PADOVA UNIVERSITY PRESS

Dolomites Research Notes on Approximation

Special Issue SA2022 – *Software for Approximation 2022*, Volume 15 · 2022 · Pages 109–140

Approximation Techniques with MATLAB®[*]

Paolo Panarese[a]

Abstract

MATLAB® software allows to analyze many approximation techniques that are fundamental for research and teaching in Numerical Computation, Applied Math and Engineering applications. In this paper, we illustrate MATLAB programming environment to proficiently tackle a broad spectrum of approximation topics, such as function approximation, multiresolution wavelet analysis, radial basis functions, multivariate scattered interpolation, surrogate optimization, kernelized support vector machines, and neural networks to build universal approximators and state estimators.

1 Introduction

MATLAB® (*MATrix LABoratory*) is a powerful environment to probe and delve into computational mathematics techniques. In this paper, we illustrate both numerical and symbolic features of MATLAB language to cover key topics in approximation area. The paper is structured as follows. In Section 1.1 we provide a brief description of MATLAB software. In Section 2, we get started with $1D$ function approximation using classical polynomials, i.e. Bernstein, Jacobi and Fourier. The symbolic engine integrated in MATLAB plays a key role to check orthogonality and compute expansion coefficients. In Section 3, we discuss time-frequency analysis using wavelet transforms and multiresolution analysis. MATLAB has both functions and interactive apps to help decomposition of signals into approximation and detail components. In Section 4, we focus on Radial Basis Functions (RBF) to implement multivariate scattered interpolation. MATLAB integrates RBFs also in other algorithms, such as in surrogate global optimization, where an RBF interpolator is used to build a surrogate of the objective function, and support vector machines with RBF kernels, for both classification and regression. In Section 5, we design Neural Networks (NN) models from scratch, i.e. to build universal approximators and estimate the state of a nonlinear dynamic system, such as an Extended Kalman Filter. We first see how a NN model can be trained in MATLAB to learn its optimal weights, and then exported into Simulink environment for more complex system simulation. In the last Section 6, we provide a list of resources to use MATLAB for teaching, research and in the Cloud. All MATLAB code in this paper has been tested in MATLAB® R2022a and is downloadable from GitHub.

1.1 MATLAB environment and ecosystem

MATLAB® is a comprehensive computational platform, suitable for scientists and engineers to analyze large dataset, develop complex models and design innovative systems. MATLAB has a rich and continuously updated documentation with plenty of working examples that help get started easily. The MATLAB environment includes:

- the MATLAB® programming language, matrix-based and natively designed to "*speak Math*", allowing state-of-art numerical linear algebra, symbolic math and the most natural expressions of computational mathematics, as well as supporting different computational paradigms (Parallel/GPU/Cloud Computing, etc.) and algorithm deployment options;
- the Simulink® environment suitable for modeling and simulate dynamic systems, represented by either block diagrams or other paradigms, such as physical multi-domain (Simscape™) or finite-state machines (Stateflow®), and supporting Model-Based Design methodology, System Engineering tools, automatic code generation, real-time simulation, and testing;
- a variety of add-on libraries, called *Toolboxes*™, specialized in a broad spectrum of scientific disciplines (Optimization, Statistics, Econometrics, Machine Learning, Deep Learning, Signal Processing, Wavelet, Image Processing, System Identification, Fuzzy Logic, etc.), key applications (Artificial Intelligence, Computer Vision, Robotics and Autonomous Systems, Robust Control Systems, Embedded Systems, Power Electronics, Predictive Maintenance, Wireless Communications, etc), and top industries (Automotive, Aerospace, Finance, Energy, Chemical, Biomedical,etc).

Moreover, the wide MATLAB ecosystem includes also:

- interoperability and coexecution with open source frameworks, such as C/C++, Python, PyTorch, TensorFlow, Keras, etc. and automatic code generation tools for C/C++, CUDA, HDL, etc.
- multiple cloud-based solutions to access MATLAB anywhere, anytime: MATLAB Online™, MATLAB Drive™, MathWorks Cloud Center, MATLAB Parallel Server clusters. We can also use containerized image of MATLAB on Docker® using MATLAB Dockerfile on Docker Hub and mpm Package Manager to install toolboxes of choice in our container.
- file sharing with MATLAB File Exchange fed by an active MATLAB community, many support packages with Add-on Explorer, and tools for Open Science, like MATLAB Live Scripts and Jupiter Notebooks with MATLAB kernel;

[*]The preface of this special issue to which the article belongs is given in [6].

[a]Ph.D., EDU Customer Success Engineer, MathWorks Academia – MathWorks srl, via Bertola 34, 10122 Torino - Italy

- integration with Science Gateways portals, like European Grid Infrastructure (EGI), making scientific methodologies and output (such as publications, data, and software) transparent and broadly accessible. MATLAB code can be uploaded on several reproducibility platforms hosting web-based MATLAB, such as Code Ocean or nanoHUB where users can interact with the code, irrespective of whether they are licensed or not.

In Table 2, at the end of this paper, there is a list of useful resources to continue a proficient and deeper usage of MATLAB.

2 Function Approximation

In this section we use MATLAB to compute approximation of 1-D functions through classic polynomials, such as Bernstein, Jacobi, and Fourier series. We leverage here on some of the symbolic capabilities integrated into the numerical MATLAB environment.

2.1 Bernstein Polynomials

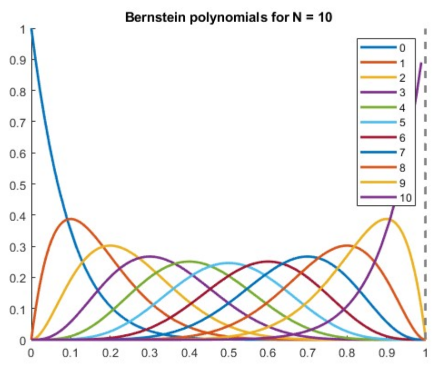
For every positive integer N, Bernstein basis polynomials are $N+1$ polynomials of degree N defined on the real interval $[0,1]$:

$$b_{N,i}\,(x) = \binom{N}{i} x^i\,(1-x)^{N-i}, \qquad x \in [0,1], \quad i = 0,1,2,\ldots,N \tag{2.1.1}$$

For any $N \geq 1$ `bernsteinMatrix(N,x)` command will compute all $N+1$ Bernstein basis polynomials: it returns a symbolic (row) array, whose $(i+1)$-th component is the i-th Bernstein polynomial. For example, for $N=10$, `B=bernsteinMatrix(10,x)` is an array with 11 polynomials, all of degree 10, where `B(1)` is $(x-1)^{10}$ and `B(11)` is x^{10}:

Listing 1: `bernsteinMatrix` command

```
N = 10; syms x real         % N is double, while x is a symbolic object
B = bernsteinMatrix(N,x)    % B is an array of symbolic objects of size 1x(N+1)
B(1), B(11)                 % B(i+1) = i-th Bernstein polynomial, i =0,1,2,..., N
Bex = expand(B)'            % expand symbolic polynomials
```



$$\begin{gathered}
x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1\\
-10\,x^{10} + 90\,x^9 - 360\,x^8 + 840\,x^7 - 1260\,x^6 + 1260\,x^5 - 840\,x^4 + 360\,x^3 - 90\,x^2 + 10\,x\\
45\,x^{10} - 360\,x^9 + 1260\,x^8 - 2520\,x^7 + 3150\,x^6 - 2520\,x^5 + 1260\,x^4 - 360\,x^3 + 45\,x^2\\
-120\,x^{10} + 840\,x^9 - 2520\,x^8 + 4200\,x^7 - 4200\,x^6 + 2520\,x^5 - 840\,x^4 + 120\,x^3\\
210\,x^{10} - 1260\,x^9 + 3150\,x^8 - 4200\,x^7 + 3150\,x^6 - 1260\,x^5 + 210\,x^4\\
-252\,x^{10} + 1260\,x^9 - 2520\,x^8 + 2520\,x^7 - 1260\,x^6 + 252\,x^5\\
210\,x^{10} - 840\,x^9 + 1260\,x^8 - 840\,x^7 + 210\,x^6\\
-120\,x^{10} + 360\,x^9 - 360\,x^8 + 120\,x^7\\
45\,x^{10} - 90\,x^9 + 45\,x^8\\
10\,x^9 - 10\,x^{10}\\
x^{10}
\end{gathered}$$

Figure 1: Bernstein basis polynomials for $N=10$. On the left: the graphs of $N+1$ polynomials. On the right: the expanded algebraic expressions.

Any real-valued continuous function f can be uniformly approximated by a weighted combination of Bernstein basis polynomials. Putting $f_i := f\left(\frac{i}{N}\right)$, the Bernstein polynomial approximation of f is

$$B_N(f)(x) := \sum_{i=0}^{N} f_i\; b_{N,i}\,(x) \overset{N\to\infty}{\longrightarrow} f\,(x), \quad \text{uniformly on } [0,1] \tag{2.1.2}$$

The larger N, the better the approximation. `bernstein` command efficiently evaluates the Bernstein polynomial at any fixed x by means of the numerically stable *de Casteljau's recursive algorithm*: $\forall x \in [0,1]$

$$\begin{cases} f_i^{(0)} := f_i \\ f_i^{(j)} := \,(1-x)\cdot\, f_i^{(j-1)} + x \cdot f_{i+1}^{(j-1)}, \quad i = 0,1,\ldots,N, \quad j = 1,2,\ldots,N \end{cases} \tag{2.1.3}$$

Listing 2: `bernstein` command

```
N = 10; syms x real
f = cos(4*pi*x).*exp(-x);   % f is continuous on [0 1] (another symbolic object)
Bf = bernstein(f, N, x)     % using de Casteljau's algorithm
fplot(Bf, [0,1])
```

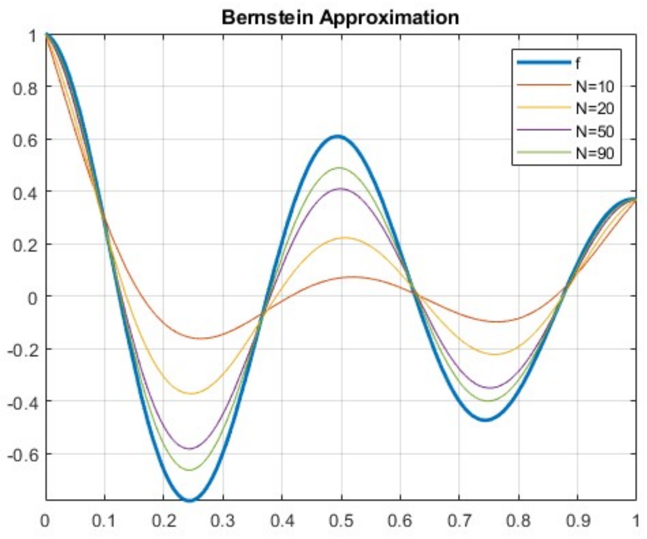


Figure 2: Bernstein polynomial approximation of a given function f with increasing number of terms $N = 10, 20, 50, 90$. Convergence is generally slow, so we need to take large N to get closer to the function.

Application of Bernstein polynomials to Computer Graphics. Bernstein basis polynomials are useful to build *Bézier curves*, after the French engineer Pierre Bézier who used them in the 1960s for designing curves for the bodywork of Renault cars. Given $N + 1$ control points in $\mathbb{R}^d$, with $d = 2$ or $d = 3$, we can compute Bézier curves by simply multiplying the Bernstein matrix B, which is a $1x(N + 1)$ row, by the control points coordinates CP, which is $(N + 1)xd$.

Listing 3: Bézier curves with $d = 2$

```
CP = [0 1; 2 4; 6 3; 5 1; 0 1];          %(N+1)xd double matrix of control points (CP)
N = size(CP,1)-1;                        % N = (number of CP) - 1
syms t real , B = bernsteinMatrix(N,t);  % B is 1x(N+1) symbolic row
phi = simplify(B*CP)';                   % B*CP is 1xd Bezier curve in R^d
fplot(phi(1), phi(2), [0, 1])
```

Interestingly, the algebraic multiplication $B * CP$ of symbolic array B by numerical matrix CP returns the parametric representation of the Bézier curve $\varphi(t) = (\varphi_1(t), ..., \varphi_d(t))$. For example, with $d = 2$, the Bézier curve for 5 control points, having the last point equal to the first, is a closed loop with parametric equations of degree 4 as follows. See Fig. 3 (left)

$$\varphi(t) = \begin{pmatrix} 4\,t\left(2\,t^3 - 7\,t^2 + 3\,t + 2\right) \\ 12\,t^3 - 24\,t^2 + 12\,t + 1 \end{pmatrix} \tag{2.1.4}$$

2.2 Orthogonal Polynomials

MATLAB Symbolic Math Toolbox allows to compute the most widely used families of orthogonal polynomials. Any two distinct polynomials e_n and e_m are *orthogonal* with respect to a particular weight function $\omega(x)$ defined on a real interval $\Omega \subset \mathbb{R}$:

$$e_n \perp e_m \Leftrightarrow \int_\Omega e_n(x)e_m(x)\omega(x)dx = 0, \quad \forall n \neq m \tag{2.2.1}$$

Orthogonal Polynomial	**Interval Ω**	**Weight $\omega(x)$**
`jacobiP(n,a,b,x)`	$[-1,1]$	$(1-x)^a(1+x)^b$
`gegenbauerC(n,a,x)`	$[-1,1]$	$(1-x^2)^{a-1/2}$
`chebyshevT(n,x)`	$[-1,1]$	$(1-x^2)^{-1/2}$
`chebyshevU(n,x)`	$[-1,1]$	$(1-x^2)^{1/2}$
`legendreP(n,x)`	$[-1,1]$	1
`laguerreL(n,x)`	$[0,\infty)$	e^{-x}
`hermiteH(n,x)`	$(-\infty,\infty)$	e^{-x^2}

Jacobi polynomials (known also as *hypergeometric polynomials*) form a complete set of orthogonal functions on the interval $[-1, 1]$ with respect to the weight function $\omega(x) = (1 - x)^a(1 + x)^b$, with $a, b > -1$. So, Jacobi polynomials are described by two

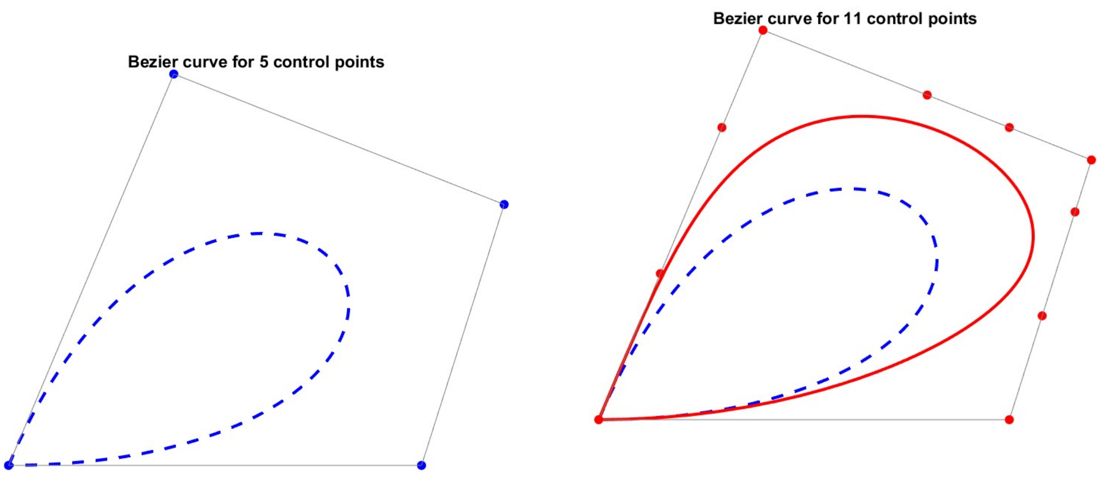

Bezier curve for 5 control points
Bezier curve for 11 control points

Figure 3: Bézier curve for a given set of control points in $\mathbb{R}^2$. On the left: 5 control points. By choosing the last point equal to the first, Bezier curve will be a closed loop. On the right: 11 control points. The more points we add, the more Bézier curve gets closer to the convex hull.

parameters a, b and can be represented as:

$$J_n^{(a,b)}(x) = \frac{1}{2^n}\sum_{k=0}^{n}\binom{n+a}{k}\binom{n+b}{n-k}(x-1)^{n-k}\,(x+1)^k, \qquad n = 0,1,2,...,N \tag{2.2.2}$$

Chebyshev and Legendre polynomials are just a special case, with $a = b = \pm 1/2$ and $a = b = 0$, respectively. See Fig. 4 and note that the weight function $\omega(x)$ is symmetric when $a = b$, while it is skewed to the left when $a > b$. `jacobiP` is a vectorized command: it means that it can compute all Jacobi polynomials with increasing degree from 0 to N with a single line:

Listing 4: Jacobi Polynomials with respect to a weight function

```
a = 6; b = 6; syms x
w = (1-x)^a*(1+x)^b; fplot(w, [-1 1])        % weight
N = 5; J = jacobiP(0:N, a, b, x)             % Jacobi polynomials of degree 0,1,2,...,N
figure, fplot(J,[-0.6,0.6]), legend(string(0:N))
```

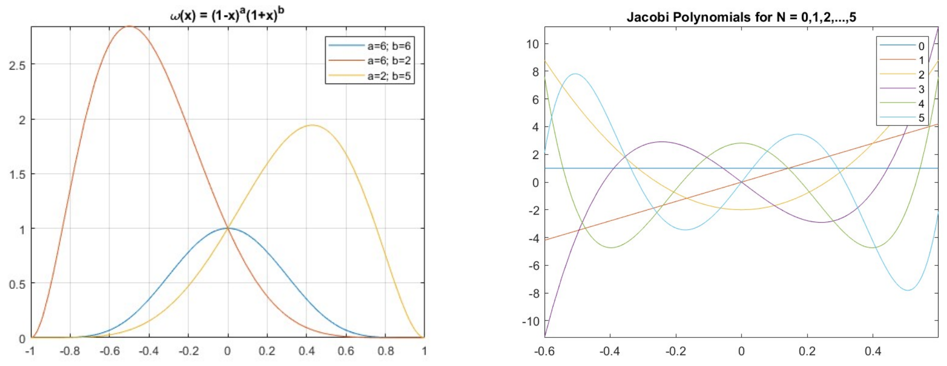


Figure 4: Jacobi polynomials. On the left: the weight function $\omega(x) = (1-x)^a * (1+x)^b$, with different a and b. On the right: the first $N+1$ Jacobi polynomials with $N = 5$ and $a = b = 6$, zoomed in on the interval $[-0.6, 0.6]$.

