Quasi-Newton Methods (10.9)

Objective

To borrow from my **Conjugate Gradient (CG)** presentation:

Minimize a function f with the following properties:

- f can be evaluated at an *N*-dimensional point **P** (i.e. $f(\mathbf{P})$)
- The gradient of f can be evaluated at an *N*-dimensional point **P** (i.e. $\nabla f(\mathbf{P})$)

Similarities to CG

- Quasi-Newton methods and CG both:
  - Require computation of f and its gradient at arbitrary points.
  - Reach the exact minimum of a quadratic form in *N* line minimizations.
  - Converge quadratically for general functions.

CG vs. Quasi-Newton Methods

- **CG's main advantage** is that it requires intermediate storage on the order of *N*, while quasi-Newton methods require storage on the order of *N* x *N*. The book notes "for any moderate *N*, this hardly matters."
- However, quasi-Newton methods:
    - Typically **converge faster** than CG (we'll get into this later).
    - Are **more robust** to ill-conditioned problems, special conditions, and numerical errors/approximations (**good general-purpose**).
    - **Don't need to restart** every *N* iterations.

But how does it do it?

In other words, what makes quasi-Newton methods tick?

Newton's Method in Optimization

- To understand quasi-Newton methods, **you need to understand actual Newton methods**.
- **The big idea:** minimize f by constructing a sequence $\{x_k\}$ from an initial guess x_0 that converges toward the minimum of f using a sequence of **second-order Taylor approximations** of f around the iterates.

Source: Newton's method in optimization @ Wikipedia

Newton's Method in Optimization

- The second-order Taylor expansion of f around x_k is

$$f(x_k + t) \approx f(x_k) + f'(x_k)t + \frac{1}{2}f''(x_k)t^2$$

- If the second derivative is positive, **this approximation is a convex function of t**, meaning we can find its minimum by setting its derivative to 0, like so

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(f(x_k) + f'(x_k)t + \frac{1}{2}f''(x_k)t^2\right) = f'(x_k) + f''(x_k)t = 0, t = -\frac{f'(x_k)}{f''(x_k)}$$

- Finally, we construct the next iteration in the sequence like so:

$$x_{k+1} = x_k + t = x_k - \frac{f'(x_k)}{f''(x_k)}$$

Source: Newton's method in optimization @ Wikipedia

Newton's Method in Optimization (with the Hessian)

- Now, with the Hessian $\mathbf{A}$ (matrix of second-order partial derivatives)
- The second-order Taylor expansion of f around x_k is

$$f(x_k + t) \approx f(x_k) + (f'(x_k) \cdot t) + \left(\frac{1}{2} \cdot t \cdot \mathbf{A} \cdot t\right)$$

- If the Hessian is positive-definite, **this approximation is a convex function of t**, meaning we can find its minimum by setting its derivative to 0, like so

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(f(x_k + t)\right) = f'(x_k) + (\mathbf{A} \cdot t) = 0, t = -\frac{f'(x_k)}{\mathbf{A}} = -\mathbf{A}^{-1} \cdot f'(x_k)$$

- Finally, we construct the next iteration in the sequence like so:

$$x_{k+1} = x_k + t = x_k - \left(\mathbf{A}^{-1} \cdot f'(x_k)\right)$$

Adapted from 10.9 with modified notation.

The Problem

- Note the following excerpts from Newton's method: **"If the second derivative is positive" / "if the Hessian is positive-definite"**
- **This is not a given!**
- From 10.9:
  "In general, far from a minimum, we have no guarantee that the Hessian is positive-definite. Taking the actual Newton step with the real Hessian can move us to points where the function is **increasing** in value."

The Solution

- Instead of using the Hessian itself, **iteratively build an approximation of it** (10.9.1):

$$\lim_{i \to \infty} \mathbf{H}_i = \mathbf{A}^{-1}$$

- We can build this approximation in such a way that it is **always positive-definite**, meaning we **always move in a downhill direction**.
    - However, even with a positive-definite approximation, we may move too far in the minimizing direction, so **we must line search** to choose the correct step along the minimizing direction.
- Then, as we approach the minimum, we approach the **true Hessian** and enjoy the **quadratic convergence of Newton's method**.

