> @objective(m, Max, dot(profit, x));
julia> @constraint(m, dot(weight, x) <= maxcap);
```

The objective is to maximize profit by a choice of values in the `x` vector, subject to the constraint that any sack can only carry weights up to a total of `maxcap`

The solution takes a similar form to above except for the use of a different optimiser `GLPK`.

```
julia> using GLPK
julia> JuMP.optimize!(m, with_optimizer(GLPK.Optimizer))
julia> println("Objective is : ", JuMP.objective_value(m))
julia> println("\nSolution is :")
julia> for i = 1:N
 print("\tx[$i] = ", JuMP.value(x[i]))
 println(", p[$i]/w[$i] = ", profit[i]/weight[i])
end
Objective is : 16.0Solution is :        x[1] = 1.0, p[1]/w[1] = 2.5
x[2] = 0.0, p[2]/w[2] = 0.375        x[3] = 0.0, p[3]/w[3] = 0.5
x[4] = 1.0, p[4]/w[4] = 3.5        x[5] = 1.0, p[5]/w[5] = 0.8        x[6]
= 1.0, p[6]/w[6] = 0.6666666666666666
```

Optim

Optim is a *native* package, with which calculations are coded in Julia without the need for separate solvers, or third-party libraries. The main call is to the `optimize()` function that requires at least a function definition and vectors the starting values.

Optionally, a value for the solution method can be supplied with one of the following:

- bfgs
- cg
- gradient_descent
- momentum_gradient_descent
- l_bfgs
- nelder_mead
- newton

The values for the optimization process are set by default, but these can be overwritten:

- xtol: Threshold tolerance in x (*1e-32*)
- ftol: Threshold tolerance in f (*1e-32*)
- grtol: Gradient tolerance (*1e-8*)
- iterations: Maximum number of iterations (*1000*)
- store_trace: Stores algorithm's state (*false*)
- show_trace: Outputs algorithm's state (*false*)

As an example consider the Rosenbrock function, which is a non-convex function and often used as a performance test problem for optimization algorithms.

The global minimum is inside a long, narrow, parabolic-shaped flat valleyed to find the

valley is trivial, however, to converge to the global minimum is difficult.

The function is defined by : $f(x, y) = (a-x)^2 + b(y-x^2)^2$
It has a global minimum at: $(x, y)=(a, a^2)$, where $f(x, y) = 0$
Usually the (a,b) parameters are chosen as (1, 100).

We can define the Rosenbrock function and solve the problem using Optim starting at (0.0,0.0) as follows:

```
# Use the BFGS method
# Hard code a = 1 and b = 100 into the function definition
julia> rosenbrock(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
julia> result = Optim.optimize(rosenbrock, zeros(2), BFGS())
Results of Optimization Algorithm * Algorithm: BFGS * Starting Point:
[0.0,0.0] * Minimizer: [0.9999999926033423,0.9999999852005353] * Minimum:
5.471433e-17 * Iterations: 16 * Convergence: true   * |x - x'| ≤ 0.0e+00:
false      |x - x'| = 3.47e-07    * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: false
|f(x) - f(x')| = 1.20e+03 |f(x)|   * |g(x)| ≤ 1.0e-08: true      |g(x)| =
2.33e-09    * Stopped by an increasing objective: false   * Reached Maximum
Number of Iterations: false * Objective Calls: 53 * Gradient Calls: 53
```

So we can see that did indeed converge in 16 iterations to a tolerance of < 1.0e-8

Stochastic Simulations

Problems encountered so far are completely *determined* by the models, and will produce the same solutions repeatedly, even the *chaotic* strange attractor given the same parametrisation and initial conditions.

Some models have terms that occur randomly, and these are called stochastics.

We have already seen examples earlier of the price a volatile stock. While the price increases roughly which the underlying price of money, fluctuations were considered to exist on a day-to-day process sampled from a Gaussian process. Time series analysis is often used to reduce the effect of such fluctuations and reveal the underlying trends, but there are certain systems where the stochastics are paramount. Typical examples are models of queueing systems that might occur in service at banks, or checkouts in supermarkets. I will discuss a particular case of the bank teller later in this section.

Simulations are often dealt with using a framework that attempts to hide the details of the coding as part of the model definition. T his has similarities with the approach we saw with the JuMP package in optimization problems. The approach is not a new one, in fact it was

introduced for simulation problems by IBM with GPSS in 1961; modern day inheritors are jDisco and SimPy.

SimJulia

The SimJulia package is similar to the Python-based SimPy module, although as it is a *native* implementation, neither a wrapper nor requires the use of PyCall. It is both a discrete event and a continuous time process simulation framework.

In discrete event models, the time is advanced in steps and individual events fire according to their schedules/triggers. For continuous events, the change is per-step and computed according to derivative as well as variable values. Clearly, being able to handle discrete events is a prerequisite for stochastic simulations, but the ability to handle continuous events is a useful addition.

The classic example is that of a queue for service in a bank, post office, or grocery shop and we will illustrate the use of SimJulia with such a model.

Bank teller example

Consider example of modeling a bank service with the following assumptions:

- Customers wait in a single queue arriving at random intervals
- There are a number of resources (tellers) who service the next customer in the queue
- Customers may decide to wait or leave depending on the length of the queue

All three may involve stochastic variates and these will need to be generated from a probability distribution, or will be based on actual empirical measurement.

- The actual distributions of arrival rates will not necessarily be Gaussian, since we cannot have negative waiting and service times. Arrivals may differ in a city situation where there is a peak around lunchtimes and a 'rush' towards closing.
- Service times may also be long tailed as some transactions may be more complex than the 'norm', and require considerably more resources.

We will be interested in determining the mean time to serve customers and may be interested in the effects of increasing the number of tellers to match demand, but also balance against the need to minimise the idle time of tellers. The principle aim of such a simulation would be to decide whether it is desirable to increase the resource (tellers) to meet demand; with the obvious trade-off between the cost of the extra tellers balanced

against the extra revenue generated.

We will need the Distributions package to provide density functions to sample against, as we have noted that uniform or normal distributions are inappropriate. We will assume in this example that the arrival times and service times follow an exponential (Poisson) distribution, although in a real simulation we noted that the modeling of these would need to be more exact. Also, we will assume if the queue length is more than a given maximum, the customer does not wait.
First, we need a function to represent the entire customer experience from arriving, through being served, to leaving, and output the relevant time steps and waiting times:

```
julia> using SimJulia, ResumableFunctions
julia> using Distributions
julia> using Printf, Random

# Define the model
julia> NUM_CUSTOMERS = 16   # total number of customers generated
julia> NUM_TELLERS = 2      # number of servers
julia> QUEUE_MAX = 2        # maximum size of queue
julia> μ = 0.4              # service rate
julia> λ = 0.9              # arrival rate

julia> arrival_dist = Exponential(1/λ) # inter-arrival time
julia> service_dist = Exponential(1/μ) # service time

#=
Set the random number seed to generate different results per run
For repeatable results this could be a specific integer value rather than
using the system clock
=#
julia> seed = ccall((:clock,"libc"),Int32,())
julia> Random.seed!(seed);
```

We need to mode the visit and this must be a resumable task as it may be suspended if no resource (i.e. bank teller) is available to service it immediately.

The function below contains a number of `@printf` statements for exemplary purposes, these cn be omitted or else triggered but the addition of a boolean DEBUG constant.

The points of task suspension/resumption are indicated by using the `@yield` macro

```
julia> queue_length = 0;
julia> queue_stack = Array{Integer,1}(undef,0);

julia> @resumable function visit(env::Environment,
                          teller::Resource,
```

```
                          id::Integer,
                          time_arrvl::Float64,
                          dist_serve::Distribution)
# customer arrives
# queue_length is a global and has to be defined as such under v1.0
#
  global queue_length
  @yield timeout(env, time_arrvl)
  @printf "Customer %2d %15s : %.3f\n" id "arrives" now(env)
  if queue_length > 0
    push!(queue_stack,id)
    println("CHECK: Length of the queue is $queue_length")
  end
  queue_length += 1

# customer starts to be served
  @yield request(teller)
  queue_length -= 1
  @printf "Customer %2d %15s : %.3f\n" id "being served" now(env)

# teller is busy
  @yield timeout(env, rand(dist_serve))

# customer leaves
  @yield release(teller)
  @printf "Customer %2d %15s : %.3f\n" id "leaves" now(env)
end
```

Armed with this functional model we need just initialise a simulation and run it.