Orthogonality can be checked out by using the `int` command to compute the definite integral on $[-1, 1]$: $\forall a, b$ and $\forall n \neq m$

Listing 5: Jacobi: check orthogonality with symbolic integration

```
a = 6; b = 6; n = 3; m = 4; syms x real,
w = (1-x)^a*(1+x)^b;
orth = int(jacobiP(n,a,b,x)*jacobiP(m,a,b,x)*w, x,-1,1) % returns 0
```

The squared norm of $J_n^{(a,b)}$, or simply J_n (dropping a and b, for short) is given by

$$C_n^2 = ||J_n||_\omega^2 = \frac{2^k}{2n+k}\,\frac{\Gamma\,(n+a+1)\Gamma\,(n+b+1)}{n!\,\Gamma\,(n+k)} \tag{2.2.3}$$

where $\Gamma(x)$ is the Gamma function and $k = a + b + 1$. As a consequence of completeness of the orthogonal basis $\{J_n\}_{n\in\mathbb{Z}}$, every function f can be approximated by a weighted sum of normalized Jacobi polynomials, the weights ϕ_n being inner products:

$$f(x) \approx \sum_{n=0}^{N} \phi_n J_n(x), \qquad \phi_n = \langle f, J_n \rangle_\omega = \frac{1}{C_n^2} \int_{-1}^{1} f(x)\, J_n(x)\, \omega(x)\, \mathrm{dx} \tag{2.2.4}$$

Jacobi approximation of a given function can be implemented in few steps: choose the number N of approximating terms; compute all symbolic Jacobi polynomials from 0 to N; evaluate them on a numerical array discretizing $[-1, 1]$; iterate on each Jacobi polynomial to compute squared norm and inner product by using `vpaintegral`; finally, perform weighted sum by matrix multiplication:

Listing 6: Jacobi approximation of a function

```
syms x real, fun = sin(2*pi*x)*exp(-x);
a = 6; b = 6; w = (1-x)^a*(1+x)^b;             % weight function
N = 5; J = jacobiP(0:N, a, b, x);              % J is sym with N+1 Jacobi polyn
k = a+b+1;
phi = zeros(1,N+1);                            % preallocation of (N+1) double coeff
for n = 0:N
  Cn_2 = 2^k/(2*n+k)*gamma(n+a+1)*gamma(n+b+1)/(factorial(n)*gamma(n+k)); % squared norm
  phi(n+1) = vpaintegral(fun * J(n+1)*w,x,-1,1)/Cn_2;  % numerical integration
end
JacobiMatrix = subs(J, x, -1:0.01:1);          % evaluate J on a numerical array
fun_approx = phi*JacobiMatrix;                 % phi is 1x(N+1),JacobiMatrix is (N+1)x201
```

Fig. 5 illustrates how Jacobi sums approximate the function better and better as we increase the number of terms.

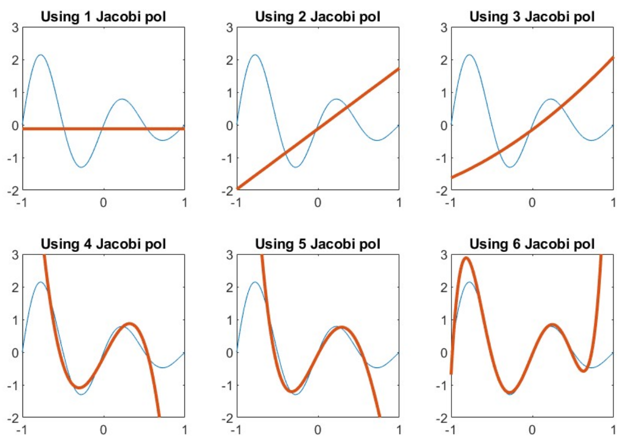


Figure 5: Jacobi approximation of a given function f for increasing number of terms.

Application of Jacobi polynomials in Robotics. Jacobi polynomials have been used in [2] (with $a = b = 6$) to parametrize the trajectory coordinates of a multi-joint robot arm and simulate *discrete movements*, such as reaching, grasping, throwing or kicking. On the contrary, *rhythmic movements*, such as walking, chewing or scratching, have been simulated by trigonometric polynomials (Fourier series, described in next section). Using Jacobi and Fourier approximation has the great advantage of leading to a low dimensional representation of the arm movements in terms of expansion coefficients.

2.3 Fourier Series

On the interval $[0, L]$, for some $L > 0$, we define the set of orthogonal functions

$$e_n(x) := \sin(\lambda_n x), \quad \lambda_n := \frac{n\pi}{L}, \quad \forall n = 1, 2, \ldots \tag{2.3.1}$$

Orthogonality is given by integral inner product:

$$e_n \perp e_m \quad \Leftrightarrow \quad \langle e_n, e_m \rangle := \int_0^L e_m(x)\, e_n(x)\, \mathrm{dx} = 0 \;\; \text{if} \;\; m \neq n \tag{2.3.2}$$

To check orthogonality, we can use `assume` command to specify assumptions on L, m, n and `int` to compute the integral on $[0, L]$. Moreover, the `simplify` command will perform algebraic simplification of the returned symbolic integral:

Listing 7: Fourier: check orthogonality and squared norm

```
syms x m n L
assume(L, "positive")
assume([m n], ["integer" "positive"]), assumeAlso(m~=n)  % assume m~=n
lambda(n) = n*pi/L;
e(n) = sin(lambda(n)*x)
I = simplify(int(e(m)*e(n), 0, L))                  % inner product <e(n),e(m)> returns 0
squared_norm = simplify(int(e(n)*e(n), 0, L))    % squared norm <e(n),e(n)> returns L/2
```

$$||e_n||^2 = \langle e_n, e_n \rangle = \frac{L}{2} \quad \forall n = 1, 2, \ldots \tag{2.3.3}$$

Now, let us consider a piecewise-continuous function $f : [0, L] \to \mathbb{R}$, possibly with some jump discontinuity. So, f can be projected on the orthogonal set $\{e_n\}_{n=1,2,\ldots,N}$ and approximated by the finite weighted sum:

$$f(x) \approx f_N(x) = \sum_{n=1}^{N} b_n\, e_n(x), \qquad b_n = \langle f, e_n \rangle = \frac{2}{L}\int_0^L f(x)\, e_n(x)\, \mathrm{d}x \tag{2.3.4}$$

In MATLAB we can compute Fourier coefficients b_n with the symbolic integration command `int` and implement the symbolic sum with `symsum` command. As an example, consider a sawtooth function with a jump:

$$f(x) = \begin{cases} x, & \text{if } x \in [0, L/2) \\ x - L/2, & \text{if } x \in [L/2, L] \end{cases} \tag{2.3.5}$$

Listing 8: Fourier approximation of a sawtooth function

```
syms x n, L = 1;                                 %  x,n are symbolic objects
assume(n, ["integer" "positive"])
f(x) = piecewise(x<L/2, x, x>=L/2, x-L/2);       % define sawtooth function
b(n) = simplify(2/L* int(f(x) * sin(n*pi/L * x), 0, L))  % Fourier coefficients
b(1:10)                                          % display the first 10 coeff
fplot(f, [0 L], "r"), axis equal, hold on
for N = [10 30]
    f_approx = symsum(b(n) * sin(n*pi/L*x), n, 1, N)     % symbolic sum of N terms
    fplot(f_approx, [0 L]);
end
legend('f(x)', 'N=10', 'N=30', 'Location', 'NW'), title('Fourier approximation')
```

Note that, since `n` is a symbolic object, `b(n)` becomes automatically a symbolic function. More importantly, `n` must be restricted to be positive integer in order to get good expressions of Fourier coefficients. For instance, for the sawtooth function (2.3.5), MATLAB returns the following:

$$b(n) = -\frac{(-1)^{n/2}\left((-1)^{n/2}+1\right)^2}{2\,n\,\pi} \qquad n = 1, 2, \ldots, N \tag{2.3.6}$$

With $N = 10$ we can observe the first ten Fourier coefficients `b(1:10)`:

$$\left(\frac{1}{\pi} \quad 0 \quad \frac{1}{3\pi} \quad -\frac{1}{2\pi} \quad \frac{1}{5\pi} \quad 0 \quad \frac{1}{7\pi} \quad -\frac{1}{4\pi} \quad \frac{1}{9\pi} \quad 0 \right)$$

and Fourier approximation of (2.3.5), truncated at the first $N = 10$ terms, is:

$$f_{10}(x) = \frac{\sin(\pi x)}{\pi} + \frac{\sin(3\pi x)}{3\pi} - \frac{\sin(4\pi x)}{2\pi} + \frac{\sin(5\pi x)}{5\pi} + \frac{\sin(7\pi x)}{7\pi} - \frac{\sin(8\pi x)}{4\pi} + \frac{\sin(9\pi x)}{9\pi} \tag{2.3.7}$$

Of course, with a larger number N of terms, we can get better approximation. Compare with $N = 30$ in Fig. 6.

3 Wavelets and Multiresolution Approximation

Fourier Transform is localized only in frequency, not in time. This is a drawback when we need to describe non-stationary and spiky signals, typically occurring in medical sciences (ECG, EEG), earth sciences (seismography, climatology, etc), or finance (stock prices, trading signals, etc). Wavelets instead are well localized both in time and frequency, so they can track frequencies changing over time. Just like a "mathematical microscope", they allow to zoom in/out singularities at different resolution.

There are two types of Wavelet Transforms: Continuous and Discrete. The Discrete Transform requires a discrete sampling grid and an orthonormal basis, that can be created by the key framework of multiresolution analysis (MRA).

After the initial ideas due to Haar in 1909, wavelets and MRA were developed by the fruitful collaboration in the 80's and 90's among mathematicians, physicists, engineers: Grossman, Morlet, Mallat [15], Meyer [18], Daubechies [8] (see also [22] and [17]). Since then many different kind of wavelets have been developed suitably for specific applications, i.e. audio/image compression, denoising, edge detection. Wavelet techniques can be efficiently implemented in MATLAB.

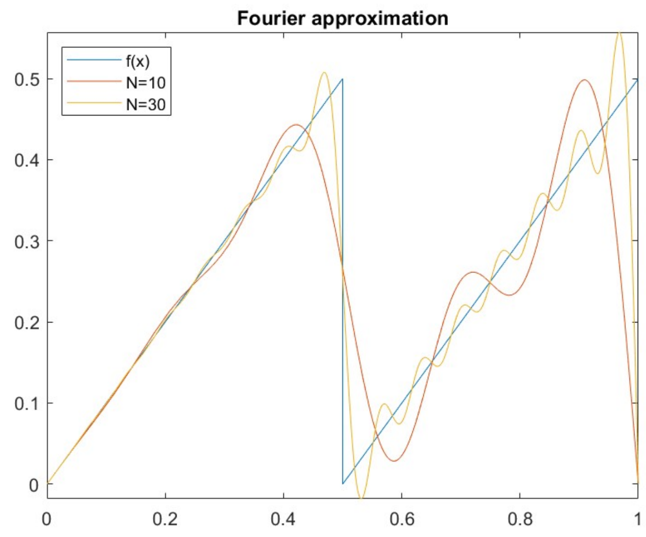


Figure 6: Fourier approximation of the sawtooth function f with jump at $x = 0.5$, comparing $N = 10$ and $N = 30$

3.1 From Fourier Transform to Continuous Wavelet Transform

We compare different transforms by discussing their *localization* properties. From signal theory, recall that:

- any $\psi(t) \in L^2(\mathbb{R})$ is a signal ($t \in \mathbb{R}$ for time), with finite *energy* $E = ||\psi||^2 = \int_{-\infty}^{+\infty} |\psi(t)|^2 dt$ and *Fourier Transform* given by: $\widehat{\psi}(\xi) = \int_{-\infty}^{+\infty} \psi(t)e^{-i\xi t}dt \in L^2(\mathbb{R})$ ($\xi \in \mathbb{R}$ for frequency);
- given ψ with $||\psi|| = 1$, and its Fourier Transform $\widehat{\psi}$, we consider their means μ_ψ and $\mu_{\widehat{\psi}}$ (see *) and variances:

$$(\Delta t)^2_\psi = var(|\psi|^2) = \int_{-\infty}^{+\infty} (t-\mu_\psi)^2|\psi(t)|^2 dt, \qquad (\Delta f)^2_\psi = var(|\widehat{\psi}|^2) = \int_{-\infty}^{+\infty} (\xi-\mu_{\widehat{\psi}})^2|\widehat{\psi}(\xi)|^2 d\xi. \tag{3.1.1}$$

  We call $(\Delta t)_\psi$ and $(\Delta f)_\psi$ the *efficient duration* and *efficient band*, respectively (i.e. where most energy is concentrated).
- the *Heisenberg's Inequality* states that $(\Delta t)_\psi(\Delta f)_\psi \geq C$ must hold for some constant $C > 0$, i.e. as the frequency content of a signal is resolved more finely, we lose information about when in time these events occur, and viceversa.

If $\lambda \in \mathbb{R}^d$, $d = 1, 2$, and ψ_λ is a function parametrized by λ (typically obtained from ψ by translation, modulation or dilation), we can build a "transform" from the t-domain to the λ-domain by: $\phi(t) \mapsto (T\phi)(\lambda) = \int_{-\infty}^{+\infty} \phi(t)\psi_\lambda(t)dt$. The transform is said "localized in time" if the corresponding $(\Delta t)_\lambda$ is small, or "localized in frequency" if $(\Delta f)_\lambda$ is small.

Transform	λ	ψ_λ	$(\Delta t)_\lambda$	$(\Delta f)_\lambda$
Time series	$\lambda = \tau$	$\psi_\tau(t) = \delta(t-\tau)$	0	∞
FFT (Fast Fourier Transform)	$\lambda = \omega$	$\psi_\omega(t) = e^{-i\omega t}$	∞	0
Short-Time FFT (Gabor Transform)	$\lambda = (\omega, \tau)$	$\psi_{\omega,\tau}(t) = g(t-\tau)e^{-i\omega t}$	$(\Delta t)_{\omega,\tau} = (\Delta t)_g$	$(\Delta f)_{\omega,\tau} = (\Delta f)_g$
CWT (Continuous Wavelet Transform)	$\lambda = (a, b)$	$\psi_{a,b}(t) = a^{-1/2}\psi(\frac{t-b}{a})$	$(\Delta t)_{a,b} = a(\Delta t)_\psi$	$(\Delta f)_{a,b} = a^{-1}(\Delta f)_\psi$

Comparing different transforms, we can see that FFT is perfectly localized in frequency ($\Delta f = 0$), but totally blind to when the frequency occurred in time ($\Delta t = \infty$). Short-time FFT and CWT are both localized in time and frequency. Short-time FFT, based on a given window g, has constant Δt and Δf (see Fig. 7 on the left). On the contrary, CWT has resolution windows changing with scale parameter a (see Fig. 7 on the right).

Let us discuss now CWT in detail. A function $\psi(t) \in L^1(\mathbb{R}) \cap L^2(\mathbb{R})$ (real or complex valued), with $||\psi||_{L^2} = 1$, is called a *wavelet* or *mother wavelet* if its Fourier transform $\widehat{\psi}$ satisfies the following *admissibility condition*:

$$0 < C_\psi = \int_0^{+\infty} \frac{|\widehat{\psi}(\xi)|^2}{|\xi|} d\xi < \infty \tag{3.1.2}$$

As $\psi \in L^1(\mathbb{R})$, $\widehat{\psi}(\xi)$ is continuous. So, from (3.1.2), it follows that $\widehat{\psi}(0) = 0$, i.e every mother wavelet has zero mean:

$$\int_{-\infty}^{+\infty} \psi(t)dt = 0. \tag{3.1.3}$$

*$\mu_\psi = \mathbb{E}(|\psi|^2) = \int_{-\infty}^{+\infty} t|\psi(t)|^2 dt$ and $\mu_{\widehat{\psi}} = \mathbb{E}(|\widehat{\psi}|^2) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \xi|\widehat{\psi}(\xi)|^2 d\xi$

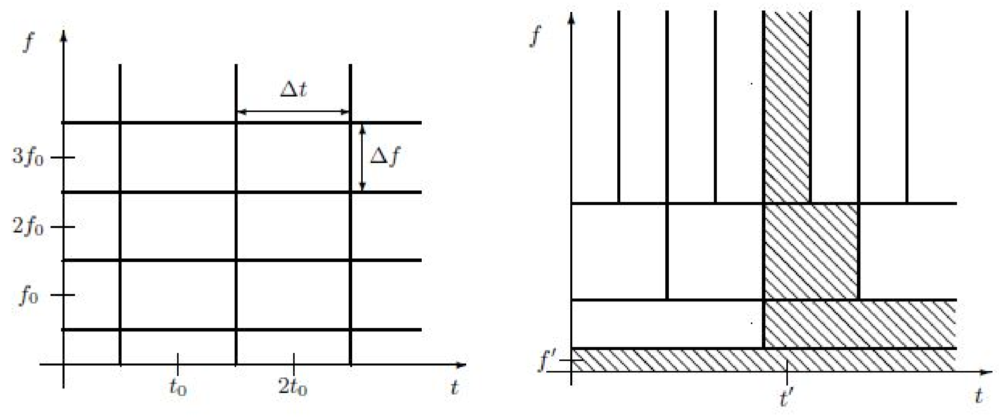


Figure 7: Coverage of Time-Frequency domain: (on the left) by Short-Time FFT, with fixed resolution windows; (on the right) by wavelets: it provides higher frequency-resolution at low frequencies $f = f'$; it provides higher time-resolution at high frequencies to capture a peak at $t = t'$.

Given a mother wavelet ψ, we define a 2-parameter family of *scaled and translated* functions (called *child wavelet*):

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\,\psi\left(\frac{t-b}{a}\right), \quad a > 0, b \in \mathbb{R} \tag{3.1.4}$$

where $a > 0$ is the *scale* or *dilation* parameter, and $b \in \mathbb{R}$ is the *shift* or *translation* parameter.

The Continuous Wavelet Transform (CWT) of a function $f \in L^2(\mathbb{R})$ is given by the inner product with $\psi_{a,b}$:

$$CWT_\psi(f)(a,b) = \langle f, \psi_{a,b}\rangle = \int_{-\infty}^{+\infty} f(t)\,\overline{\psi}_{a,b}(t)\,dt, \quad (a,b) \in \mathbb{R}_+\backslash\{0\}x\mathbb{R} \tag{3.1.5}$$

where $\overline{\psi}_{a,b}$ denotes the complex conjugate of $\psi_{a,b}$ in case of a complex wavelet. Remarkably, if (3.1.2) holds, CWT is invertible (*Calderón-Grossman-Morlet theorem*):

$$f(t) = \frac{1}{C_\psi}\int_{-\infty}^{+\infty}\int_{0}^{+\infty} CWT_\psi(f)(a,b)\frac{1}{\sqrt{a}}\,\psi\left(\frac{t-b}{a}\right)\frac{1}{a^2}da\,db \tag{3.1.6}$$

Examples of wavelets used in CWT. "Mexican Hat" wavelets and Morlet wavelets are real, symmetric and can be defined by an explicit formula. Morse wavelets are analytic, i.e. complex-valued and described through their Fourier transforms, which must be supported only on the positive real axis. Here are some examples of frequently used wavelets:

$$\begin{aligned} &\psi(t) = \frac{2}{\sqrt{3}\,\pi^{1/4}}(1-t^2)e^{-t^2/2} && \text{Mexican Hat or Ricker} \\ &\psi(t) = \cos(5t)\,e^{-t^2/2} && \text{Morlet} \\ &\widehat{\psi}(\xi) = C_{\gamma,P}\,\xi^{P^2/\gamma}\,e^{-\xi^\gamma}, \quad \xi \geq 0 && \text{Morse} \end{aligned} \tag{3.1.7}$$

where in the Morse wavelet, $\gamma \geq 1$ controls the symmetry, $P^2 \in [\gamma, 40\gamma]$ is the time-bandwidth product, and $C_{\gamma,P}$ is a normalizing constant. When $\gamma = 3$, Morse wavelet is symmetric, closely approximates Morlet wavelet, and has the minimum Heisenberg area.