The Heart of the Solution

- Of course, this begs the question, **how do we build the approximation?**
- We will show **SciPy**'s implementation of the **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** update formula.
- Let:
  - $\mathbf{I}$ be the identity matrix.
  - $\overrightarrow{s_k}$ be the change in position from rounds $k \to k+1$
  - $\overrightarrow{y_k}$ be the change in gradient from rounds $k \to k+1$
  - $\rho_k = \frac{1}{\langle \overrightarrow{s_k}, \overrightarrow{y_k} \rangle}$ (inverse of the inner/dot product of $\overrightarrow{s_k}$ and $\overrightarrow{y_k}$)
    - $\langle [5 \quad 3], [9 \quad 7] \rangle = (5 \cdot 9) + (3 \cdot 7) = 66$
  - $\mathbf{H}_k$ be the Hessian approximation from round k
- $\mathbf{H}_{k+1} = \left(\mathbf{I} - (\overrightarrow{s_k} * \overrightarrow{y_k} * \rho_k)\right) \cdot \mathbf{H}_k \cdot \left(\mathbf{I} - (\overrightarrow{y_k} * \overrightarrow{s_k} * \rho_k)\right) + (\rho_k * \overrightarrow{s_k} * \overrightarrow{s_k})$
  - Dimensionality is mismatched, this'll be explained later.

Bonus: H converges to A^{-1} in N steps, if f is a quadratic form!

Next Up

- BFGS is the **default method** used in SciPy's `minimize` function when no constraints or bounds are provided.
- **SciPy** is a highly performant, production-ready library for scientific computing.
- **To wrap up this presentation, we'll walk through SciPy's implementation of BFGS.**

```
def _minimize_bfgs(fun, x0, args=(), jac=None, callback=None,
                   gtol=1e-5, norm=np.inf, eps=_epsilon, maxiter=None,
                   disp=False, return_all=False, finite_diff_rel_step=None,
                   xrtol=0, c1=1e-4, c2=0.9,
                   hess_inv0=None, **unknown_options):
    """
    Minimization of scalar function of one or more variables using the
    BFGS algorithm.

    Options
    -------
    disp : bool
        Set to True to print convergence messages.
    maxiter : int
        Maximum number of iterations to perform.
    gtol : float
        Terminate successfully if gradient norm is less than `gtol`.
    norm : float
        Order of norm (Inf is max, -Inf is min).
    eps : float or ndarray
        If `jac is None` the absolute step size used for numerical
        approximation of the jacobian via forward differences.
    return_all : bool, optional
```

```
f = sf.fun
myfprime = sf.grad

old_fval = f(x0)
gfk = myfprime(x0)

k = 0
N = len(x0)
I = np.eye(N, dtype=int)
Hk = I if hess_inv0 is None else hess_inv0

# Sets the initial step guess to dx ~ 1
old_old_fval = old_fval + np.linalg.norm(gfk) / 2

xk = x0
if retall:
    allvecs = [x0]
warnflag = 0
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                     old_fval, old_old_fval, amin=1e-100,
```

```
f = sf.fun
myfprime = sf.grad

old_fval = f(x0)
gfk = myfprime(x0)
```

Initialize the value of the function and its gradient

```
k = 0
N = len(x0)
I = np.eye(N, dtype=int)
Hk = I if hess_inv0 is None else hess_inv0

# Sets the initial step guess to dx ~ 1
old_old_fval = old_fval + np.linalg.norm(gfk) / 2

xk = x0
if retall:
    allvecs = [x0]
warnflag = 0
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
```

```
f = sf.fun
myfprime = sf.grad

old_fval = f(x0)
gfk = myfprime(x0)

k = 0
N = len(x0)
I = np.eye(N, dtype=int)
Hk = I if hess_inv0 is None else hess_inv0

# Sets the initial step guess to dx ~ 1
old_old_fval = old_fval + np.linalg.norm(gfk) / 2

xk = x0
if retall:
    allvecs = [x0]
warnflag = 0
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                     old_fval, old_old_fval, amin=1e-100,
```

