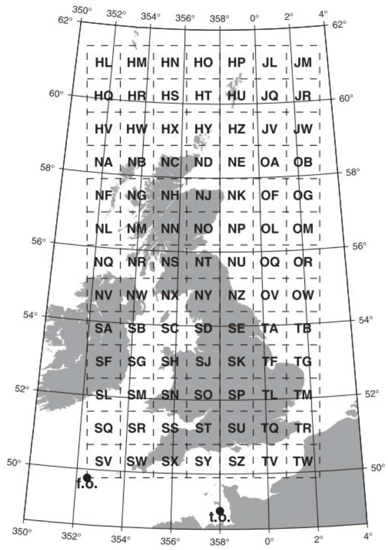
# Movable Type Scripts

## Convert between Latitude/Longitude & OS National Grid References

Some people have asked me about converting between latitude/longitude & Ordnance Survey grid references. The maths seems extraordinarily complex (and way beyond me!), but the Ordnance Survey explain the resulting formulae very clearly in Annex C of their *Guide to coordinate systems in Great Britain*.

OS Grid References are based on 100km grid squares identified by letter-pairs, followed by digits which identify a sub-square within the grid square, as explained on the OS *Interactive Guide to the National Grid*. 6-digit references identify 100m grid squares; 8 digits identify 10m grid squares, and 10 digits identify 1m squares. TG51401317 represents a 10m box with its (south-west) origin 51.40km across, 13.17km up within the TG square.



OS National Grid, with true origin & false origin

*Functional demo*

*Enter OS grid references or latitude/longitude values into the test boxes to try out the calculations*:

OS Grid Ref | TG 51409 13177 | ≡ | 651409,313177

Lat/Lon (WGS84) | 52° 39′ 28.72″ N | 001° 42′ 57.74″ E

(SW corner of grid square)

Lat/Lon (OSGB36) | 52° 39′ 27.25″ N | 001° 43′ 04.47″ E

no longer used (since 2014)

Display calculation results as: ◉ deg/min/sec ○ decimal degrees

An alternative way of expressing OS Grid References is as all-numeric eastings and northings, usually in metres. As square TG is six squares across, three squares up within the grid, grid reference TG 5140 1317 can also be expressed as 651400,313170.

As the surface of the earth is curved, and maps are flat, mapping involves a *projection* of the (ellipsoidal) curved surface onto a (plane) flat surface. The Ordnance Survey grid is a transverse Mercator projection.

### Latitude/longitudes require a datum

Before I started writing these geodesy scripts, I assumed a latitude/longitude point was a latitude/longitude point, and that was that. Since then, I have discovered that if you're being accurate, things get more complicated, and you need to know what *datum* you are working within.

Historically, cartographers worked at finding the ellipsoid which provided the closest mapping to the local *geoid*. The geoid is equivalent to the local mean-sea-level (or its equivalent in land-locked locales), and is determined by the local force of gravity. These ellipsoids were then fixed to local datums (fixed reference frames established through cartographic surveys). Simplifying enormously, this approach has now been broadly replaced by global geocentric ellipsoids fixed to global & continental datums (or reference frames) which can be readily mapped to each other to account for tectonic plate shifts. The best known global datum is WGS84, which is used by GPS systems; its more accurate siblings are (date-dependant) ITRFs. Continental datums include NAD-83 for North America, ETRS89 for Europe, GDA94 for Australia, etc.

Back in the 1930's, the UK Ordnance Survey defined 'OSGB-36' as the datum for the UK, based on the 'Airy 1830' ellipsoid. In 2014, they deprecated OSGB-36 in favour of WGS-84 for latitude/longitude coordinates, but OSGB-36 is still the basis for OS grid references. The Greenwich Observatory – historically the 'prime meridian' – is around 000°00′05″W on the WGS-84 datum (a difference of a bit over 100 metres): and moving every year. (I find the history of cartography facinating: I will put together a reading list sometime).

So to convert a (WGS84) latitude/longitude point to an OS grid reference, it must first be converted from the WGS84 datum to the OSGB36 datum, then have the transverse Mercator projection applied to transform it from a curved surface to a flat one.

### Datum conversions

To convert between datums, a 'Helmert transformation' is used. The Ordnance Survey explain the details in section 6 (and the annexes) of their *Guide to coordinate systems in Great Britain*.