In MATLAB `waveinfo` command returns some information on the wavelets. The `wavefun` command allows to visualize numerical approximations of wavelets: inputs are the wavelet name, and number of iterations; outputs are the values of ψ, and a grid of points where ψ is evaluated:

Listing 9: Wavelets used for CWT: Mexican Hat and Morlet

```
waveinfo('mexh'), waveinfo('morl')          % wavelets information
[psi1,xval1] = wavefun('mexh',10);          % Mexican Hat wavelet
[psi2,xval2] = wavefun('morl',10);          % Morlet wavelet
figure, subplot(1,2,1), plot(xval1, psi1)
subplot(1,2,2), plot(xval2, psi2)
```

Example of CWT and comparison with FFT. In the following example, we show that wavelets have better localization properties than FFT and short-time FFT. We define two sampled signals s_1, s_2, starting from the same frequency values $10, 30, 50, 100Hz$, but with different frequency distribution over time: s_1 contains all the 4 frequencies at any time; s_2 is defined with a single frequency in each of 4 distinct intervals. Moreover, both signals last 1 second and have a peak at $t = 0.5$.

In MATLAB we can use `fft`, `spectrogram` and `cwt` commands to compute Fast Fourier transform, short-time FFT and Continuous Wavelet Transform, respectively. Note that in the code:

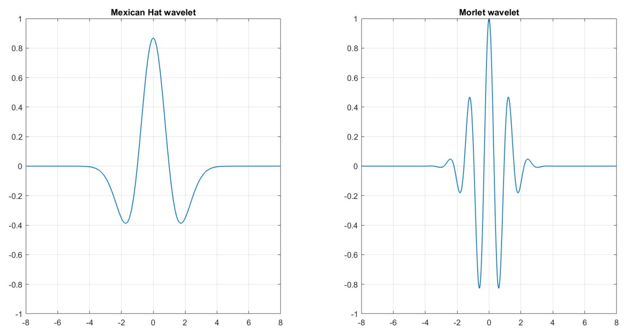


Figure 8: Wavelets $\psi(t)$ used for CWT: Mexican Hat (on the left) and Morlet wavelet (on the right).

- we can apply `fft` just once to compute the FFT of all signals if stored in the columns of a matrix (we say `fft` is "vectorized" and can operate down along each column), while we have to loop over each signal to apply `spectrogram` and `cwt`.
- `spectrogram` with no output arguments plot time-frequency diagram, with frequency along x-axis by default. We can specify sample rate as the 5th input and the frequency location (y-axis) as 6-th input for better comparison.
- `cwt` command uses the analytic Morse wavelet (with parameters $\gamma = 3$ and $P^2 = 60$) by default. Morlet wavelet can also be optionally used. `cwt` with no output arguments plot a scalogram with log of frequency along y-axis by default. To avoid the log of frequency we have to run `cwt` with 2 output arguments and then create a surface.

In Fig. 9, we clearly see that FFT is unable to capture the difference of the two signals (column 2) and totally blind to the peak. On the contrary, short-time FFT and CWT (Column 3 and 4) are able to show the change of frequency across the time. CWT can capture the instant of the peak more accurately.

Listing 10: Wavelets: comparing FFT vs spectrogram vs CWT

```
f = [10 30 50 100]; Fs = 1000;                                  % same 4 frequencies
t = (0:1/Fs:1)'; N = numel(t); N2= floor(N/2);
fvec = (0:N2-1)*Fs/N;
mask = [t<0.25  (t>=0.25&t<0.5) (t>=0.5&t<0.75) (t>0.75)];      % logical matrix
harmonics = cos(2*pi*t*f);                          % matrix with harmonics in 4 columns
signals = zeros(numel(t),2);                        % preallocate matrix for 2 signals
signals(:,1) = sum(harmonics,2);                                % build signal_1
signals(:,2) = sum(harmonics .* mask,2);                        % build signal_2
signals = signals + 5*(t>=0.5&t<=0.501);                        % add peak at t = 0.5
F = abs(fft(signals)).^2/N;                                     % FFT of all signals
figure, tiledlayout(2,4)
for i = 1:size(signals,2)
  nexttile,plot(t, signals(:,i)), ylim([-4 10])                  % plot signals in time
  ax(1)=nexttile;, plot(F(1:N2,i), fvec)                         % plot FFT
  ax(2)=nexttile;, spectrogram(signals(:,i),[],[],[],Fs,'yaxis') % short-time FFT
  colorbar("off")                                                % get rid of colorbar
  [cfs,frq] = cwt(signals(:,i), 'morse', Fs);                    % compute CWT
  ax(3)=nexttile;, surface(t,frq,abs(cfs)), shading flat         % scalogram
  set(ax,'YLim',[0, 150])                                        % set common range
end
```

3.2 Discrete Wavelet Transform and Multiresolution Analysis

The CWT (3.1.5) is continuously defined for any scale $a > 0$ and time-shift $b \in \mathbb{R}$, so it can be numerically approximated through any sampling of the scale-shift plane. The *Discrete Wavelet Transform* (DWT) actually refers to a particular choice of sampling, i.e.

1. DWT uses a standard *dyadic lattice* in (3.1.4) and (3.1.5) with: $a = 2^j$ and $b = n2^j$, for any $j, n \in \mathbb{Z}$ (see [†])
2. DWT uses particular mother wavelets ψ generating *orthonormal* (or biorthogonal) bases of $L^2(R)$:

$$\mathcal{B}^{\psi} = \left\{\psi_{j,n}(t) = 2^{-j/2}\,\psi\left(2^{-j}t - n\right) :\ n \in \mathbb{Z},\ j \in \mathbb{Z}\right\} \tag{3.2.1}$$

3. DWT uses mother wavelets ψ that must be *compactly supported* (hence the entire family (3.2.1) is compactly supported).

[†]This is the choice done in MATLAB and in most engineering and applied sciences. In mathematical literature, we can find $a = 2^{-j}$ and $b = n2^{-j}$, so that (3.2.1) becomes $\psi_{j,n}(t) = 2^{j/2}\,\psi\left(2^j t - n\right)$. In this case, the approximation spaces described in the item 2. would be $V_{j+1} \supset V_j$.

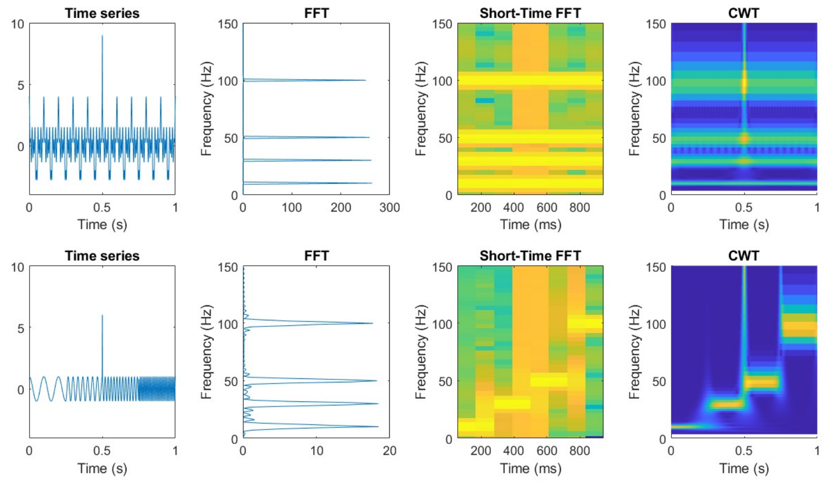


Figure 9: Comparison of FFT, Short-Time FFT and CWT. The two signals on the left (Column 1) contain exactly the same frequencies $10, 30, 50, 100$ Hz, but differently distributed over time. Moreover, they both contain a spike at time $t = 0.5$. FFT (Column 2) just shows the same 4 frequencies, ignoring the different time-distribution and the peak. On the contrary, Short-time FFT and CWT (Column 3 and 4) are able to capture the change of frequency across the time. Moreover, CWT is able to identify the instant of the peak more precisely.

Under these conditions, there is an efficient DWT implementation with a fast (pyramid) algorithm using only finite impulse response (FIR) filters, i.e. the Fast Wavelet Transform, with $O(N)$ complexity, if N is the number of points in the signal (see [23]).

To built such special orthonormal bases (ONB) made of wavelets, Mallat [15] and Meyer [18] developed in 1989 a key framework called *multiresolution analysis* (MRA). Let us review briefly the key elements of MRA.

1. There are two types of wavelet: father and mother. The *father wavelet* φ, called also *scaling function*, with non-zero mean, is responsible to represent the smooth approximation and low-frequency parts of signal, while the *mother wavelet* ψ, with zero mean, is responsible to describe the detail and high-frequency components:

$$\int_{-\infty}^{+\infty} \varphi(t)dt = 1, \qquad \int_{-\infty}^{+\infty} \psi(t)dt = 0 \tag{3.2.2}$$

2. The father wavelet φ "generates" nested closed spaces V_j, called *approximation spaces*, that behave like "zooming grids" at multiple resolution levels j. More precisely, the couple $\left(\{V_j\}_{n\in\mathbb{Z}},\ \varphi \in V_0\right)$ is called a *multiresolution analysis* (MRA) of $L^2(\mathbb{R})$ if, by definition, the following conditions hold:
   - $V_{j+1} \subset V_j \subset V_{j-1}$ and $\bigcup_{n\in\mathbb{Z}} V_j$ is dense in $L^2(\mathbb{R})$
   - V_j's are *self-similar in scale*, i.e. $f(t/2) \in V_{j+1} \Leftrightarrow f(t) \in V_j \Leftrightarrow f(2t) \in V_{j-1}$
   - $\mathcal{B}_j^{\varphi} = \left\{\varphi_{j,n}(t) = 2^{-j/2}\,\varphi\left(2^{-j}t - n\right) :\ n \in \mathbb{Z}\right\}$ is a "Riesz basis" [‡] of V_j.

   As $\varphi_{1,0} \in V_1 \subset V_0 = \overline{\operatorname{span}}\{\varphi(t-n) : n \in \mathbb{Z}\}$, we get the *refinement or two-scale equation* for the father wavelet:

$$\varphi(t) = \sqrt{2}\sum_{n\in\mathbb{Z}} h_n \varphi(2t - n) \tag{3.2.3}$$

   where h_n are called *low-pass filter coefficients*.

3. The mother wavelet ψ "generates" mutually orthogonal spaces $W_j \in L^2(\mathbb{R})$, called *detail spaces*. More precisely, given an MRA, we define $W_j \perp V_j$ the orthogonal complement, i.e. $W_j \oplus^{\perp} V_j = V_{j-1}$, and $\psi \in W_0$ is defined, on top of φ [§], such that:
   - $\bigoplus_{j\in\mathbb{Z}}^{\perp} W_j$ is dense in $L^2(\mathbb{R})$, i.e. W_j's are an orthogonal decomposition of $L^2(\mathbb{R})$
   - $\mathcal{B}_j^{\psi} = \left\{\psi_{j,n}(t) = 2^{-j/2}\,\psi\left(2^{-j}t - n\right) :\ n \in \mathbb{Z}\right\}$ is an ONB of W_j. Remarkably, $\mathcal{B}^{\psi} = \bigcup_{j\in\mathbb{Z}} \mathcal{B}_j^{\psi}$ is ONB for $L^2(\mathbb{R})$.

   As $\psi \in W_0 \subset V_{-1} = \overline{\operatorname{span}}\{\sqrt{2}\varphi(2t-n)\}$, we get the *refinement or two-scale equations* for the mother wavelet:

$$\psi(t) = \sqrt{2}\sum_{n\in\mathbb{Z}} g_n \varphi(2t - n) \tag{3.2.4}$$

   where g_n are called *high-pass filter coefficients*.

[‡] $\{e_k\}$ is a *Riesz basis* if $\exists A, B > 0$ such that $\forall f \in L^2(\mathbb{R})$ $A||f||^2 \le \sum_k < f, e_k >^2 \le B||f||^2$. Riesz basis is a generalization of orthonormal basis (for which $A = B = 1$). Their advantage is that they are easier to be found.

[§] Fourier Transform of (3.2.3) shows that $\widehat{\varphi}(\xi) = H(\xi/2)\widehat{\varphi}(\xi/2)$, with $H(\xi) := 1/\sqrt{2}\sum_{n\in\mathbb{Z}} h_n e^{-in\xi}$. If we define $\widehat{\psi}(\xi) := H_1(\xi/2)\widehat{\varphi}(\xi/2)$, with $H_1(\xi) = e^{i\xi}\overline{H(\xi+\pi)}$, we can prove that $< \psi, \varphi(t-n) >= 0$ $\forall n \in \mathbb{Z}$, so $\psi \in W_0$. Moreover, $\mathcal{B}_0^{\psi} = \{\psi(t-n) : n \in \mathbb{Z}\}$ is an ONB for W_0.

4. The low-pass and high-pass filter coefficients satisfy

$$\sum_n h_n = \sqrt{2}, \quad \sum_n h_n^2 = 1, \quad \sum_n g_n = 0, \quad \sum_n g_n^2 = 1, \quad \sum_n h_n g_n = 0. \tag{3.2.5}$$

Now, by applying $V_{j-1} = W_j \oplus^\perp V_j$ recursively, we get that for any resolution level $J \geq 1$

$$V_0 = W_1 \oplus^\perp V_1 = W_1 \oplus^\perp (W_2 \oplus^\perp V_2) = W_1 \oplus^\perp W_2 \oplus^\perp \ldots \oplus^\perp (W_J \oplus^\perp V_J) \tag{3.2.6}$$

It follows that for any $f \in V_0$ and $J \geq 1$, we can decompose $f = d_1 + d_2 + \ldots + d_J + a_J$, where $d_j = \mathcal{P}_{W_j} f \in W_j$ is the detail obtained by the projection operator $\mathcal{P}_{W_j}$ on W_j and $a_J = \mathcal{P}_{V_J} f \in V_J$ is the approximation obtained by projection $\mathcal{P}_{V_J}$ on V_J:

$$d_j(t) = \mathcal{P}_{W_j} f(t) = \sum_{n\in\mathbb{Z}} < f, \psi_{j,n} > \psi_{j,n}(t), \qquad a_J(t) = \mathcal{P}_{V_J} f(t) = \sum_{n\in\mathbb{Z}} < f, \varphi_{J,n} > \varphi_{J,n}(t). \tag{3.2.7}$$

When choosing a compactly-supported wavelet (like Daubechies), the coefficients $< f, \psi_{j,n} >$ (same for $< f, \varphi_{J,n} >$) are the Discrete Wavelet Transform (DWT) of f:

$$DWT_\psi(f)(j,n) = \langle f, \psi_{j,n} \rangle = \int_{-\infty}^{+\infty} f(t)\, \overline{\psi_{j,n}}(t)\, dt \tag{3.2.8}$$

If we put $D_j f(t) := 2^{-j} f(2^{-j} t)$, then $\psi_{j,n}(t) = 2^{j/2} D_j \psi(t - 2^j n)$. So $d_j(t) = 2^j \sum_{n\in\mathbb{Z}} < f(u), D_j\psi(u - 2^j n) > D_j\psi(t - 2^j n)$, where the DWT coefficients can be rewritten

$$< f(u), D_j\psi(u - 2^j n) >= \int_{-\infty}^{\infty} f(u) D_j \psi(u - 2^j n) du = (f * D_j\psi(-u))(\zeta)\Big|_{\zeta = 2^j n} \tag{3.2.9}$$

i.e. with engineer's language, the detail operator $\mathcal{P}_{W_j}$ can be expressed as a convolution (or high-pass filter) followed by a downsampling. Similarly, the approximation operator $\mathcal{P}_{V_J}$ can be expressed as a convolution (or low-pass filter) followed by a downsampling. Fig. 10 illustrates this idea.

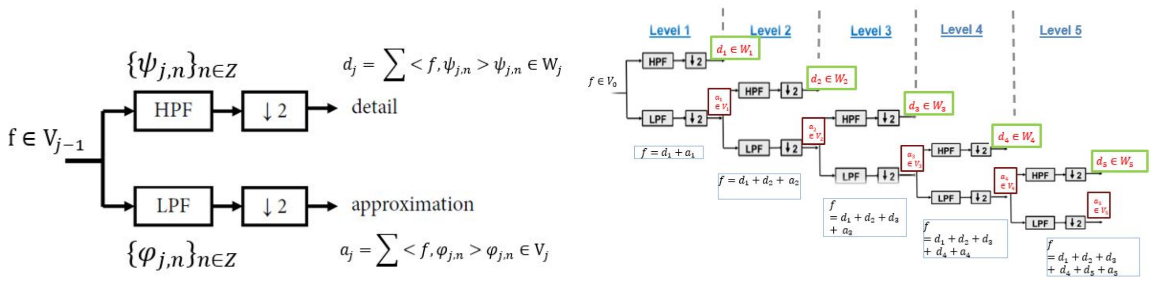

$\{\psi_{j,n}\}_{n\in Z}$
$d_j = \sum < f, \psi_{j,n} > \psi_{j,n} \in W_j$
$f \in V_{j-1}$
HPF
↓2
detail
LPF
↓2
approximation
$\{\varphi_{j,n}\}_{n\in Z}$
$a_j = \sum < f, \varphi_{j,n} > \varphi_{j,n} \in V_j$
Level 1
Level 2
Level 3
Level 4
Level 5
$f \in V_0$
$d_1 \in W_1$
$d_2 \in W_2$
$d_3 \in W_3$
$d_4 \in W_4$
$d_5 \in W_5$
$a_1 \in V_1$
$a_2 \in V_2$
$a_3 \in V_3$
$a_4 \in V_4$
$a_5 \in V_5$
$f = d_1 + a_1$
$f = d_1 + d_2 + a_2$
$f = d_1 + d_2 + d_3 + a_3$
$f = d_1 + d_2 + d_3 + d_4 + a_4$
$f = d_1 + d_2 + d_3 + d_4 + d_5 + a_5$

Figure 10: Left: Approximation and detail operators can be represented by filters, low-pass and high pass, respectively, followed by a dyadic downsampling. Right: iterated decomposition at Level 5: $f \approx d_1 + a_1 = d_1 + (d_2 + a_2) = \ldots = d_1 + d_2 + d_3 + d_4 + (d_5 + a_5)$.

In MATLAB we can choose among many discrete wavelets. Each one has specific properties which make is suitable for specific applications. The most important properties we may take care of are: the number of vanishing moments (⇒ sparsity of coefficients), symmetry (⇒ linear phase), and orthogonality (⇒ energy preserving). For instance, the wavelet families in the following list are all compactly supported, orthogonal and widely used in practical analysis:

- *Haar*: compactly supported wavelet, symmetric and orthogonal, but not continuous (the oldest and the simplest wavelet). Application: edge detection, feature extraction. Example: `haar, db1`
- *Daubechies*: compactly supported wavelets, with extremal phase and highest number of vanishing moments for a given support width. Quite asymmetric (non linear phase). Associated scaling filters are minimum-phase filters. Application: image denoising. Examples: `db2, db3, db4, ..., db45`.
- *Symlet*: compactly supported wavelets, built to be as nearly symmetric as possible, and with the highest number of vanishing moments for a given support width. Associated scaling filters are near linear-phase filters. Application: ECG. Examples: `sym2, sym3, sym4, sym5, ....`
- *Coiflet*: compactly supported wavelets, symmetric, with highest number of vanishing moments for both ϕ and ψ for a given support width. Examples: `coif2, coif3, coif4, coif5`.
- *Biorthogonal*: compactly supported, biorthogonal spline wavelets for which symmetry and exact reconstruction are possible with FIR filters (in orthogonal case it is impossible except for Haar). Application: image compression and reconstruction. Examples: `bior3.5, bior3.7, bior3.9, bior4.4, bior6.8, ....`

To get a list of all the wavelet families available in MATLAB, we can type ≫ `waveinfo` or ≫ `wavemngr('read',[])`. For a particular wavelet family, i.e. `wnamefam = 'db'`, and a particular wavelet, i.e. `wname = 'db4'`, we can use the following commands:

- `waveinfo(wnamefam)` to get detailed information about a wavelet family;
- $[\varphi, \psi, xi]$ `= wavefun(wname,iter)` to compute scaling φ and mother wavelet ψ;
- `[LoD,HiD] = wfilters(wname)` to compute the Lowpass and Highpass filter coefficients for Decomposition;
- `[LoD,HiD,LoR,HiR] = wfilters(wname)` to compute lowpass and highpass filter for Reconstruction as well.