Set H_0 to the identity matrix I

```
f = sf.fun
myfprime = sf.grad

old_fval = f(x0)
gfk = myfprime(x0)

k = 0
N = len(x0)
I = np.eye(N, dtype=int)
Hk = I if hess_inv0 is None else hess_inv0

# Sets the initial step guess to dx ~ 1
old_old_fval = old_fval + np.linalg.norm(gfk) / 2

xk = x0
if retall:
    allvecs = [x0]
warnflag = 0
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                     old_fval, old_old_fval, amin=1e-100,
```

Initialize guess of step along minimizing direction

```
    gnorm = vecnorm(gfk, ord=norm)
    while (gnorm > gtol) and (k < maxiter):
        pk = -np.dot(Hk, gfk)
        try:
            alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                     _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                          old_fval, old_old_fval, amin=1e-100,
                                          amax=1e100, c1=c1, c2=c2)
        except _LineSearchError:
            # Line search failed to find a better solution.
            warnflag = 2
            break

        sk = alpha_k * pk
        xkp1 = xk + sk

        if retall:
            allvecs.append(xkp1)
        xk = xkp1
        if gfkp1 is None:
            gfkp1 = myfprime(xkp1)

        yk = gfkp1 - gfk
        gfk = gfkp1
        k += 1
        intermediate_result = OptimizeResult(x=xk, fun=old_fval)
```

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

2/5 termination conditions:

(1) Norm of gradient less than tolerance (virtually 0 by default)

(2) Iterations hit maximum

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

Calculate search direction (p_k) using the approximated Hessian

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

Perform a line search to determine the optimal step size (α_k) along the search direction. Terminate on failure.

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

Calculate the change in position s_k and get the current point by applying s_k to the previous point.

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

Calculate the gradient at the current point

```
gnorm = vecnorm(gfk, ord=norm)
while (gnorm > gtol) and (k < maxiter):
    pk = -np.dot(Hk, gfk)
    try:
        alpha_k, fc, gc, old_fval, old_old_fval, gfkp1 = \
                 _line_search_wolfe12(f, myfprime, xk, pk, gfk,
                                      old_fval, old_old_fval, amin=1e-100,
                                      amax=1e100, c1=c1, c2=c2)
    except _LineSearchError:
        # Line search failed to find a better solution.
        warnflag = 2
        break

    sk = alpha_k * pk
    xkp1 = xk + sk

    if retall:
        allvecs.append(xkp1)
    xk = xkp1
    if gfkp1 is None:
        gfkp1 = myfprime(xkp1)

    yk = gfkp1 - gfk
    gfk = gfkp1
    k += 1
```

Calculate the change in gradient y_k, save the current gradient, and increment the iteration.

```
    break
gnorm = vecnorm(gfk, ord=norm)
if (gnorm <= gtol):
    break

#  See Chapter 5 in  P.E. Frandsen, K. Jonasson, H.B. Nielsen,
#  O. Tingleff: "Unconstrained Optimization", IMM, DTU.  1999.
#  These notes are available here:
#  http://www2.imm.dtu.dk/documents/ftp/publlec.html
if (alpha_k*vecnorm(pk) <= xrtol*(xrtol + vecnorm(xk))):
    break

if not np.isfinite(old_fval):
    # We correctly found +-Inf as optimal value, or something went
    # wrong.
    warnflag = 2
    break

rhok_inv = np.dot(yk, sk)
# this was handled in numeric, let it remains for more safety
# Cryptic comment above is preserved for posterity. Future reader:
# consider change to condition below proposed in gh-1261/gh-17345.
if rhok_inv == 0.:
    rhok = 1000.0
    if disp:
```

```
    break
gnorm = vecnorm(gfk, ord=norm)
if (gnorm <= gtol):
    break
```