```
# initialize simulation <: environment
julia> sim = Simulation()

#initialize service resources
julia> service = Resource(sim, NUM_TELLERS)

# initialize customers and set arrival time
# customers arrive randomly baed on Poisson distribution
julia> arrival_time = 0.0
julia> for i = 1:NUM_CUSTOMERS
 arrival_time += rand(arrival_dist)
 @process visit(sim, service, i, arrival_time, service_dist)
end
```

Note that the visits are scheduled by another macro in SimJulia: `@process`

```
# Run the simulation
julia> run(sim)
```

```
Customer  1         arrives : 0.492Customer  1    being served :
0.492Customer  2         arrives : 1.484Customer  2    being served :
1.484Customer  3         arrives : 2.148Customer  1          leaves :
3.784Customer  3    being served : 3.784Customer  2          leaves :
5.943Customer  4         arrives : 6.323Customer  4    being served :
6.323Customer  5         arrives : 6.335Customer  3          leaves :
7.092Customer  5    being served : 7.092Customer  4          leaves :
8.095Customer  6         arrives : 8.123Customer  6    being served :
8.123Customer  5          leaves : 9.034Customer  7         arrives :
9.860Customer  7    being served : 9.860Customer  8         arrives :
10.718Customer  9         arrives : 11.957CHECK: Length of the queue is
1Customer 10         arrives : 13.250CHECK: Length of the queue is
2Customer  6          leaves : 13.291Customer  8    being served :
13.291Customer 11         arrives : 14.107CHECK: Length of the queue is
2Customer  8          leaves : 14.165Customer  9    being served :
14.165Customer  7          leaves : 14.458Customer 10    being served :
14.458Customer  9          leaves : 15.091Customer 11    being served :
15.091Customer 12         arrives : 15.392Customer 13         arrives :
16.693CHECK: Length of the queue is 1Customer 14         arrives :
16.975CHECK: Length of the queue is 2Customer 10          leaves :
17.670Customer 12    being served : 17.670Customer 12          leaves :
17.836Customer 13    being served : 17.836Customer 15         arrives :
18.990CHECK: Length of the queue is 1Customer 16         arrives :
19.089CHECK: Length of the queue is 2Customer 13          leaves :
20.798Customer 14    being served : 20.798Customer 14          leaves :
21.864Customer 15    being served : 21.864Customer 11          leaves :
23.278Customer 16    being served : 23.278Customer 15          leaves :
24.209Customer 16          leaves : 29.442
```

Notice in this simulation the queue length is never greater than 2 and we can check on the number of customers which had to wait by examine the `queue_stack` array

```
# Check on which customers had to wait
# (Output as its adjoint to get all the results on a single line)
julia> queue_stack'
1×7 Adjoint{Int64,Array{Int64,1}}: 9  10  11  13  14  15  16
```

Summary

This chapter has covered a diverse range of topics arising from the discipline of scientific computing with a certain amount of cherry-picking on my part. Julia is now especially blessed with a number of packages which can be applied to scientific problems and so the reader is encouraged to look at the various community groups for additional information. We began by looking at classical linear algebra problems, the solutions of which are provided by routines from within the Julia BASE and STDLIB systems.

For the remaining sections, we turned to a variety of packages and applied them to examples from signal processing, optimization, and the solution of ordinary and stochastic differential equations and touched upon the support Julia provides for differentiation and integration of functions.

Finally, we looked at the solution of problems such as those which have a random (stochastic) component and saw how this can be simulated by various packages within Julia .
In the next chapter, we will look in greater detail at the question of production of graphics and data visualization and will see that Julia's various approaches are especially rich and diverse.

Index