The following code visualize father φ, mother ψ, high-pass and low-pass filters for 3 Daubechies wavelets `db2, db4, db16`:

Listing 11: Wavelets: comparison of Daubechies wavelets `db2, db4` and `db16`

```
waveinfo('db')                              % Daubechies family (include 45 wavelets)
figure, wname = ["db2", "db4", "db16"];   % compare some Daubechies wavelets
Nw = length(wname);
for i = 1:Nw
  [phival,psival,tval] = wavefun(wname(i),10);   % compute phi and  psi
  [Low,High] = wfilters(wname(i));               % compute filters coefficients

  subplot(3,Nw,i), hold on, plot(tval, phival, "r")  % plot scaling phi
  plot(tval, psival, "b")                            % plot mother psi
  title(wname(i)), legend("\phi","\psi")
  subplot(3,Nw,Nw+i), stem(High,"b")             % plot highpass filter
  subplot(3,Nw,2*Nw+i), stem(Low,"r")            % plot lowpass filter
end
```

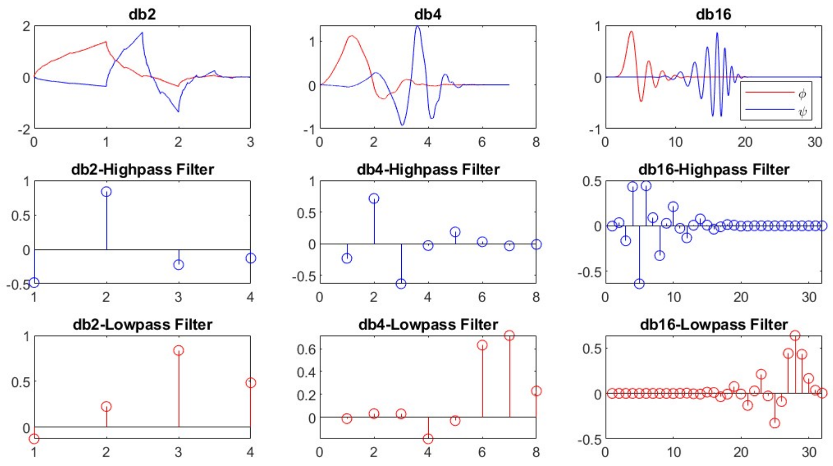


Figure 11: Daubechies wavelets `dbN`, with N vanishing moments and filter length of $2N$: compare `db2`, `db4`, `d16` (from left to right). Top row: the scaling ϕ (in red) and mother wavelet ψ (in blue). Central row: high-pass filter coefficients. Bottom row: low-pass filter coefficients.

Daubechies orthogonal wavelets `dbN` are widely used because they also have compact support, N vanishing moments, i.e. $\int x^n \psi_{dbN}(t)dt = 0$ for $n = 0, 1, 2, ..., N$, and filter length $2N$, i.e only finitely many h_n, $n = 0, 1, ..., (2N-1)$ are nonzero. Moreover, $g_n = (-1)^{(n+1)} h_{(2N-1)-n}$, $n = 0, 1, 2, 3$, with h_n and g_n defined in (3.2.4) and (3.2.3). For example, `db2` has 2 vanishing moments, filter length 4 and the low-pass filter coefficients h_n, $n = 0, 1, 2, 3$ are exactly:

$$h_0 = \frac{1-\sqrt{3}}{4\sqrt{2}} \quad h_1 = \frac{3-\sqrt{3}}{4\sqrt{2}} \quad h_2 = \frac{3+\sqrt{3}}{4\sqrt{2}} \quad h_3 = \frac{1+\sqrt{3}}{4\sqrt{2}} \tag{3.2.10}$$

while $g_0 = -h_3, g_1 = h_2, g_2 = -h_1, g_3 = h_0$. The values returned by `wfilters` command allow to verify the theory and all the relations (3.2.5):

Listing 12: Wavelets: low-pass and high-pass filters for Daubechies `db2`

```
>> [h,g] = wfilters("db2")
h =
     -0.1294     0.2241     0.8365     0.4830     % h0, h1, h2, h3
g =
     -0.4830     0.8365    -0.2241    -0.1294     % g0=-h3, g1 = h2, g2 = -h1, g3=h0

>> [sum(h)  sum(h.^2)  sum(g)  sum(g.^2)   h*g']
ans =
       1.4142     1.0000    -0.0000     1.0000          0
```

In MATLAB we can verify (3.2.6) and compute approximation $A_J \in V_J$ and detail components $D_j \in W_j$, $j = 1, 2, \ldots J$, of 1D signals x in many ways:

- DWT and inverse DWT (at a single level):
  - `[A,D] = dwt(x,waveletname)`
  - `x = idwt(A,D,waveletname)`
- Decomposition at level J: `[`C_J, L_J`] = wavedec(x,J,waveletname)`
- Reconstruction at any level $j = 1, 2, \ldots, J$:
  - A_j `= wrcoef('a',`C_J, L_J`,waveletname,j)`
  - D_j `= wrcoef('d',`C_J, L_J`,waveletname,j)`

where L_J=`[length(A), length(`D_J`), ..., length(`D_1`), length(x)]` is the bookkeeping vector (used to parse decomposition vector C_J). See also `appcoef` and `detcoef`. Similarly, we have functions for 2D images: `dwt2`, `idwt2`, `wavedec2`, `wrcoef2`, `appcoef2`, `detcoef2`.

Listing 13: Wavelets: approximation and detail decomposition with `wrcoef`

```matlab
load sumsin, ydata = sumsin'; J = 5;
figure, plot(ydata)
[C,L] = wavedec(ydata, J, 'db4');      % J-level decomposition using db4
figure
A = zeros(numel(ydata), J);
D = zeros(numel(ydata), J);
for j = 1:J
    A(:,j) = wrcoef('a', C, L,'db4',j);    % approximation Aj in Vj
    D(:,j) = wrcoef('d', C, L,'db4',j);    % detail Dj in Wj
    subplot(J,2,2*j-1), plot(A(:,j))
    ylim([-3 3]), title("A_"+j+"\in V_"+j)
    subplot(J,2,2*j), plot(D(:,j))
    ylim([-3 3]), title("D_"+j+"\in W_"+j)
end
% Check that Aj = A(j+1)+D(j+1) as Vn=V(n+1)+W(n+1)
max(abs(ydata - (A(:,1) + D(:,1))))     % f = A1 + D1  (err=7.35e-12)
max(abs(A(:,1) - (A(:,2) + D(:,2))))    % A1 = A2 + D2 (err=4.44e-16)
max(abs(A(:,2) - (A(:,3) + D(:,3))))    % A2 = A3 + D3 (err=8.88e-16)
max(abs(A(:,3) - (A(:,4) + D(:,4))))    % A3 = A4 + D4 (err=1.11e-15)
max(abs(A(:,4) - (A(:,5) + D(:,5))))    % A4 = A5 + D5 (err=1.33e-15)

J = 5;  % check that ydata = AJ+(D1+...+DJ) any J =1,2,...,5
max(abs(ydata - (A(:,J) + sum(D(:,1:J),2)))) % (err=7.35e-12)

% Optional: Extract the coarse scale approximation
A5 = appcoef(C,L,'db4');
% Extract only one detail signal at a specific level
D5 = detcoef(C,L,5);
% Extract all detail signals
[D1,D2,D3,D4,D5] = detcoef(C,L,1:5);
```

We can also compute *Maximal Overlap DWT* (also known as undecimated DWT) and its multiresolution analysis by `modwt` and `modwtmra` commands. Both use `sym4` wavelet by default. Moreover:

- `E = modwt(x,wname,J)` is energy-preserving, i.e. `modwt` partitions a signal's energy across detail and scaling coefficients.
- `M = modwtmra(E,wname)` accepts the matrix E returned by `modwt` as input and returns a matrix M with $J+1$ rows. The j^{th} row of M is the projection $d_j \in W_j$ of the signal onto the detail subspace W_j, for $j = 1, 2, \ldots, J$. The last row $M(J+1,:)$ corresponds to the approximation $a_J \in V_J$. This means that the original signal can be recovered by a column-wise sum of all the projections stored on the rows.

Listing 14: Wavelets: Multiresolution Analysis with `modwt` and `modwtmra`

```matlab
load wecg.mat wecg, ydata = wecg;     % load ECG signal, sampling frequency 180 Hz
tdata = (0:numel(ydata)-1)/180;       % (1xN) samples, with N = 2048
% DWT and MRA
wname = 'db4'; J = 6;                 % choose Daubechies wavelet db4 and J=6
E = modwt(ydata, wname, J);           % compute Maximal Overlap DWT
mraMatrix = modwtmra(E, wname);       % compute MRA (J+1) x N matrix
ydata_mra = sum(mraMatrix);           % sum down along columns, getting (1 x N) row

% Visualize original and reconstructed signals
figure,  plot(tdata,ydata,'b--.', LineWidth=1, MarkerSize=8);
hold on, plot(tdata,ydata_mra,'r', LineWidth=1);
```

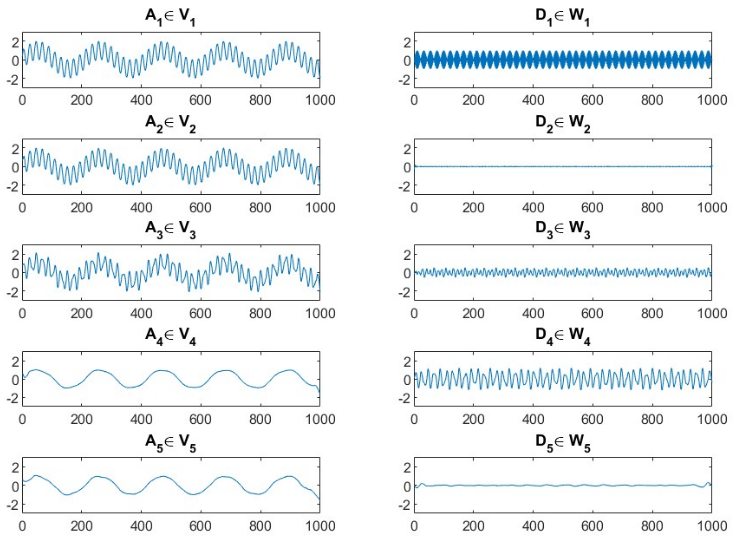


Figure 12: Multiresolution of a signal $f \in V_0$ at Level $J = 5$. See (3.2.6). First row: The signal f is decomposed as $f = A_1 + D_1$. Second row: $A1$ is decomposed as $A_1 = A_2 + D_2$. Third row: $A2$ is decomposed as $A_2 = A_3 + D_3$, and so on. Hence, the original signal is decomposed into $f = D_1 + D_2 + ... + D_5 + A_5$, with $D_j \in W_j$ and $a_J \in V_J$.

```
% Visualize decomposition
alldata = [tdata', mraMatrix'];       % transpose and concatenate to get N x (J+2)
T = array2table(alldata, VariableNames=["time","D"+Level:-1:1,"Approx"]);
figure, stackedplot(T, "XVariable", 1) % plot the approximation and N details
```

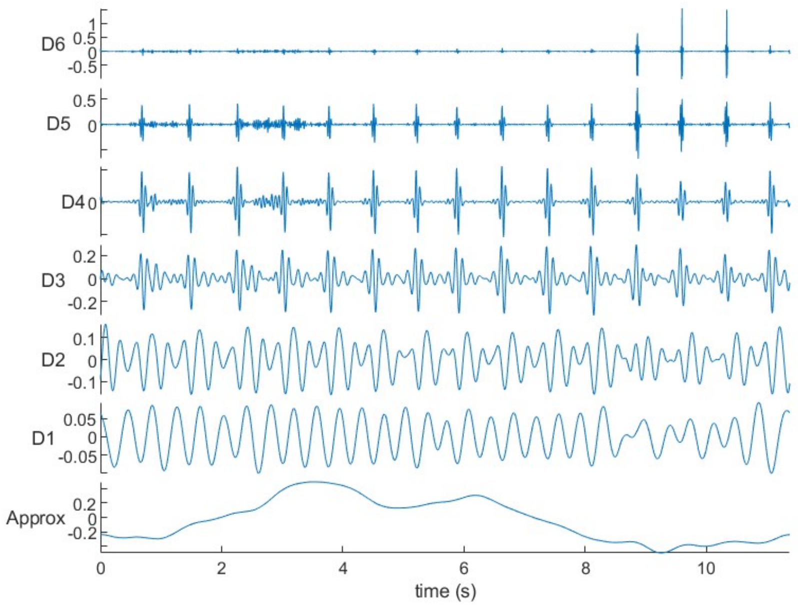
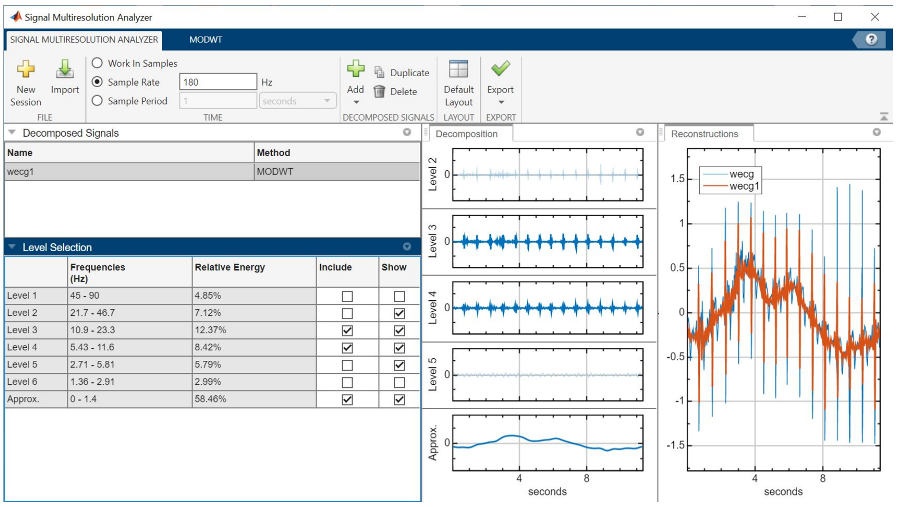


Figure 13: Multiresolution of a signal at Level $J = 6$, see (3.2.6) using `modwt` and `modwtmra` with `db4` wavelet.

We can perform MRA also by using the interactive App called *Signal Multiresolution Analyzer* (from the App tab of MATLAB desktop or by typing » `signalMultiresolutionAnalyzer`). We can quickly import a signal, choose a wavelet, pick the approximation level, and visualize the decomposition. For each level, the app allows to explore the energy percentage for each component and select which levels we would like to include (typically those with highest energy content) to reconstruct the signal. The App allows also to export the reconstructed signal, the decomposition matrix or automatically generate the MATLAB script with the commands. In the example in Fig. 14 we can see that, after a decomposition at level 6, the top three levels with the highest relative energy are:

- approximation (projection on space V_6) with the highest energy 58.46%
- detail at level 3 (projection on space W_3) with energy 12.37%
- detail at level 4 (projection on space W_4) with energy 8.42%.


Signal Multiresolution Analyzer

SIGNAL MULTIRESOLUTION ANALYZER | MODWT

( ) Work In Samples
[✓ Sample Rate] 180 Hz
( ) Sample Period 1 seconds

Decomposed Signals

Name	Method
wecg1	MODWT

Level Selection

	Frequencies (Hz)	Relative Energy	Include	Show
Level 1	45 - 90	4.85%	☐	☐
Level 2	21.7 - 46.7	7.12%	☐	☑
Level 3	10.9 - 23.3	12.37%	☑	☑
Level 4	5.43 - 11.6	8.42%	☑	☑
Level 5	2.71 - 5.81	5.79%	☐	☑
Level 6	1.36 - 2.91	2.99%	☐	☐
Approx.	0 - 1.4	58.46%	☑	☑

Figure 14: Signal Multiresolution Analyzer App in MATLAB: decomposition of an ECG signal (with frequency 150Hz) has been computed at level 6 with db4 wavelet; we can see corresponding energy for each level on the left panel. We can select the levels with the highest energy, i.e. the approximation + detail at level 3 + detail at level 4 , and observe the reconstruction by including only these terms on the right plot.

4 Radial Basis Functions

Radial basis functions (RBF) are a means to approximate multivariable functions that are too hard or too time consuming to be evaluated, so they can be sampled (or measured) only in a finite number of points. Since RBFs are radially symmetric functions that are shifted by points in a multidimensional normed linear space, they form a convenient data-dependent approximation space (as suggested by the Mairhuber-Curtis theorem). RBFs have many advantages: scalable to any dimension, high accuracy, fast convergence, no need for triangulation. RBFs are also used in machine learning, surrogate optimization and neural networks, as we will briefly see in the following sections.

4.1 Multivariate Scattered interpolation

We consider the problem of interpolating *scattered data* $X = \{x_k\}_{k=1}^N$ and $F = \{f_k\}_{k=1}^N$, with $x_k \in \mathbb{R}^d$ and $f_k \in \mathbb{R}$. We assume:

- x_k are multivariate ($d \geq 1$), pairwise distinct, and meshless (i.e. they don't need to form a regular grid)
- $f_k = f(x_k)$ are the only known (or sampled) values for some unknown (or too complex) function f
- We look for a simpler model P_f interpolating f, i.e. satisfying N conditions $P_f(x_k) = f_k$ $k = 1, 2, ..., N$.

The fundamental model for P_f is a linear combination of *radial basis functions*, dependent on the distance from each x_k:

$$P_f(x) = \sum_{k=1}^{N} \lambda_k \, \phi(||x - x_k||), \qquad P_f(x_i) = f_i \quad \forall i = 1, 2, ..., N \tag{4.1.1}$$

where $||\cdot - x_k||$ is any norm-induced distance (Euclidean, Minkowski, Mahalanobis, etc.) and $\phi : [0, \infty) \to \mathbb{R}$ is a radially symmetric function, chosen among:

$$\begin{aligned}
\phi(r) &= r && \text{(linear)}\\
\phi(r) &= r^3 && \text{(cubic)}\\
\phi(r) &= r^2 \log(r) && \text{(thin-plate spline)}\\
\phi(r) &= \sqrt{1 + (\epsilon r)^2} && \text{(multiquadric)}\\
\phi(r) &= e^{-(\epsilon r)^2} && \text{(Gaussian)}
\end{aligned} \tag{4.1.2}$$

Data are shaped such that X is Nxd matrix and F is a $Nx1$ column. If D_X is the dxd symmetric square matrix of all pairwise distances, i.e.

$$D_X = \begin{pmatrix} 0 & ||x_1 - x_2|| & ||x_1 - x_3|| & \cdots & \cdots & \cdots & ||x_1 - x_N|| \\ ||x_2 - x_1|| & 0 & ||x_2 - x_3|| & \cdots & \cdots & \cdots & ||x_2 - x_N|| \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ ||x_N - x_1|| & ||x_N - x_2|| & ||x_N - x_3|| & \cdots & \cdots & \cdots & 0 \end{pmatrix} \tag{4.1.3}$$

the problem (4.1.1) becomes equivalent to the linear system

$$\phi(D_X)\,\lambda = F \tag{4.1.4}$$

where ϕ is applied to any entry of D_X. The linear system (4.1.4) can be efficiently solved in MATLAB.