Repeat of one of the termination conditions in the function's `while` loop.

```
#  See Chapter 5 in  P.E. Frandsen, K. Jonasson, H.B. Nielsen,
#  O. Tingleff: "Unconstrained Optimization", IMM, DTU.  1999.
#  These notes are available here:
#  http://www2.imm.dtu.dk/documents/ftp/publlec.html
if (alpha_k*vecnorm(pk) <= xrtol*(xrtol + vecnorm(xk))):
    break

if not np.isfinite(old_fval):
    # We correctly found +-Inf as optimal value, or something went
    # wrong.
    warnflag = 2
    break

rhok_inv = np.dot(yk, sk)
# this was handled in numeric, let it remains for more safety
# Cryptic comment above is preserved for posterity. Future reader:
# consider change to condition below proposed in gh-1261/gh-17345.
if rhok_inv == 0.:
    rhok = 1000.0
    if disp:
```

```
    break
gnorm = vecnorm(gfk, ord=norm)
if (gnorm <= gtol):
    break
```

Terminate if step size is below the tolerance.

```
#  See Chapter 5 in  P.E. Frandsen, K. Jonasson, H.B. Nielsen,
#  O. Tingleff: "Unconstrained Optimization", IMM, DTU.  1999.
#  These notes are available here:
#  http://www2.imm.dtu.dk/documents/ftp/publlec.html
if (alpha_k*vecnorm(pk) <= xrtol*(xrtol + vecnorm(xk))):
    break
```

```
if not np.isfinite(old_fval):
    # We correctly found +-Inf as optimal value, or something went
    # wrong.
    warnflag = 2
    break

rhok_inv = np.dot(yk, sk)
# this was handled in numeric, let it remains for more safety
# Cryptic comment above is preserved for posterity. Future reader:
# consider change to condition below proposed in gh-1261/gh-17345.
if rhok_inv == 0.:
    rhok = 1000.0
    if disp:
```

```
    break
gnorm = vecnorm(gfk, ord=norm)
if (gnorm <= gtol):
    break

#  See Chapter 5 in  P.E. Frandsen, K. Jonasson, H.B. Nielsen,
#  O. Tingleff: "Unconstrained Optimization", IMM, DTU.  1999.
#  These notes are available here:
#  http://www2.imm.dtu.dk/documents/ftp/publlec.html
if (alpha_k*vecnorm(pk) <= xrtol*(xrtol + vecnorm(xk))):
    break
```

Terminate if the function value is infinite

```
if not np.isfinite(old_fval):
    # We correctly found +-Inf as optimal value, or something went
    # wrong.
    warnflag = 2
    break

rhok_inv = np.dot(yk, sk)
# this was handled in numeric, let it remains for more safety
# Cryptic comment above is preserved for posterity. Future reader:
# consider change to condition below proposed in gh-1261/gh-17345.
if rhok_inv == 0.:
    rhok = 1000.0
    if disp:
```

```
    if not np.isfinite(old_fval):
        # We correctly found +-Inf as optimal value, or something went
        # wrong.
        warnflag = 2
        break

    rhok_inv = np.dot(yk, sk)
    # this was handled in numeric, let it remains for more safety
    # Cryptic comment above is preserved for posterity. Future reader:
    # consider change to condition below proposed in gh-1261/gh-17345.
    if rhok_inv == 0.:
        rhok = 1000.0
        if disp:
            msg = "Divide-by-zero encountered: rhok assumed large"
            _print_success_message_or_warn(True, msg)
    else:
        rhok = 1. / rhok_inv

    A1 = I - sk[:, np.newaxis] * yk[np.newaxis, :] * rhok
    A2 = I - yk[:, np.newaxis] * sk[np.newaxis, :] * rhok
    Hk = np.dot(A1, np.dot(Hk, A2)) + (rhok * sk[:, np.newaxis] *
                                       sk[np.newaxis, :])

fval = old_fval
```

```
    if not np.isfinite(old_fval):
        # We correctly found +-Inf as optimal value, or something went
        # wrong.
        warnflag = 2
        break
```