The procedure is:

2. convert polar lat/long/height φ, λ, h to geocentric 'ECEF' cartesian co-ordinates x, y, z (using the source ellipsoid parameters *a*, *b*, *f*)
3. apply 7-parameter Helmert transformation; this applies a 3-dimensional shift & rotation, and a scale factor
4. convert cartesian co-ordinates back to polar lat/long/height (using the destination ellipsoid parameters)

**1 convert (geodetic) latitude/longitude to geocentric cartesian coordinates:**

$e^2 = (a^2 - b^2) / a^2$ *eccentricity of source ellipsoid (= 2·f − f²)*

$\nu = a / \sqrt{1 - e^2 \cdot \sin^2\varphi}$ *transverse radius of curvature*

$x = (\nu + h) \cdot \cos\varphi \cdot \cos\lambda$

$y = (\nu + h) \cdot \cos\varphi \cdot \sin\lambda$

$z = ((1 - e^2) \cdot \nu + h) \cdot \sin\varphi$

**2 apply Helmert transform**

The 7-parameter Helmert transform is given by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}' = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} 1+s & -r_z & r_y \\ r_z & 1+s & -r_x \\ -r_y & r_x & 1+s \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Hence:

$x' = t_x + (1+s) \cdot x - r_z \cdot y + r_y \cdot z$

$y' = t_y + r_z \cdot x + (1+s) \cdot y - r_x \cdot z$

$z' = t_z - r_y \cdot x + r_x \cdot y + (1+s) \cdot z$

**3 convert cartesian co-ordinates back to latitude/longitude**

While the conversion from geodetic to cartesian is straightforward, converting cartesian to geodetic is a complex problem.

There are a number of possible approaches: this uses Bowring's 1985 method (*The Accuracy of Geodetic Latitude and Height Equations – Survey Review Vol 28, 218, Oct 1985*), which provides approximately µm precision on the earth ellipsoid, in a concise formulation. If you have more demanding requirements, you could compare e.g. Fukushima (2006), Vermeille (2011), Karney (2012), and others.

$e^2 = (a^2 - b^2) / a^2$ *1st eccentricity of ellipsoid (= 2·f − f²)*

$\varepsilon^2 = (a^2 - b^2) / b^2$ *2nd eccentricity of ellipsoid (= e² / (1−e²))*

$\nu = a / \sqrt{1 - e^2 \cdot \sin^2\varphi}$ *length of normal terminated by minor axis*

$p = \sqrt{x^2 + y^2}$ *distance from minor axis*

$R = \sqrt{p^2 + z^2}$ *polar radius*

$\tan\beta = (b \cdot z)/(a \cdot p) \cdot (1 + \varepsilon^2 \cdot b/R)$ *parametric latitude* (17)

$\tan\varphi = (z + \varepsilon^2 \cdot b \cdot \sin^3\beta) / (p - e^2 \cdot a \cdot \cos^3\beta)$ *geodetic latitude* (18)

$\tan\lambda = y / x$ *longitude*

$h = p \cdot \cos\varphi + z \cdot \sin\varphi - a^2/\nu$ *height above ellipsoid* (7)

**Accuracy**: the Ordnance Survey say a single Helmert transformation between WGS84 and OSGB36 is accurate to within about 4–5 metres – for greater accuracy, a 'rubber-sheet' style transformation, which takes into account distortions in the 'terrestrial reference frame' (TRF), must be used ('~~OSTN02~~' 'OSTN15'): this is a different level of complexity, if you require such accuracy, I expect you already know more about OSGB transformations than me.

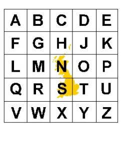
## Transverse Mercator projection

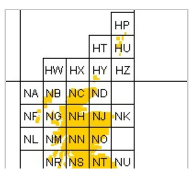
With the latitude/longitude in the OSGB-36 datum, a transverse Mercator projection is then applied to obtain a rectilinear grid. The OSGB uses the Lee–Redfearn implementation of the Gauss–Krüger projection.

The OSGB national grid uses a true origin of 49°N, 2°W. A false origin of −100 km north, 400 km east is then applied, so that eastings are always positive, and northings do not exceed 1,000 km.