To implement this method in MATLAB, we consider the following scattered data in the unit square $[0,1]x[0,1]$:

- x_k: Quasi-random (low-discrepancy) 2D sequence, i.e. a Halton sequence
- f_k: Franke's bivariate test function (implemented in MATLAB with 2 input arguments)

Listing 15: RBF: Halton quasi-random dataset for testing

```
d = 2;                                  % d=space dimension
N = (2^4+1)^2;                          % N=number of scattered points in R^d
%  Generate quasi-random Halton set (see MATLAB Doc for Skip and Leap properties)
p = haltonset(d,'Skip',1e3,'Leap',1e2); % define quasi-random object in d-dim
p = p.scramble('RR2');                  % use scramble method (reverse-radix)
X = p.net(N);                           % use net method to generate N points in R^d
F = franke(X(:,1), X(:,2));             % compute Franke's test function
scatter3(X(:,1),X(:,2),F), view([70 35])
```

We can now create the RBF interpolator in MATLAB with three simple steps:

1. choose a distance and compute pairwise distance matrix D_X (4.1.3) by using `pdist` command;
2. choose an RBF ϕ (4.1.2) and apply it to each entry of the distance matrix;
3. solve the linear system (4.1.4) by using *backslash* operator (Gauss direct methods):

$$\lambda = \phi(D_X) \setminus F \tag{4.1.5}$$

Listing 16: RBF: Build the RBF interpolator

```
dist_name = 'Euclidean';           % define a distance
phi = @(r) r.^3;                   % define an RBF
D = pdist(X, dist_name);           % 1) compute pairwise distances
A = squareform(phi(D));            % 2) apply RBF to distances
lambda = A\F                       % 3) solve linear system A*lambda = F
```

To visualize the interpolating surface, we need to prepare uniform mesh for each dimension, stretch them into columns and compute pairwise distances between these uniform points and the original scattered ones by using `pdist2` command:

Listing 17: Visualization of the RBF interpolator

```
n = 30;                                  % mesh size
gridx = linspace(0,1,n);
gridy = linspace(0,1,n);
[xe,ye] = meshgrid(gridx, gridy);          % d uniform meshes (nxn)
eX = [xe(:), ye(:)];                       % stretch into n^2 x d
eDist = pdist2(eX,X, dist_name);           % pairwise distances to get n^2 x N
eA = phi(eDist);                           % apply RBF
f_interp = eA*lambda;                      % multiply by lambda (Nx1) to get n^2 x 1
f_interp_mat = reshape(f_interp,n,n);      % reshape into nxn
figure, hold on
scatter3(X(:,1),X(:,2),F, "b", "filled")
surf(xe,ye,f_interp_mat), view([70 35])
shading interp, colormap hot, alpha(0.5)
```

Another useful model is the scattered interpolator with *polynomial precision*, able to exactly reproduce a linear interpolator when the original data are already linear. It can be obtained from (4.1.1) by adding polynomial terms to the RBFs. For simplicity let us formulate the model in $\mathbb{R}^2$ and add a polynomial of degree 1 in 2 variables:

$$P_f(x) = \sum_{k=1}^{N} \lambda_k\, \phi\,(||x - x_k||) + \underbrace{C_1 + C_2 x^{(1)} + C_3\, x^{(2)}}_{\text{polynomial precision}}\,, \qquad x = \left(x^{(1)}, x^{(2)}\right) \in \mathbb{R}^2 \tag{4.1.6}$$

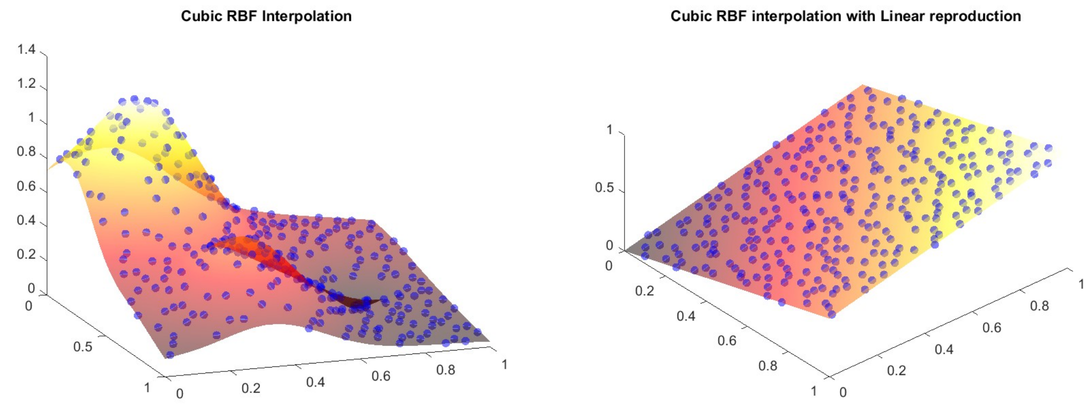


Figure 15: Scattered data interpolation: on the left: with cubic radial basis function for Franke's test function; on the right: with cubic radial basis function and linear polynomial precision for some linear data

As we have 3 more unknowns, we can add 3 more conditions. For details, see [9]. If we put $P = \begin{bmatrix} 1 & x^{(1)} & x^{(2)}, \end{bmatrix}$ we obtain an augmented linear system, similar to (4.1.4):

$$\begin{pmatrix} \phi(D_x)_{NxN} & P_{Nx3} \\ (P^T)_{3xN} & 0_{3x3} \end{pmatrix} \begin{pmatrix} \lambda_{Nx1} \\ C_{3x1} \end{pmatrix} = \begin{pmatrix} F_{Nx1} \\ 0_{3x1} \end{pmatrix} \tag{4.1.7}$$

MATLAB code for solving the augmented linear system (4.1.7) is straightforward. See Fig. (15) (on the right).

Listing 18: RBF: Cubic RBF interpolator with linear polynomial precision

```
D = squareform(pdist(X, "euclidean"));  % compute distance matrix
phi = @(r) r.^3;                        % define cubic RBF
A = phi(D);                             % apply RBF to Distance matrix
P = [ones(N,1), X];                     % define P=[1 xi yi ] as N x 3
A0 = [A P; P' zeros(3)];                % define augmented interpolation matrix
F0 = [F; zeros(3,1)];                   % add zeros to F
lambda0 = A0\F0;                        % solve linear system A0*lambda0 = F0
```

4.2 Surrogate Optimization

A *surrogate* is a simpler function that approximates another function and takes less time to evaluate. RBF interpolators can be used to generate a surrogate of the objective function in a constrained optimization problem. *Surrogate optimization* is best-suited when we have a time-consuming objective function (i.e. expensive to evaluate, nonsmooth or kind of black-box with no analytical input/output expression): to search for a point that minimizes the objective function, simply evaluate its surrogate on thousands of points randomly chosen in a limited box, and take the best value as an approximation to the minimizer of the objective function.

Surrogate optimization attempts to find a *global minimum* of the constrained optimization problem:

$$x^* = \operatorname*{argmin}_{x\in\Omega\subseteq\mathbb{R}^n} f(x) \;\; \text{such that} \begin{cases} \text{lb} \le x \le \text{ub} \\ x_i \text{ integer for some } i \\ \text{A}x \le b \\ A_{\text{eq}}\, x = b_{\text{eq}} \\ c(x) \le 0 \end{cases} \tag{4.2.1}$$

where $x \in \Omega \subset \mathbb{R}^n$ (optionally, some x_i's can be assumed to be integer); lb and ub are lower and upper bounds, respectively; $Ax \le b$ and $A_{eq} = b_{eq}$ are linear inequalities and equalities; $c(x) \le 0$ are non-linear constraints.

In MATLAB, `surrogateopt` function implements a derivative-free algorithm to solve (4.2.1) by approximating the objective $f(x)$ with a surrogate $s(x)$ given by a *cubic RBF interpolator with linear tail* as described in Section 4.1 (which also minimizes a measure of *bumpiness* as defined by H.M. Gutmann in [11]):

$$f(x) \approx s(x) = \sum_{k=1}^{N} \lambda_k\, ||x - x_k||^3 \; + (cx + d) \tag{4.2.2}$$

The *merit function* is a convex combination of two terms: a scaled surrogate and a scaled distance

$$\mu_\omega(x) := \omega \frac{s(x) - s_{min}}{s_{max} - s_{min}} + (1-\omega)\frac{d_{max} - d(x)}{d_{max} - d_{min}} \tag{4.2.3}$$

where $d(x)$ is the minimum distance of the point x from trial points. A small value of ω looks at points that are far from evaluated points, leading the search to new regions. A large value of ω gives importance to the surrogate values, causing the search to minimize the surrogate. Typically, ω cycles through the values: 0.3, 0.5, 0.8, and 0.95 (see MATLAB Documentation [19] for details). The surrogate optimization algorithm alternates between two phases (see Fig. 16):

- Phase A: Construct Surrogate
  1. Sample within the bounds by generating a small amount of quasi-random sample points and evaluate the objective function at these trial points (`MinSurrogatePoints`, default is $\max(20, 2n)$).
  2. Create a surrogate model of the objective function by using an RBF interpolator (4.2.2) through these trial points. Identify the best point (found since the last surrogate reset). This point is called *incumbent point*
- Phase B: Search for Minimum
  3. Sample with a large number of pseudorandom points (about $10^2 - 10^3$) near the *incumbent point*. Evaluate the merit function (4.2.3) at these trials, but not at any point within `MinSampleDistance` of a previously evaluated point (default is 10^{-6}). The point with the lowest merit function value is called the *adaptive point*.
  4. Evaluate the objective at the adaptive point, and update the surrogate based on this point and its value. If the objective function value at the adaptive point is sufficiently lower than the incumbent value, then the solver deems the search successful (`success=success+1`) and sets the adaptive point as the new incumbent. Otherwise, the solver deems the search unsuccessful (`failure = failure+1`) and does not change the incumbent.
  5. Update the dispersion of the sample distribution upwards if 3 successes occur before $\max(n, 5)$ failures. Update the dispersion downwards if $\max(n, 5)$ failures occur before 3 successes.
  6. Continue from step 3 until all trial points are within `MinSampleDistance` of the evaluated points. At that time, reset the surrogate by discarding all adaptive points from the surrogate, reset the scale, and go back to step 1 to create `MinSurrogatePoints` new random trial points for evaluation.

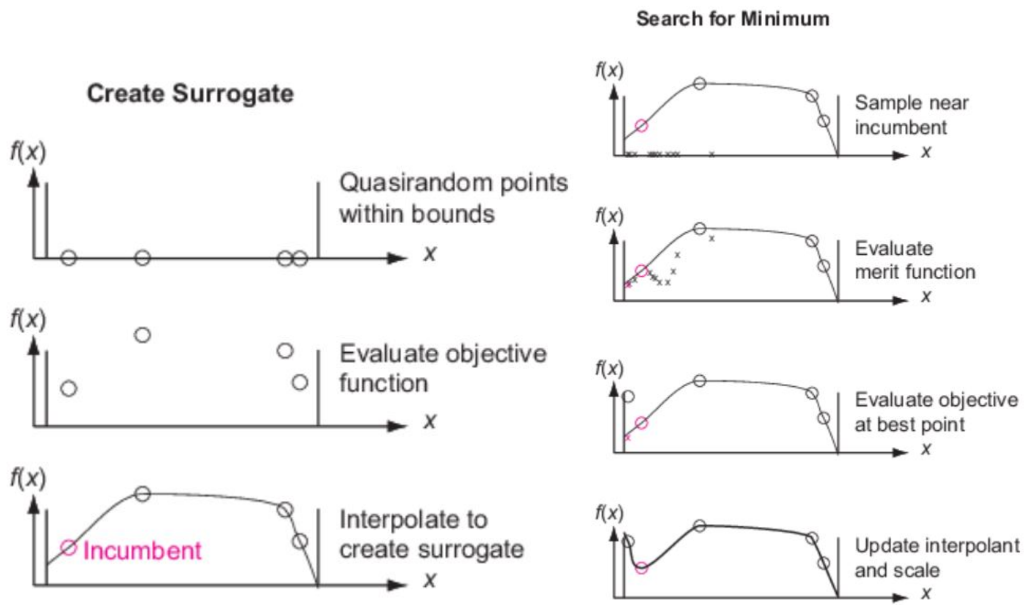

Search for Minimum
Create Surrogate
f(x)
Quasirandom points within bounds
x
Evaluate objective function
Interpolate to create surrogate
Incumbent
Sample near incumbent
Evaluate merit function
Evaluate objective at best point
Update interpolant and scale

Figure 16: The two phases of Surrogate Optimization algorithm: 1) Construct the surrogate; 2) Search for Minimum

In MATLAB, the interface to call `surrogateopt` is

```
[x, fval, exitflag] = surrogateopt(@objconstr,lb,ub,intcon,A,b,Aeq,beq,options),
```

where lb and ub are mandatory finite bound constraints for all variables and `@objconstr` is a function handle pointing at the objective function with signature: `function S = objconstr(x)`. The output S must be a structure array with 2 fields: `S.Fval` and `S.Ineq`, the objective value and the nonlinear inequality constraints $c(x)$, respectively. Note that the non-linear constraints must be incorporated together with the objective function.

Solving a problem with surrogate optimization. As an example, let us try to solve the following problem where the objective requires to solve a system of ordinary differential equations (ODE): change the position and angle of a bow to throw an arrow as far as possible beyond a fixed wall and staying below a ceiling. The decision variable is $x = (x_1, x_2) \in \mathbb{R}^2$, where x_1 = the initial distance of the bow from the wall, and x_2 = the initial angle of the arrow. Suppose we are given the following data:

- the wall is 20 m high. (Fix the origin of coordinate system on the base of the wall.);
- the ceiling is 60 m high;

- nonlinear air resistance slows the arrow proportionally to the square of velocity, with constant $\mu = 0.01$;
- gravity acts on the arrow, accelerating it downward with constant $g = 9.81\ m/s^2$;
- the arrow has an initial velocity of $v_0 = 85$ m/s;
- initial conditions (initial distance from wall, initial angle): $x^{(0)} = [-50; \pi/6]$;
- *bound constraints*: $-200 \leq x_1 \leq -1$ and $0.05 \leq x_2 \leq \frac{\pi}{2} - 0.05$;
- *wall constraint*: if the trajectory crosses the wall at a height less than 20, the trajectory is infeasible;
- *ceiling constraint*: if the trajectory crosses the ceiling at height 60, the trajectory is infeasible.

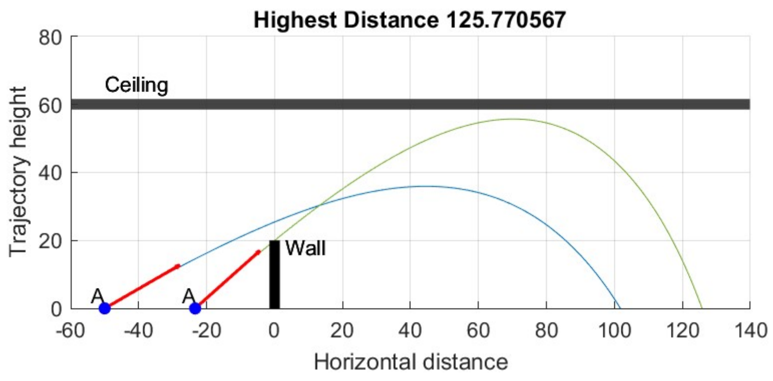


Figure 17: Surrogate Optimization problem: find position A and angle of a bow to throw an arrow as far as possible beyond a wall and staying below a ceiling. As a first guess, taking initial position $x_1^{(0)} = -50$ and initial angle $x_2^{(0)} = \pi/6$, we can achieve the distance 101.75. After optimization, getting a little closer $x_1^{(final)} = -23.38$ and increasing the initial angle $x_2^{(final)} = 0.73$, we will achieve the best distance 125.77

As ODE solvers require the model to be a first-order system, we define a 4-dim state column vector $q = (q_1, q_2, q_3, q_4)^T$, with (q_1, q_2) for the position in the plane and (q_3, q_4) for the velocities, so the dynamics of the arrow is given by:

$$\frac{d}{dt} q(t) = \begin{bmatrix} q_3(t) \\ q_4(t) \\ -\mu \left\| \left(q_3(t), q_4(t)\right) \right\| q_3(t) \\ -\mu \left\| \left(q_3(t), q_4(t)\right) \right\| q_4(t) - g \end{bmatrix} \tag{4.2.4}$$

To solve the problem in MATLAB, we define a structure array with all the problem parameters:

Listing 19: Surrogateopt: defining a structure for problem parameters

```
param.wallheight = 20;
param.peak = 60;
param.resistancecoeff = 0.01;
param.initialspeed = 85;
param.lb = [-200; 0.05];
param.ub = [-1; pi/2-0.05];
param.gravity = 9.81;
param.x0 = [-50; pi/6];
param.stoptime = 15;
```

We also define three functions:

- `ModelEquations`: a nested function defining the ODE system (4.2.4);
- `ModelSolver`: a function calling `ode45` solver to return the ODE solution;
- `ObjectiveConstr`: the main function for surrogate optimization returning a structure S with two special fields:
  - $S.Fval$: returns the negative distance to be minimized;
  - $S.Ineq$: returns the two constraints (height ≥ WALL and peak ≤ CEILING).

Listing 20: Surrogateopt: ODE solver calling a nested function

```
function sol = ModelSolver(x,param)
% x(1) = distance from wall, x(2) = angle. Change initial 2-D point x to 4-D q0
q0 = [x(1);0; param.initialspeed*cos(x(2)); param.initialspeed*sin(x(2))];
% Solve ODE
sol = ode45(@ModelEquations,[0,param.stoptime],q0);

```

```
    function dq = ModelEquations(~,q)  % NESTED function
        dq = zeros(4,1);               % initialize state q as 4D
        v = norm(q(3:4));              % norm of the velocity
        dq(1) = q(3);
        dq(2) = q(4);
        dq(3) = -param.resistancecoeff *v*q(3);                 % horizontal accel
        dq(4) = -param.resistancecoeff *v*q(4) - param.gravity; % vertical accel
    end
end
```

Note that:

- `ModelEquations` is a nested function in `ModelSolver`, so that it shares the `param` structure from the parent workspace. Because we are using nested functions, all functions must be terminated by `end`;
- `ModelSolver` is responsible to convert 2D input x into 4D q as required by `ModelEquations`;
- `ObjectiveConstr` calls `fzero` twice to determine both the achieved distance when the trajectory height is zero and the achieved height when the arrow passes over the wall. To find the peak of trajectory, it calls `fminbnd`.