Calculate ρ_k

```
    rhok_inv = np.dot(yk, sk)
    # this was handled in numeric, let it remains for more safety
    # Cryptic comment above is preserved for posterity. Future reader:
    # consider change to condition below proposed in gh-1261/gh-17345.
    if rhok_inv == 0.:
        rhok = 1000.0
        if disp:
            msg = "Divide-by-zero encountered: rhok assumed large"
            _print_success_message_or_warn(True, msg)
    else:
        rhok = 1. / rhok_inv
```

```
    A1 = I - sk[:, np.newaxis] * yk[np.newaxis, :] * rhok
    A2 = I - yk[:, np.newaxis] * sk[np.newaxis, :] * rhok
    Hk = np.dot(A1, np.dot(Hk, A2)) + (rhok * sk[:, np.newaxis] *
                                       sk[np.newaxis, :])

fval = old_fval
```

```
    if not np.isfinite(old_fval):
        # We correctly found +-Inf as optimal value, or something went
        # wrong.
        warnflag = 2
        break

    rhok_inv = np.dot(yk, sk)
    # this was handled in numeric, let it remains for more safety
    # Cryptic comment above is preserved for posterity. Future reader:
    # consider change to condition below proposed in gh-1261/gh-17345.
    if rhok_inv == 0.:
        rhok = 1000.0
        if disp:
            msg = "Divide-by-zero encountered: rhok assumed large"
            _print_success_message_or_warn(True, msg)
    else:
        rhok = 1. / rhok_inv
```

Calculate H_{k+1}

```
    A1 = I - sk[:, np.newaxis] * yk[np.newaxis, :] * rhok
    A2 = I - yk[:, np.newaxis] * sk[np.newaxis, :] * rhok
    Hk = np.dot(A1, np.dot(Hk, A2)) + (rhok * sk[:, np.newaxis] *
                                       sk[np.newaxis, :])

fval = old_fval
```

```
    if not np.isfinite(old_fval):
        # We correctly found +-Inf as optimal value, or something went
        # wrong.
        warnflag = 2
        break

    rhok_inv = np.dot(yk, sk)
    # this was handled in numeric, let it remains for more safety
    # Cryptic comment above is preserved for posterity. Future reader:
    # consider change to condition below proposed in gh-1261/gh-17345.
    if rhok_inv == 0.:
        rhok = 1000.0
        if disp:
            msg = "Divide-by-zero encountered: rhok assumed large"
            _print_success_message_or_warn(True, msg)
    else:
        rhok = 1. / rhok_inv

    A1 = I - sk[:, np.newaxis] * yk[np.newaxis, :] * rhok
    A2 = I - yk[:, np.newaxis] * sk[np.newaxis, :] * rhok
    Hk = np.dot(A1, np.dot(Hk, A2)) + (rhok * sk[:, np.newaxis] *
                                       sk[np.newaxis, :])
```

```
fval = old_fval
```

End of iteration k

Choosing CG or Quasi-Newton Methods

- CG is a good choice when memory is **constrained** or N is **large enough** for quasi-Newton methods' $O(N^2)$ memory usage to be an issue.
- CG is a good choice when **a good guess for the starting point** is available.
- CG is a good choice when the numerical errors and other edge cases are **not a concern**.

- Quasi-Newton methods are a good choice in **compute-abundant scenarios**.
- Quasi-Newton methods are a good choice when the **starting point may be far away from the minimum**.
- Quasi-Newton methods are a good choice when **many iterations are acceptable**.
- Quasi-Newton methods are a good choice when the minimization technique **must be robust**.

Thanks!

Joshua Sheldon
jsheldon2022@my.fit.edu

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik** and content by **Swetha Tandri**