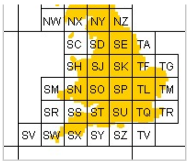
## OSGB national grid references

Aside from the transformation maths, the other tricky bit of the script is converting grid letter-pairs to/from numeric eastings & northings. To follow what's going on, it is worth noting that the letter-pairs define a 5x5 grid of 500km squares, each containing 5x5 sub-grids of 100km squares; the eastings & northings work from a 'false origin' at grid square SV, which is displaced from grid square AA by 10 squares E, 19 squares N, with the northing axis inverted; and letter 'I' is skipped.



OSGB Grid References apply to Great Britain only (Ireland and the Channel Islands have separate references).




### Example usage of OS-GridRef library

To convert an Ordnance Survey grid reference to a (WGS-84) latitude/longitude point:

```
<script type="module">
    import OsGridRef from 'https://cdn.jsdelivr.net/npm/geodesy@2/osgridref.js';
    const gridref = OsGridRef.parse('TL 44982 57869');
    const wgs84 = gridref.toLatLon();
    console.log(wgs84.toString('d', 2)); // '52.20°N, 000.12°E'
</script>
```

To convert a (WGS-84) latitude/longitude point to an Ordnance Survey grid reference:

```
<script type="module">
    import { LatLon } from 'https://cdn.jsdelivr.net/npm/geodesy@2/osgridref.js';
    const wgs84 = new LatLon(52.2, 0.12);
    const gridref = wgs84.toOsGrid();
    console.log(gridref.toString()); // 'TL 44982 57869'
</script>
```

See documentation for full details.

---

Distances between OS grid reference points are straightforward to calculate by pythagoras, once references are converted to numeric form. For UTM coordinates & MGRS grid references, see my UTM-MGRS page. For other scripts for calculating distances, bearings, etc between latitude/longitude points, see my Lat/Long page.

See below for the JavaScript source code of the transverse Mercator projection and the datum transformation, also available on GitHub. Full documentation is available, as well as a test suite.

Note I use Greek letters in variables representing maths symbols conventionally presented as Greek letters: I value the great benefit in legibility over the minor inconvenience in typing (if you encounter any problems, ensure your `<head>` includes `<meta charset="utf-8">`, and use UTF-8 encoding when saving files).

With its untyped C-style syntax, JavaScript reads remarkably close to pseudo-code: exposing the algorithms with a minimum of syntactic distractions. These functions should be simple to translate into other languages if required, though can also be used as-is in browsers and Node.js.

For convenience & clarity, I have extended the base JavaScript `Number` object with `toRadians()` and `toDegrees()` methods: I don't see great likelihood of conflicts, as these are ubiquitous operations.

I offer these scripts for free use and adaptation to balance my debt to the open-source info-verse. You are welcome to re-use these scripts [under an MIT licence, without any warranty express or implied] provided solely that you retain my copyright notice and a link to this page.

If you would like to show your appreciation and support continued development of these scripts, I would most gratefully accept donations.

If you need any advice or development work done, I am available for consultancy.

If you have any queries or find any problems, contact me at scripts-geo@movable-type.co.uk.

*© 2002-2022 Chris Veness*

---

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
 Survey Grid Reference functions                          (c) Chris Veness 2005-2021  */
                                                                          MIT Licence  */
ɔle-type.co.uk/scripts/latlong-gridref.html                                            */
ɔle-type.co.uk/scripts/geodesy-library.html#osgridref                                  */
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

ɔnEllipsoidal, { Dms } from './latlon-ellipsoidal-datum.js';



 Survey OSGB grid references provide geocoordinate references for UK mapping purposes.

ion implemented here due to Thomas, Redfearn, etc is as published by OS, but is inferior
- as used by e.g. Karney 2011.

ancesurvey.co.uk/documents/resources/guide-coordinate-systems-great-britain.pdf.

3 grid references cover Great Britain only; Ireland and the Channel Islands have their
rences.

: these formulae are based on ellipsoidal calculations, and according to the OS are
 to about 4-5 metres - for greater accuracy, a geoid-based transformation (OSTN15) must




d 2015 to work with WGS84 by default, OSGB36 as option;
ancesurvey.co.uk/blog/2014/12/confirmation-on-changes-to-latitude-and-longitude
```

```
F  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */


nalGrid = {
gin:  { lat: 49, lon: -2 },                  // true origin of grid 49°N,2°W on OSGB36 datum
igin: { easting: -400e3, northing: 100e3 }, // easting & northing of false origin, metres from true origin
ctor: 0.9996012717,                         // scale factor on central meridian
id:   LatLonEllipsoidal.ellipsoids.Airy1830,

sh National Grid uses t/o 53°30′N, 8°W, f/o 200kmW, 250kmS, scale factor 1.000035, on Airy 1830 Modified ellipsoid