Listing 21: Surrogateopt: objective function (distance) including the nonlinear constraints (wall and ceiling)

```
function S = ObjectiveConstr(x, param)
sol = ModelSolver(x,param);
horiz_pos = @(t) deval(sol,t,1);
vert_pos = @(t) deval(sol,t,2);

% Find time t when trajectory height = 0 and horizontal position at that time
t0_height0 = fzero(vert_pos,[1e-2,param.stoptime]);
dist = horiz_pos(t0_height0);

% What is the height when the arrow crosses the wall at x = 0?
if horiz_pos(15) > 0
    t0_wall = fzero(horiz_pos,[0,param.stoptime]);
    height = vert_pos(t0_wall);
else
    height = vert_pos(param.stoptime);
end

% What the maximum height achieved?
t0_peak = fminbnd(@(t) -vert_pos(t), 1e-2,param.stoptime, optimset(TolX=1e-8));
peak = vert_pos(t0_peak);

% define output structure
S.Fval = -dist;                         % Objective: negative of distance
S.Ineq(1) = param.wallheight - height; % height >= WALL, ie WALL-height <= 0
S.Ineq(2) = peak - param.peak;         % peak <= CEILING, ie peak-CEILING<=0
end
```

Finally, we are ready to run the surrogate optimization:

- we change interface of `@(x) ObjectiveConstr(x,param)` by embedding the `param` structure, and obtaining a function depending only on x, which is exactly what `surrogateopt` needs;
- `PlotFcn='surrogateoptplot'` and `Display='iter'` are useful training options to visualize how the algorithm is progressing;
- the option `UseParallel=true` allows to start a parallel pool of workers able to maintain a queue of points on which to evaluate the objective function; a scheduler takes points from the queue in a FIFO fashion and assigns them to workers as they become idle, asynchronously.

More options can be explored with » `optimoptions("surrogateopt")` or » `doc surrogateopt`.

Listing 22: Surrogateopt: defining options and running surrogateopt

```
new_interface = @(x) ObjectiveConstr(x,param);  % embed param to change interface

opts = optimoptions('surrogateopt', InitialPoints=param.x0,...
         PlotFcn='surrogateoptplot', Display='iter', ...
         MaxFunctionEvaluations=500, MinSurrogatePoints=30, MinSampleDistance=1e-8,...
         UseParallel=true);

[xsolution,distance,exitflag,output] = ...
         surrogateopt(new_interface,param.lb,param.ub,opts)
```

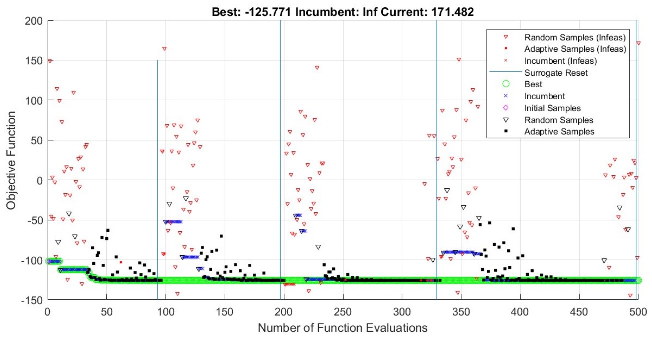


Figure 18: Surrogate Optimization algorithm: `PlotFcn='surrogateoptplot'` provided this graphic showing the two phases. 1) Construct the surrogate (4.2.2), with random sampling and incumbent points (the best of surrogate); 2) Search for global minimum, with random sampling around incumbent point and adaptive points (points with the lowest merit function (4.2.3) since last surrogate reset)

4.3 Kernelized Support Vector Machines

Support Vector Machines (SVM) are versatile machine learning models proposed by V. Vapnik, for both classification (1992) and regression (in 1996), both linear and nonlinear. They can be easily implemented in MATLAB with `fitcsvm` and `fitrsvm` functions (`fitc***` is for classification, `fitr***` is for regression).

Support Vector Classification (C-SVM) is an approximation of the decision boundary between two classes: the boundary not only separates the two classes, but also stays as far away from the closest training instances as possible. SVM-classifier fits the widest possible "street" separating two classes. This is called *large margin classification*. The decision boundary is fully determined by the instances located on the edge of the street, called *support vectors*. If we strictly impose that all instances be off the street and on the correct side (no misclassification at all), this is called *hard margin classification*. This would be ideal, but there are issues: it only works if the dataset is linearly separable, and it is quite sensitive to outliers. More realistically, we'd better look for a good balance between keeping the street as large as possible and limiting *margin violations* (i.e. instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

Given a dataset of N points with n features, $\{x^{(i)} \in \mathbb{R}^n\}_{i=1,2,\dots,N}$, assume data belong to one of two classes ("positive" or "negative") and that the two classes are linearly separable. Then we look for a linear boundary of the form $y = w \cdot x + b$, where $w \in \mathbb{R}^n$ is an unknown "slope" vector and $b \in \mathbb{R}$ is an intercept. We define $t^{(i)} = 1 \iff w \cdot x^{(i)} + b$ is "positive"; $t^{(i)} = -1 \iff w \cdot x^{(i)} + b$ is "negative". The *soft margin problem* is a quadratic programming (QP) problem, whose *primal problem* is:

$$\min_{w\in\mathbb{R}^n,\, b\in\mathbb{R},\, \zeta\in\mathbb{R}^N} \frac{1}{2} w' \cdot w + C \sum_{i=1}^{N} \zeta_i \tag{4.3.1}$$
$$\text{subject to } t^{(i)}(w \cdot x^{(i)} + b) \geq 1 - \zeta_i \text{ and } \zeta_i \geq 0 \text{ for } i = 1, 2, \dots, N$$

where ζ_i are N slack variables to measure how much i^{th} instance is allowed to violate the margin, C is a hyperparameter (Box Constraint) trading off between two conflicting objectives: making the slack variables as small as possible to reduce margin violations, and making the norm of w as small as possible to increase the margin. The *dual problem* is:

$$\min_{\alpha\in\mathbb{R}^N} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t^{(i)} t^{(j)} < x^{(i)}, x^{(j)} > - \sum_{i=1}^{N} \alpha_i \tag{4.3.2}$$
$$\text{subject to } \alpha_i \geq 0 \text{ for } i = 1, 2, \dots, N$$

where α_i are N Lagrange multipliers, $\alpha_i \neq 0 \iff x^{(i)}$ is a support vector. Remarkably, the dual problem depends on inner products of training data $< x^{(i)}, x^{(j)} >$.

Example 1. The magic of the Kernel Trick.
The *kernel trick* allows to extend linear SVM to non linearly separable data. RBF kernels play a key role. To see this, generate 200 non linearly separable random points in $\mathbb{R}^2$, i.e. 100 "red" points (class "-1") uniformly distributed in the unit disk, and 100 "blue" points (class "+1") in the ring of rays 1 and 2. These points are NOT linearly separable in $\mathbb{R}^2$ (see Fig. 19), but become linearly separable if embedded in higher dimension space $\mathbb{R}^3$ through a feature map like this:

$$\varphi : \mathbb{R}^2 \longrightarrow \mathbb{R}^3 : (x, y) \longmapsto \left(x^2, \sqrt{2}\, xy, y^2\right) \tag{4.3.3}$$

Listing 23: Generate non linearly separable random points in a disk

```
r = sqrt(rand(100,1));                % random radius
t = 2*pi*rand(100,1);                 % random angle
data1 = [r.*cos(t), r.*sin(t)];       % points of class -1 (RED)
```

```
r2 = sqrt(3*rand(100,1)+1);          % random radius for ring
t2 = 2*pi*rand(100,1);               % random angle for ring
data2 = [r2.*cos(t2), r2.*sin(t2)]; % points of class +1 (BLUE)
data3 = [data1;data2];               % concatenate data together
theclass = ones(200,1);              % assign class +/-1
theclass(1:100) = -1;
```

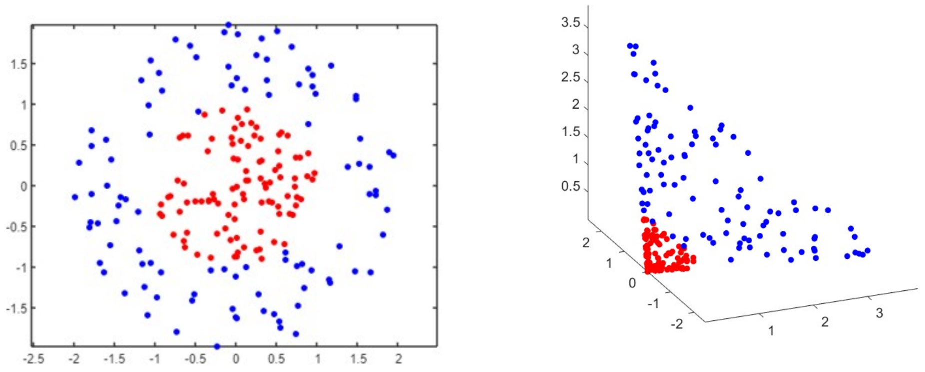


Figure 19: Non linearly separable random points in $\mathbb{R}^2$ become linearly separable in $\mathbb{R}^3$ by a feature map (4.3.3)

When applying φ to the training data, the dual problem (4.3.2) will contain $<\varphi(x^{(i)}),\varphi(x^{(j)})>$. In general it will be very difficult (even impossible) to know φ. The magic is that the inner product in the feature space boils down to a kernel function on the original space, so we don't need to know φ:

$$<\varphi(u),\varphi(v)>=K(u,v). \tag{4.3.4}$$

For example, the corresponding kernel for (4.3.3) satisfying (4.3.4) is the 2^{nd} degree polynomial $K(u,v)=(u'\cdot v)^2$. According to Mercer's theorem, for every kernel $K(u,v)$ there exists a feature map φ satisfying (4.3.4). With Gaussian RBF kernels, φ would actually map data in an ∞-dim Hilbert space. Next, we see how to apply the kernel trick in MATLAB.

Example 2. SVM Classification with RBF Kernel Trick and Box Constraint
`fitcsvm` first solves (4.3.2) to find Lagrange multipliers α, then will solve the primal (4.3.1) with respect to W and b. MATLAB uses the following solvers: SMO (Sequential Minimal Optimization), ISDA (Iterative Single Data Algorithm), L1QP (L1-quadprog). See »`doc fitrsvm` for details. The kernel function can be specified using the `KernelFunction` option:

- `'rbf'` or `'Gaussian'`: $K(u,v)=e^{-\|\frac{1}{\gamma}u-v\|^2)}$, with `KernelScale` parameter γ ($\gamma=1$ by default)
- `'linear'`: $K(u,v)=u'\cdot v$
- `'polynomial'`: $K(u,v)=(1+u'\cdot v)^q$, with `PolynomialOrder` parameter q ($q=3$ by default)
- `'myKernel'`, if we have our own kernel function with signature `K=myKernel(U,V)`

The output of `fitcsvm` is an object, let us say `mySVM`, belonging to the `ClassificationSVM` class. Objects are instances of a class, defined in the sense of OOP (Object Oriented Programming), containing properties and methods. All properties are accessible with the `dot` notation, i.e. `mySVM.Alpha`, `mySVM.W`, `mySVM.Solver`, `mySVM.IsSupportVector`, etc. We can also use the method `predict` to extract labels and scores of new data. Both syntaxes `predict(mySVM, —)` and `mySVM.predict(—)` are valid. Scores are useful to visualize the decision boundary as a contour plot.

Listing 24: SVM classification with RBF kernel

```
mySVM = fitcsvm(data3, theclass, ClassNames =[-1,1], ...
        KernelFunction ='rbf', KernelScale = 1, BoxConstraint = 1);
alpha = mySVM.Alpha             % returns just non null alpha of support vectors
supVec = data3(mySVM.IsSupportVector, :)            % Identify support vectors
% Predict scores over a regular meshgrid to visualize boundary
[x1Grid, x2Grid] = meshgrid(min(data3(:,1)):0.02:max(data3(:,1)),...
                            min(data3(:,2)):0.02:max(data3(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
[~, scores] = predict(mySVM,xGrid);
boundary = reshape(scores(:,2),size(x1Grid));
% Visualization of data, support vectors and decision boundary
figure, gscatter(data3(:,1),data3(:,2),theclass,'rb','.');
hold on, ezpolar(@(x) 1);
plot(supVec(:,1),supVec(:,2),'ko', 'MarkerSize', 10, 'LineWidth',1);
contour(x1Grid,x2Grid, boundary, [0 0], 'k', 'LineWidth',2);
```

`BoxConstraint` is a (positive) hyperparameter of `fitcsvm`, useful to balance between hard vs soft implementation. It works as a regularization knob to control the maximum penalty imposed on margin violations and prevent from overfitting:

- a smaller `BoxConstraint` means a wider street (soft-margin), but larger number of support vectors and more margin violations.
- a larger `BoxConstraint` means a narrower street, fewer support vectors and fewer margin violations (but possibly longer training times). As an extreme, `BoxConstraint=`∞ means hard margin classification.

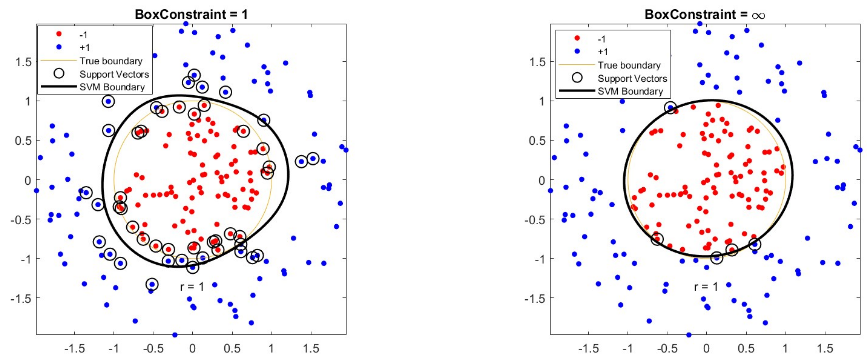


Figure 20: SVM classification with `KernelFunction='rbf'`. *Left*: Soft-margin with `BoxConstraint = 1`. *Right*: Hard-margin with `BoxConstraint=` ∞ .

Example 3: SVM Regression with RBF Kernel and custom kernel. Support Vector Regression (SVR) reverses the objective of SVM classification: instead of trying to fit the largest possible street (or margin) separating two classes while limiting margin violations, SVM regression tries to fit as many instances as possible ON the street, while limiting instances OFF the street. The width of the street is controlled by a hyperparameter $\epsilon > 0$ (the larger ϵ, the larger the street). Adding more training instances within the margin does not affect the model's predictions; thus the model is said *ϵ-insensitive*, also known as `L1` *loss*. SVR relies on kernel functions. Given a dataset $(x_i, y_i)_i$, the goal is to find a function $f(x)$ that deviates from y_i by a value not greater than ϵ for each training point x_i, and at the same time is as flat as possible.

In MATLAB, `fitrsvm` computes SVM regression using $\epsilon = iqr(y_i)/13.49$ by default. It supports kernel functions (`'rbf'` by default) and returns an object in the `RegressionSVM` class.

Listing 25: SVM Regression with RBF kernel

```
y = franke(data3(:,1), data3(:,2));
regSVM = fitrsvm(data3,y, 'KernelFunction','rbf', ...
                       'BoxConstraint',10, 'Epsilon',0.05, 'Standardize',true);

flag = regSVM.ConvergenceInfo.Converged     % Check convergence (1 = Converged)
num_supVec = numel(regSVM.Alpha)            % Number of support vectors (non null Alpha)
idx_supVec = regSVM.IsSupportVector;        % logical array to localize support vectors
```

To define a custom kernel, we first define a function with signature `K = mysigmoid(U,V)`, returning the Gram matrix (4.3.4), and then assign the function name to `KernelFunction` option. In the following example, a new kernel using the hyperbolic tangent is defined.

Listing 26: SVM Regression with custom kernel

```
regSVM_mykernel = fitrsvm(data3,y, 'KernelFunction',"mysigmoid", ...
               'BoxConstraint',1e4, 'Epsilon',0.04, 'Standardize',true);

function K = mysigmoid(U,V)     % custom kernel definition
        gamma = 0.5; c = -1;
        K = tanh(gamma*U*V' + c);     % U mxp, V = nxp, K = mxn
end
```

Finally, we may need to experiment a bit before we identify the values of `BoxConstraint`, `KernelScale` and `Epsilon` that ensure convergence and best approximation. So, we can consider to find hyperparameters that minimize cross-validation loss by using automatic *hyperparameter optimization*. For more details, look into the MATLAB Documentation searching for »`doc BayesianOptimization`.

Listing 27: SVM Regression with automatic hyperparameter optimization

```
opt = struct('AcquisitionFunctionName', 'expected-improvement-plus')
regSVM_Optim = fitrsvm(data3,y, 'KernelFunction', 'rbf', ...
    'OptimizeHyperparameters','auto', 'HyperparameterOptimizationOptions',opt);
bestHyperparam = bestPoint(regSVM_Optim.HyperparameterOptimizationResults)
```

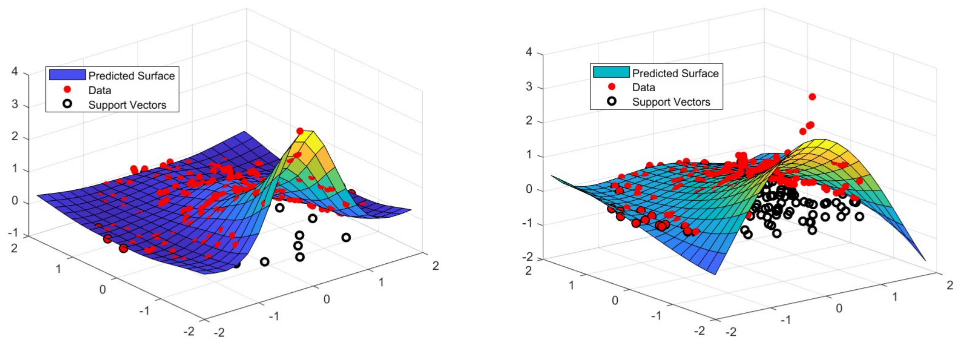


Figure 21: SVM Regression. *Left*: `KernelFunction='rbf', BoxConstrain=10, Epsilon=0.05`. *Right*: Custom Kernel function `BoxConstrain=1e6, Epsilon=0.04`

5 Neural Networks

In sessions 5.1 and 5.2 we explain neural networks as universal approximators to approximate any (continuous) map $f : \mathbb{R}^n \to \mathbb{R}^m$. In session 5.3 we see how a neural network can approximate a complex dynamic system for which analytic expressions between input and output may be hard to find or even may not exist. See [1] and [5] for an introduction to neural networks.

5.1 Universal Approximators

Neural networks can approximate at any accuracy large families of target functions. As proven in 1989 by Cybenko, Hornik, Stinchcombe, White (see [5]), any continuous function $f : X \subseteq \mathbb{R}^n \to Y \subseteq \mathbb{R}^m$ can be approximated by a neural network with a single arbitrary-width hidden layer with nonlinear activation function. We are going to introduce some terminology and then an example in MATLAB to demonstrate this result.

Neural networks contain unknown parameters θ that must be tuned to get the "best" approximation $\hat{f}_\theta$ of the unknown f via a *supervised learning process* which consists in:

- given a large dataset of input-output pairs $X_N = (x^{(i)})_{i=1,2,\ldots,N}$ and $Y_N = (y^{(i)})_{i=1,2,\ldots,N}$, drawn from a joint probability distribution $\mathbb{P}(X,Y)$
- given a *loss function* $L(y, \hat{f}_\theta(x))$ (root mean squared error for regression, or cross-entropy for classification)

determine the best θ^* to minimize the *expected loss* (known also as *generalization error* or risk):

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\ \mathbb{E}_X(\mathbb{E}_{Y|X}(L(Y, \hat{f}_\theta(X)))) = \underset{\theta}{\operatorname{argmin}} \int_X \int_{f(X)} L(y, \hat{f}_\theta(x)) \underbrace{\mathbb{P}(Y = y | X = x)}_{\text{posterior}} dy \underbrace{\mathbb{P}(X = x)}_{\text{marginal}} dx \tag{5.1.1}$$

The architecture of a feed-forward neural network can be defined as a concatenation of layers, each layer made of a number of artificial neurons characterized by learnable matrix parameters $\theta = (W, b)$ and a nonlinear activation function σ: if the i^{th} layer has p neurons and $(i+1)^{th}$ layer has q neurons, then the i^{th} layer is described by:

$$i^{th} \text{ layer}: \quad \eta^{(i)} = \sigma(W^{(i)}\xi^{(i)} + b^{(i)}), \qquad \xi^{(i)} \in \mathbb{R}^p; \quad \eta^{(i)} \in \mathbb{R}^q \tag{5.1.2}$$

where $W^{(i)}$ is a (qxp) weight matrix, $b^{(i)}$ is a $(qx1)$ bias , and $\sigma : \mathbb{R}^q \to \mathbb{R}^q$ is a scalar nonlinear activation function that can be applied component-wise.

In MATLAB, many `*Layer` commands are available to build a neural network from scratch:

- an `*InputLayer` must come first depending on input type, i.e.`featureInputLayer`,`imageInputLayer`, etc
- neuron's linear part $W^{(i)}\xi^{(i)} + b^{(i)}$ is implemented by `fullyConnectedLayer(OutputSize)`
- neuron's activation σ can be implemented with different layers `reluLayer`, `sigmoidlayer`,etc
- loss function come at the end, implemented by either `regressionLayer` or `classificationLayer`

Table 1 shows the activation layers available in MATLAB. Typically, the most widely used activations for hidden layers are: reLu, tanh, softPlus; softmax is used after last fully connected layer for classification.

As an example, let us see now how we can build and train a neural network in MATLAB from scratch to approximate a given map $f : \mathbb{R}^n \to \mathbb{R}^m$.

To get started, fix the input size $n = 10$ and the output size $m = 20$. Let us choose to have a single hidden layer, namely a fully connected layer, and fix its number of neurons $num\_neurons = 50$. Then we can pick any non linearity, such as a sigmoid. So, to create a regression network, we can concatenate the main layers into a layer array:

MATLAB Layer	Activation Name	Function
`sigmoidLayer`	Sigmoid (or Logistic)	$\sigma(x) = 1/(1 + e^{-x})$
`softmaxLayer`	SoftMax	$\sigma(x) = e^x / \sum(e^x)$
`softplusLayer`	SoftPlus	$\sigma(x) = \log(1 + e^x)$
`tanhLayer`	Hyperbolic Tangent	$\sigma(x) = \tanh(x)$
`swishLayer`	Swish (or Sigmoid Linear Unit)	$\sigma(x) = x/(1 + e^{-x})$
`reluLayer`	ReLU (Rectified Linear Unit)	$\sigma(x) = \max(x, 0) = \begin{cases} x, \text{ if } x \geq 0 \\ 0, \text{ if } x < 0 \end{cases}$
`leakyReluLayer`(α)	Leaky ReLU	$\sigma(x) = \begin{cases} x, \text{ if } x \geq 0 \\ \alpha x, \text{ if } x < 0 \end{cases}$
`clippedReluLayer(C)`	Clipped ReLU	$\sigma(x) = \begin{cases} C, \text{ if } x \geq C \\ x, \text{ if } 0 \leq x \leq C \\ 0, \text{ if } x < 0 \end{cases}$
`eluLayer`(α)	Exponential Linear Unit	$\sigma(x) = \begin{cases} x, \text{ if } x \geq 0 \\ \alpha(e^x - 1), \text{ if } x < 0 \end{cases}$

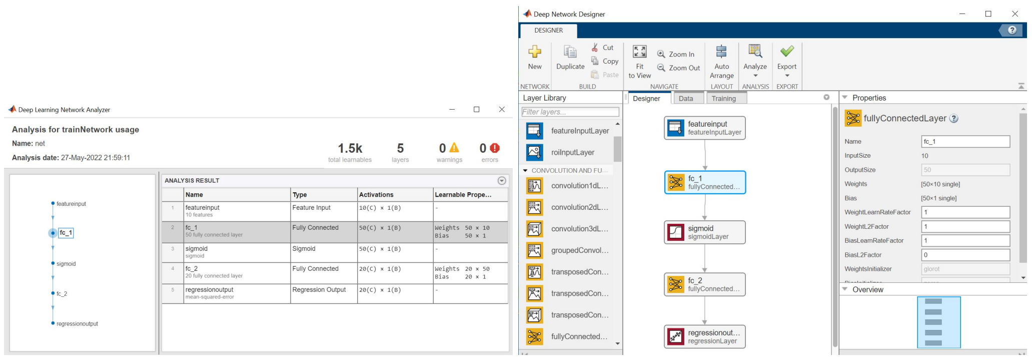
Table 1: Activation layers

Listing 28: Build a neural network by concatenating different layers

```matlab
n = 10; m = 20; num_neurons = 50;
layers = [featureInputLayer(n)           % Input layer
    fullyConnectedLayer(num_neurons)     % Hidden layer
    sigmoidLayer                         % Activation
    fullyConnectedLayer(m)               % Output layer
    regressionLayer];                    % loss layer
net = layerGraph(layers);
analyzeNetwork(net)
```

The `analyzeNetwork(net)` command (see Fig. 22, on the left) helps visualize and understand the architecture of the network, check that the architecture has been defined correctly, and detect problems before training, including missing or unconnected layers, incorrectly sized layer inputs, incorrect number of layer inputs, and invalid graph structures. The diagram on the left shows the layers connections. On the right, we can also check the number of learnable parameters for each layer. In this example, with input size 10, number of neurons 50 in hidden layer, and output size 20, the first fully-connected layer contains a (50x10) weight matrix and (50x1) bias, while the second fully-connected layer has a (20x50) weight matrix and (20x1) bias. In total, 1570 learnable parameters.

The `deepNetworkDesigner` app (see Fig. 22, on the right) allows to import an existing network or build the network from scratch by dragging the layers from the Layer Library.



Analysis for trainNetwork usage
Name: net
Analysis date: 27-May-2022 21:59:11

1.5k total learnables | 5 layers | 0 warnings | 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...
1	featureinput 10 features	Feature Input	10(C) × 1(B)	-
2	fc_1 50 fully connected layer	Fully Connected	50(C) × 1(B)	Weights 50 × 10 Bias 50 × 1
3	sigmoid sigmoid	Sigmoid	50(C) × 1(B)	-
4	fc_2 20 fully connected layer	Fully Connected	20(C) × 1(B)	Weights 20 × 50 Bias 20 × 1
5	regressionoutput mean-squared-error	Regression Output	20(C) × 1(B)	-

fullyConnectedLayer

Name	fc_1
InputSize	10
OutputSize	50
Weights	[50×10 single]
Bias	[50×1 single]
WeightLearnRateFactor	1
WeightL2Factor	1
BiasLearnRateFactor	1
BiasL2Factor	0
WeightsInitializer	glorot

Figure 22: (On the left): `analyzeNetwork`; (on the right) `deepNetworkDesigner`

Before starting the training, let us generate synthetic data in a random fashion:

Listing 29: Synthetic data to train the Neural Network

```matlab
N = 100;                         % N = number of samples
A = rand(m,n); x = randn(n,N);   % n = input size, m= output size
b = A*x + 0.1*randn(m,N);
xtrain = x';                     % xtrain is N x n
btrain = b';                     % btrain is N x m
```

Let us define some options with `trainingOptions` command. For instance we can choose:

- the solver, among three available options: `sgdm` (*Stochastic Gradient Descent with Momentum*), `rmsprop` (*Root Mean Square Propagation*) and `adam` (`Adaptive Moment` estimation). The solver is mandatory: depending on the solver we choose, we get a specific training option object;
- the maximum number of epochs (an *epoch* corresponds to a full pass over the entire training set)
- the initial *learning rate* and other properties regarding its schedule;
- the `ExecutionEnvironment` (choosing among `'auto', 'cpu', 'gpu', 'multi-gpu', 'parallel'`).
- the visualization of training progress by `Plots="training-progress"`.

To start the training, we use the main `trainNetwork` command: we can visualize the training progress and check if the loss is decreasing at each iteration (An iteration is one step taken in the solver algorithm towards minimizing the loss function using a mini-batch). After some computation, the returned object net contains all learnt parameters. For example, we can extract the weight matrix of the second fully-connected layer with $net.Layers(2).Weights$

Listing 30: Neural Networks: `trainingOptions` and `trainNetwork`

```
opt = trainingOptions("adam", Plots = "training-progress", MaxEpochs = 1500, ...
        InitialLearnRate = 0.1, LearnRateSchedule = "piecewise", ...
        LearnRateDropPeriod = 600, LearnRateDropFactor = 0.5, ...
        ExecutionEnvironment= "auto");
% run the training
net = trainNetwork(xtrain, btrain, layers, opt);
W1 = net.Layers(2).Weights                          % Extract Weights of fc-layer 2
B1 = net.Layers(2).Bias
W2 = net.Layers(4).Weights                          % Extract Weights of fc-layer 4
B2 = net.Layers(4).Bias
```

5.2 Radial Basis Function Networks

A Radial Basis Function (RBF) network is another universal approximator of any continuous function $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}^Q$ defined on any compact subset Ω, first formulated in a 1988 paper by Broomhead and Lowe. It is made of two main layers:

- a hidden layer of neurons with non-linear radial basis activation (typically, Gaussian), whose number of neurons is initially set to 0 and incremented gradually during the training process to fit the target output until it meets the specified mean squared error goal;
- an output layer of purely linear neurons.

The weights and biases of each neuron in the hidden layer define the position and width of a radial basis function. For example, we can use `radbas` basic command to create a Gaussian RBF. Fig. 23 shows a weighted sum of shifted Gaussian RBFs.

Listing 31: Neural Networks: `radbas` to compute weighted and shifted Gaussian RBFs

```
x = -3:.1:3; c = [0; 1.5; -2];     % x = row 1x61, c = column 3x1
omega = [1.5 0.5 1];               % omega = row 1x3
a = radbas(x-c);                   % x-c = a matrix 3x61 (implicit expansion)
plot(x, omega.*a'), hold on        % plot of 3 wighted RBFs
plot(x, omega*a,LineWidth=2)       % plot of the sum of the 3 RBF
```

To approximate $f : \mathbb{R}^d \to \mathbb{R}^Q$, given N training points in $\mathbb{R}^d$ and N output in $\mathbb{R}^Q$, we can define and train an RBF network by using the `newrb` command. To predict the value on new M points in $\mathbb{R}^d$, we collect coordinates in a dxM matrix and apply the RBF network object, returning a new QxM output array:

- `RBFnet = newrb(in, out, MSEgoal, RBFspread, MaxNumNeurons)`

The 4th input is the spread parameter of Gaussian RBF: the larger spread is, the smoother the function approximation. Too large a spread means a lot of neurons are required to fit a fast-changing function. Too small a spread means many neurons are required to fit a smooth function, and the network might not generalize well. We may need to call `newrb` with different spreads to find the best value for a given problem.

In the next example, we take input size $d = 2$ and output size $Q = 2$. In this case, after we train the RBF network, we can make predictions on a uniform grid and visualize Q surfaces. Fig. 24 shows these surfaces.

Listing 32: Neural Networks: training an RBF network to approximate $f : \mathbb{R}^2 \to \mathbb{R}^2$

```
N = 100; d = 2;                                     % N input in R^d, d=2
in = randn(d,N);                                    % in (dxN), out (QxN)
% generate syntehic data for Q=2 outputs
Q = 2;                                              % Q = 2 output size
out = sin(in(1,:)).*exp(-0.5*in(1,:)) + sin(in(2,:)).*exp(-0.1*in(2,:));
out(2,:)= cos(2*in(1,:)).*exp(-0.1*in(1,:))+cos(3*in(2,:)).*exp(-0.2*in(2,:));
% Train radial Basis Neural network
```

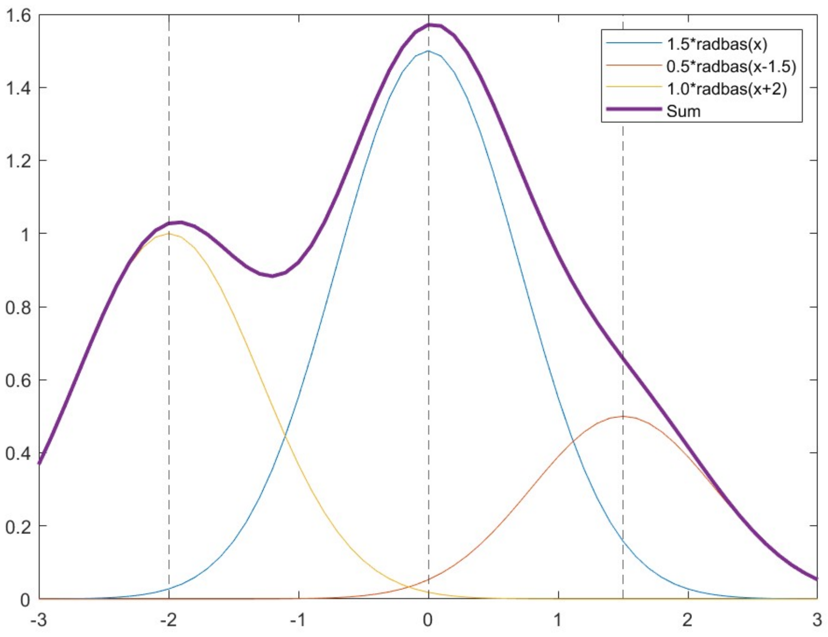


Figure 23: Weighted sum of three shifted Gaussian RBFs

```
MSEgoal = 0.02; RBFspread = 1;
RBFnet2 = newrb(in, out, MSEgoal, RBFspread);     % train the RBF neural network
% Prediction on a grid of new points
Nx = 20; Ny = 20;
xnew = linspace(min(in(1,:)), max(in(1,:)), Nx);
ynew = linspace(min(in(2,:)), max(in(2,:)), Ny);
[Xnew,Ynew] = ndgrid(xnew,ynew);                  % Nx x Ny uniform grid
newIn = [Xnew(:),Ynew(:)]';                       % d x M (M=Nx*Ny)
newOut = RBFnet2(newIn);                          % Q x M predictions
 % Visualization of Q surfaces
figure
for q = 1:Q
  subplot(1,Q,q)
  scatter3(in(1,:), in(2,:), out(q,:), 'filled') % plot given data
  newOut_q = reshape(newOut(q,:),Nx,Ny);          % reshape qth predictions
  hold on, surf(Xnew,Ynew,newOut_q)               % draw qth surface
  title("Target Q=" + q), shading interp
end
```

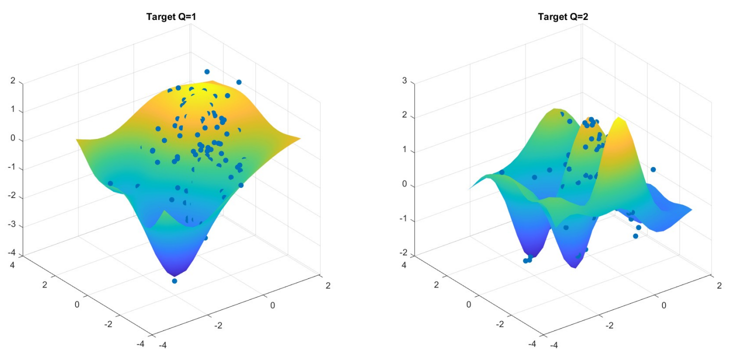


Figure 24: Radial Basis Neural Networks trained to approximate $f : \mathbb{R}^2 \to \mathbb{R}^2$

5.3 Neural Networks to approximate complex EKF systems

Neural Networks (NN) are "black-box" models that sometimes can be a valid alternative to very complicated "white-box" models. To illustrate the flexibility of neural networks, we want to describe a real striking application, where a simple neural network is used to approximate a complex dynamic systems, like a rechargeable battery, and provide an accurate estimate of unmeasurable quantities, like its State of Charge.

Rechargeable batteries, in particular *lithium-ion (Li-ion) batteries*, were discovered after intensive research in ’70 and ’80 [¶], commercialized in 1991 and are now widely used in everything, from electric/hybrid vehicles to portable electronics (laptops, mobile phones, etc). Battery Management Systems (BMS) are complex electronic systems required to monitor the *State Of Charge* (SOC) of the batteries, which is critical information to prevent from system blockage, overcharging, operating outside the safe operating area. But SOC cannot be measured directly, so an accurate estimate of SOC is key to ensure reliable and affordable electrified vehicles and devices (see [7]). Moreover, estimation algorithm must guarantee the simplest code implementation.

By definition, SOC is the ratio of the available level of charge $Q(t)$ to the maximum level Q_{max} available when the battery is fully charged (also, the maximum capacity expressed by the manufacturer in Amp-hour, Ah). So SOC is a percentage given by $SOC(t) = -Q(t)/Q_{max}$. Taking derivatives and putting $I(t) = dQ/dt$, we get the *SOC differential equation*:

$$\frac{d}{dt} SOC(t) = -\frac{I(t)}{Q_{max}} \tag{5.3.1}$$

Estimating SOC is still a significant engineering challenge due to the nonlinear temperature, health, and SOC dependent behavior of Li-ion batteries, so different estimation methods are available (see [7] for a review). Let us compare three methods: Coulomb Counting (CC), Extended Kalman Filter (EKF), and Neural Network (NN). In summary, CC turns out to be simple but too poor; EKF is more accurate, but very complicated and with heavy code implementation; NN is the most efficient approach allowing similar accuracy as the EKF, but lighter code implementation.

First method: Coulomb Counting method (CC).

Coulomb Counting method is the simplest method to estimate SOC (see [20]), obtained by approximating the derivative in (5.3.1) with a finite difference. The accuracy of CC method is poor because it ignores all temperature and voltage effects.

$$SOC(t) = SOC(t - \Delta t) + \frac{I(t)}{Q_{max}} \Delta t \tag{5.3.2}$$

Second method: Extended Kalman Filter (EKF) in Simulink.

The Kalman Filter is a more advanced algorithm, based on an iterative prediction-correction process, that can be applied to estimate the internal state of a dynamic system, using its model (i.e. its physical laws), and multiple noisy measurements from sensors. It was introduced by R. Kalman in 1960, first for linear systems, then extend to non linear ones. Because of its efficiency, it is still widely used nowadays in signal processing, control and navigation systems, econometrics, etc. There exists some excellent literature such as [10] and [21] addressing derivation and theory behind the Kalman filter. See also MATLAB Documentation [19].

In this case, to estimate SOC, EKF leverages on three measurements, namely voltage V, current I, and temperature T. Briefly, voltage is estimated from measured current and temperature, and then compared with the measured voltage, finally the voltage error is incorporated back into the SOC estimation:

$$SOC = \hat{f}_{EKF}(I, V, T) \tag{5.3.3}$$

To implement EKF method, we can leverage on Simulink environment, where the `EKF block` is already available in the Control System libraries. Preprocessed input data (I, V, T) for this example have been collected from [14]. Simulink is fed by variables loaded in MATLAB workspace. We can set automatic ODE solver (variable-step in simulation, or fixed-step[‖] for automatic embedded C code generation). Running the simulation for about 13 hours, the solver efficiently computes the SOC estimation as shown in Fig. 26.

Third method: Neural Networks (NN).

In contrast to previous EKF approach, which requires precise parameters and knowledge of the battery composition as well as its physical response, using *Neural Networks* is a totally data-driven approach that instead requires minimal knowledge of the battery or its nonlinear behavior. Remember that each neuron operates on input x as $y = \sigma(wx + b)$, where w and b are learnable parameters to be optimized and σ is a non-linear *activation* (see 1). Neurons are aggregated in different kind of layers, such as Fully Connected Layers. To help the network to estimate SOC better, we provide some additional historical memory by two more inputs: average current and average voltage:

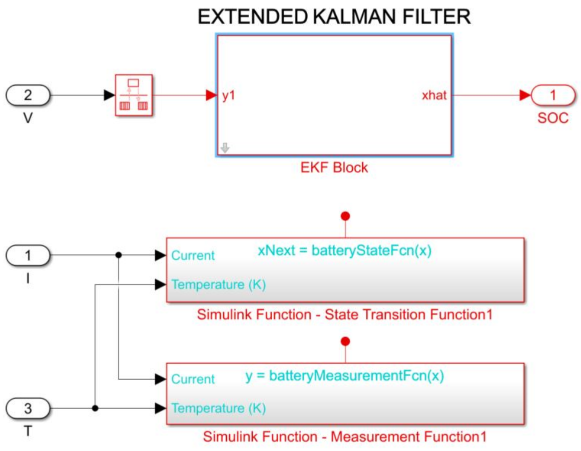
$$SOC = \hat{f}_{NN}(I, V, T, I_{avg}, V_{avg}) \tag{5.3.4}$$

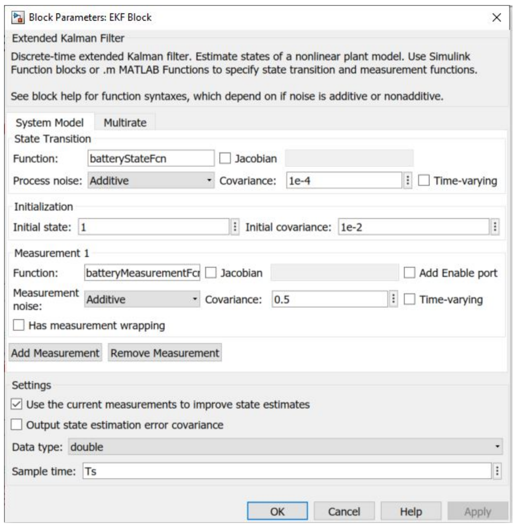
Let us see in details three steps required to build a Neural network in MATLAB and then, optionally, reuse it into Simulink.

Step 1: define a feedforward Neural Network from scratch.
Similarly to what we did in section 5.1, we concatenate some layers:

[¶]Li-ion batteries were discovered by M. S. Whittingham, J. Goodenough and A. Yoshino, who won the Nobel Prize for Chemistry 2019.

[‖]Note that fixed-step solvers have no zero-crossing and no checking of state error.


EXTENDED KALMAN FILTER
2
V
y1
xhat
1
SOC
EKF Block
1
I
Current
xNext = batteryStateFcn(x)
Temperature (K)
Simulink Function - State Transition Function1
Current
y = batteryMeasurementFcn(x)
Temperature (K)
3
T
Simulink Function - Measurement Function1


Block Parameters: EKF Block

Extended Kalman Filter

Discrete-time extended Kalman filter. Estimate states of a nonlinear plant model. Use Simulink Function blocks or .m MATLAB Functions to specify state transition and measurement functions.

See block help for function syntaxes, which depend on if noise is additive or nonadditive.

System Model | Multirate

State Transition

Function: batteryStateFcn ☐ Jacobian

Process noise: Additive Covariance: 1e-4 ☐ Time-varying

Initialization

Initial state: 1 Initial covariance: 1e-2

Measurement 1

Function: batteryMeasurementFcı ☐ Jacobian ☐ Add Enable port

Measurement noise: Additive Covariance: 0.5 ☐ Time-varying

☐ Has measurement wrapping

Add Measurement | Remove Measurement

Settings

☑ Use the current measurements to improve state estimates

☐ Output state estimation error covariance

Data type: double

Sample time: Ts

OK | Cancel | Help | Apply

Figure 25: *SOC estimation with Extended Kalman Filter in Simulink*. Left: the main EKF Block receives voltage V s input and returns SOC estimate. EKF Block calls two functions: one for the SOC state equation 5.3.1; the other for measurement function. Both have current I and temperature T as inputs and are implemented with Simulink Function blocks. Right: the EKF block parameters dialog, where we type the two Simulink Function names.

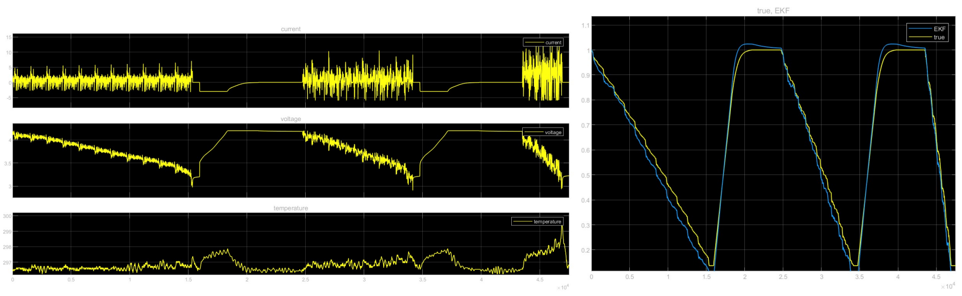


Figure 26: *Simulation of SOC estimator with EKF method in Simulink*. Left: 3 input signals, i.e. current, voltage and temperature. Right: SOC output, and comparison of true SOC with the SOC estimated with EKF.

- an input layer with 5 input features $(I, V, T, I_{avg}, V_{avg})$;
- one hidden fully-connected layer with 64 neurons;
- some non-linear activation layers, like `reluLayer` (see Table 1);
- a fully connected layer with 1 single output (for SOC) and a regression layer to compute loss function.

Listing 33: Neural Networks: define the layer architecture for SOC estimation

```
numFeatures = 5;
numResponses = 1;
numHiddenNeurons = 64;   % Number of hidden neurons
% Define network architecture
layers = [ featureInputLayer(numFeatures,"Normalization","zerocenter") % 5 inputs
           fullyConnectedLayer(numHiddenNeurons)      % 1st hidden layer
           reluLayer
           fullyConnectedLayer(numResponses)      % output layer (1 single output for SOC)
           reluLayer
           regressionLayer];              % loss function (RMSE)
```

Step 2: Specify training options and run the training.

- Use `trainingOptions` command to set multiple options: the solver (i.e. `"adam"` for ADAptive Moment), the gradient threshold, the number of epochs, the minibatch size, the initial learning rate, the learning rate drop factor, a learning rate drop period, the validation data, the validation frequency, the execution environment (GPU vs CPU), etc.
- Use `trainNetwork` to run the training to optimize all learnable parameters.

  ```
  net = trainNetwork (trainX , trainY ,layers , options ).
  ```

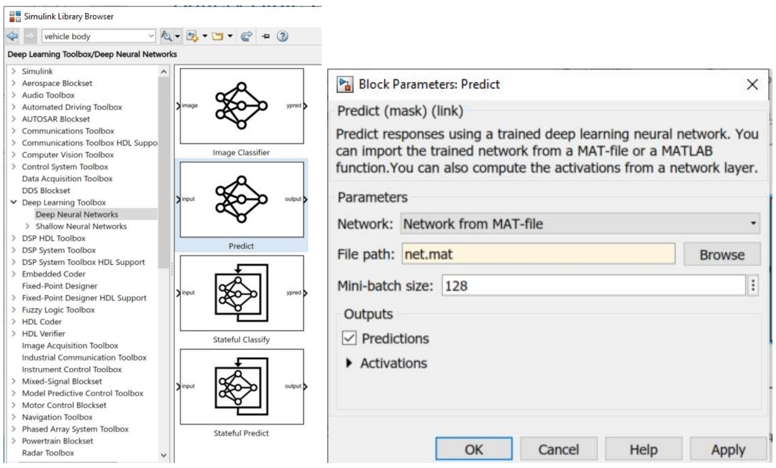
Listing 34: Neural Networks: train the network for SOC estimation

```matlab
% Define training options
options = trainingOptions("adam", Plots = "training-progress", ...
        MaxEpochs = 1200, Shuffle = "every-epoch", ...
        MiniBatchSize = 128, GradientThreshold = 1, ...
        InitialLearnRate = 0.01, LearnRateSchedule = "piecewise", ...
        LearnRateDropFactor = 0.1, LearnRateDropPeriod = 400, ...
        ValidationData = {valX, valY}, ValidationFrequency = 30, ...
        ExecutionEnvironment = "auto");
% Run training
net = trainNetwork(trainX,trainY,layers,options);

% save network for reuse  in Simulink
save net.mat net   % save the network in a mat file

% Use test dataset to estimate RMSE
Y_predicted = predict(net,testX);
residuals = testY - Y_predicted;
RMSE = sqrt(mean(residuals).^2);
```

Note: the *Experiment Manager* App can help set up the training options as it allows to automatically run many tests with different training options, and check which one yields the best performance. Search in MATLAB Documentation for details.

Step 3: (optional) Reuse the trained neural network in Simulink.
As an option, the trained NN could be integrated into Simulink to take advantage of the system-level simulation environment and the automatic code generation feature (i.e. the Simulink model could be converted automatically into C/C++ code by using Embedded Coder®). To do this, we can do the following:

- save the trained network `net` into a `.mat` file, i.e. `save net.mat net`
- check you have installed *MATLAB Coder Interface for Deep Learning Libraries* add-on to call optimized libraries, like MKL-DNN. If not, click on Add-Ons→Get Add-Ons from Home of MATLAB Desktop and search for this add-on name
- open the Simulink Editor (from MATLAB Home, click on New→Simulink Model→Blank Model). Then click on Library Browser, scroll down to Deep Learning Toolbox→Deep Neural Networks, and drag the Predict Block
- double click on the block to open its Block Parameter dialog, and in the File Path field type the `mat` filename where the trained network was saved (see Fig. 27 and Fig. 28).



Block Parameters: Predict

Predict (mask) (link)

Predict responses using a trained deep learning neural network. You can import the trained network from a MAT-file or a MATLAB function.You can also compute the activations from a network layer.

Parameters

Network: Network from MAT-file

File path: net.mat

Mini-batch size: 128

Outputs

[✓] Predictions

Activations

Figure 27: Simulink Library Browser and the Predict Block in Deep Learning Toolbox (on the left); the dialogue of Predict Block (on the right)

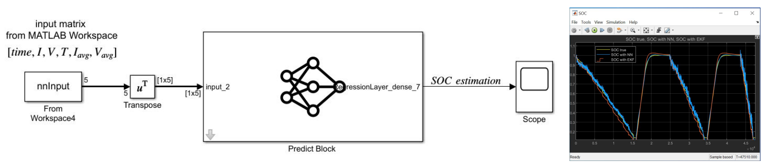


Figure 28: Left: Block diagram in Simulink with the Predict Block. Right: comparison of SOC estimations (true, NN and EKF).

Resources	Research	Teaching	***Links***
MATLAB® Code of this paper	✓	✓	github.com/ppanares2/Approximation-with-MATLAB
MATLAB® Academia	✓	✓	www.mathworks.com/academia
Campus-Wide License	✓	✓	www.mathworks.com/products/matlab-campus
MATLAB Documentation	✓	✓	www.mathworks.com/help
MATLAB Discover How to Solve	✓	✓	www.mathworks.com/discovery
MATLAB-based Books	✓	✓	www.mathworks.com/academia/books
MATLAB File Exchange	✓		www.mathworks.com/matlabcentral
MATLAB Add-Ons	✓		www.mathworks.com/products/matlab/add-on-explorer
MATLAB Communities	✓		.../matlabcentral/content/communities
The MATLAB Blog	✓		https://blogs.mathworks.com/matlab/
MATLAB for Research	✓		www.mathworks.com/academia/research
MathWorks Excellence in Innovation	✓		github.com/mathworks/MathWorks-Excellence...
MathWorks® Books Program	✓		www.mathworks.com/academia/books/join
MATLAB in Science Gateways	✓		.../academia/research/science-gateways
MATLAB for Open Science	✓	✓	www.mathworks.com/discovery/open-science
Using MATLAB with Python	✓	✓	.../products/matlab/matlab-and-python
MATLAB in the Cloud	✓		www.mathworks.com/solutions/cloud
MATLAB Online™	✓	✓	matlab.mathworks.com/
MATLAB Drive™	✓	✓	drive.matlab.com
MATLAB Cloud Center	✓	✓	cloudcenter.mathworks.com/
MathWorks Reference Architectures	✓		github.com/mathworks-ref-arch
MATLAB Dockerfile	✓		hub.docker.com/r/mathworks/matlab
MATLAB for Teaching		✓	www.mathworks.com/academia/educators
MATLAB Grader™		✓	grader.mathworks.com/
MATLAB Grader™ for LMS		✓	.../products/matlab-grader/lms.html
MATLAB Courseware		✓	www.mathworks.com/academia/courseware
MATLAB Onramp Training		✓	matlabacademy.mathworks.com/#getting-started
Textbooks by Cleve Moler		✓	www.mathworks.com/moler
Cleve Moler on Mathematics		✓	blogs.mathworks.com/cleve
MATLAB for Computational Thinking		✓	.../discovery/computational-thinking
Mathematical Modeling		✓	.../solutions/mathematical-modeling
MathWorks Math Modeling Challenge		✓	.../mathworks-math-modeling-challenge
MATLAB Hackathons		✓	github.com/mathworks/awesome-matlab-hackathons
MATLAB Student Competition	✓	✓	www.mathworks.com/academia/student-competitions

Table 2: Resources available in MATLAB Ecosystem for research and teaching

6 Conclusion

MATLAB environment offers a unique platform to explore a variety of algorithms for approximation tasks, using commands or interactive apps. In this paper, we have included snippets of MATLAB code to see how to implement different topics:

- function approximation through orthogonal polynomials and Fourier series;
- wavelet analysis and multiresolution decomposition;
- multivariate scattered interpolation with radial basis functions;
- surrogate global optimization via a cubic RBF interpolator with linear precision;
- kernelized support vector machines for both regression and classification;
- universal approximators with neural networks or RBF networks;
- neural networks to approximate an Extended Kalman Filter (to estimate the state of charge of a rechargeable battery).

MATLAB, with its broad spectrum of Toolboxes, included in the Campus-Wide License available in most of Universities, is a great tool able to inspire cutting-edge research and engaging lectures.

In Table 2 there is a list of additional resources from MATLAB ecosystem to continue a proficient and deeper usage of MATLAB.

Acknowledgements

The author wishes to warmly thank Prof. Alessandra De Rossi for the precious collaboration and support.

References

[1] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*. Springer, 2018, ISBN-13: 978-3319944623

[2] A. Biess, M. Nagurka, T. Flash, *Simulating discrete and rhythmic multi-joint human arm movements by optimization of nonlinear performance indices*. Biological Cybernetics (95), 2006, pp. 31-53

[3] S. L. Brunton, J. N. Kutz, *Data Driven Science and Engineering: Machine Learning, Dynamical Systems and Control*. Springer Series in the Data Sciences, 2019, ISBN-13: 978-1108422093

[4] M. D. Buhmann, *Radial Basis Functions: Theory and Implementation*. Cambridge Monogr. Appl. Comput. Math., vol. 12, Cambridge Univ. Press, Cambridge, 2003.

[5] O. Calin, *Deep Learning Architectures: A Mathematical Approach*. Springer Series in the Data Sciences, 2020, ISBN-13: 978-3030367206

[6] R. Cavoretto, A. De Rossi. Software for Approximation 2022 (SA2022). *Dolomites Res. Notes Approx.*, Special Issue SA2022, 15:i–ii, 2022.

[7] W. Y. Chang, *The State of Charge Estimating Methods for Battery: A Review*. ISRN Applied Mathematics, Vol. 2013, Article ID 953792

[8] I. Daubechies, *Ten Lectures on Wavelets*. CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1992, ISBN-13: 978-0898712742

[9] G. E. Fasshauer, *Meshless Approximation Methods with MATLAB*. World Scientific Publishing Co., Interdisciplinary Mathematical Sciences, Vol.6, 2007, ISBN-13: 978-9812706331

[10] M. S. Grewal, A. P. Andrews, *Kalman Filtering: Theory and Practice with MATLAB*. Wiley-IEEE Press, 4th Edition, 2015, ISBN-13: 978-1118851210

[11] H. M. Gutmann, *A Radial Basis Function Method for Global Optimization*. Journal of Global Optimization (19), 2001, pp. 201-227

[12] S. Hubbert, Q. T. Lê Gia, T. M. Morton, *Spherical radial basis functions: theory and applications*. Springer Briefs in Mathematics, 2015, ISBN-13: 9783319179384

[13] A. Iske, *Approximation Theory and Algorithms for Data Analysis*. Springer, Texts in Applied Mathematics (68), 2018, ISBN-13: 978-3030052270

[14] P. Kollmeyer, C. Vidal, M. Naguib, M. Skells, *LG 18650HG2 Li-Ion Battery Data and Example Deep Neural Network XEV SOC Estimator Script*. Mendeley, 2020, https://doi.org/10.17632/CP3473X7XV.3

[15] S. G. Mallat, *A Theory for Multiresolution Signal Decomposition: The Wavelet Representation*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 11, No. 7, 1989, p. 674-693

[16] S. G. Mallat, *A Wavelet Tour of Signal Processing: The Sparse Way*. Academic Press, 1999, ISBN-13: 978-0124666061

[17] M. Mehra, *Wavelets Theory and Its Applications. A First Course*. Forum for Interdisciplinary Mathematics, Vol. 37, Springer, 2018, ISBN-13: 978-9811325946

[18] Y. Meyer, *Wavelets and Operators*. Cambridge Studies in Advanced Mathematics, Vol. 37, Cambridge University Press, 1995, ISBN-13: 978-0521458696

[19] MathWorks, *MATLAB Documentation*. www.mathworks.com/help

[20] K. S. Ng, C. S. Moo, Y. P. Chen, Y. C. Hsieh, *Enhanced Coulomb counting method for estimating state-of-charge and state-of-health of lithium-ion batteries*. Applied Energy, 2009, vol. 86, no. 9, pp. 1506-1511

[21] D. Simon, *Optimal State Estimation: Kalman, H^∞, and Nonlinear Approaches*. Wiley-Interscience, 2006, ISBN-13: 978-0471708582

[22] G. Strang, T. Nguyen, *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1996, ISBN-13: 978-0961408879

[23] A. Teolis, *Computational signal processing with wavelets*. BirkhÃ¤user, 1998, ISBN-13: 978-1461286721