References with methods to parse and convert them to latitude/longitude points.

dRef {


tes an OsGridRef object.

am {number} easting - Easting in metres from OS Grid false origin.
am {number} northing - Northing in metres from OS Grid false origin.

mple
port OsGridRef from '/js/geodesy/osgridref.js';
nst gridref = new OsGridRef(651409, 313177);

ctor(easting, northing) {
s.easting = Number(easting);
s.northing = Number(northing);

(isNaN(easting)  || this.easting<0  || this.easting>700e3) throw new RangeError(`invalid easting '${easting}'`);
(isNaN(northing) || this.northing<0 || this.northing>1300e3) throw new RangeError(`invalid northing '${northing}'`);




erts 'this' Ordnance Survey Grid Reference easting/northing coordinate to latitude/longitude
corner of grid square).

e OS Grid References are based on OSGB-36, the Ordnance Survey have deprecated the use of
-36 for latitude/longitude coordinates (in favour of WGS-84), hence this function returns
84 by default, with OSGB-36 as an option. See www.ordnancesurvey.co.uk/blog/2014/12/2.

 formulation implemented here due to Thomas, Redfearn, etc is as published by OS, but is
rior to Krüger as used by e.g. Karney 2011.

am   {LatLon.datum} [datum=WGS84] - Datum to convert grid reference into.
urns {LatLon}       Latitude/longitude of supplied grid reference.

mple
nst gridref = new OsGridRef(651409.903, 313177.270);
nst pWgs84 = gridref.toLatLon();                    // 52°39′28.723″N, 001°42′57.787″E
 to obtain (historical) OSGB36 lat/lon point:
nst pOsgb = gridref.toLatLon(LatLon.datums.OSGB36); // 52°39′27.253″N, 001°43′04.518″E

n(datum=LatLonEllipsoidal.datums.WGS84) {
st { easting: E, northing: N } = this;

st { a, b } = nationalGrid.ellipsoid;            // a = 6377563.396, b = 6356256.909
st φ0 = nationalGrid.trueOrigin.lat.toRadians(); // latitude of true origin, 49°N
st λ0 = nationalGrid.trueOrigin.lon.toRadians(); // longitude of true origin, 2°W
st E0 = -nationalGrid.falseOrigin.easting;       // easting of true origin, 400km
st N0 = -nationalGrid.falseOrigin.northing;      // northing of true origin, -100km
st F0 = nationalGrid.scaleFactor;                // 0.9996012717

st e2 = 1 - (b*b)/(a*a);                         // eccentricity squared
st n = (a-b)/(a+b), n2 = n*n, n3 = n*n*n;        // n, n², n³

 φ=φ0, M=0;
{
 φ = (N-N0-M)/(a*F0) + φ;

 const Ma = (1 + n + (5/4)*n2 + (5/4)*n3) * (φ-φ0);
 const Mb = (3*n + 3*n*n + (21/8)*n3) * Math.sin(φ-φ0) * Math.cos(φ+φ0);
 const Mc = ((15/8)*n2 + (15/8)*n3) * Math.sin(2*(φ-φ0)) * Math.cos(2*(φ+φ0));
 const Md = (35/24)*n3 * Math.sin(3*(φ-φ0)) * Math.cos(3*(φ+φ0));
 M = b * F0 * (Ma - Mb + Mc - Md);                // meridional arc

hile (Math.abs(N-N0-M) >= 0.00001);  // ie until < 0.01mm

st cosφ = Math.cos(φ), sinφ = Math.sin(φ);
st ν = a*F0/Math.sqrt(1-e2*sinφ*sinφ);            // nu = transverse radius of curvature
st ρ = a*F0*(1-e2)/Math.pow(1-e2*sinφ*sinφ, 1.5); // rho = meridional radius of curvature
st η2 = ν/ρ-1;                                    // eta = ?

st tanφ = Math.tan(φ);
st tan2φ = tanφ*tanφ, tan4φ = tan2φ*tan2φ, tan6φ = tan4φ*tan2φ;
st secφ = 1/cosφ;
st ν3 = ν*ν*ν, ν5 = ν3*ν*ν, ν7 = ν5*ν*ν;
st VII = tanφ/(2*ρ*ν);
st VIII = tanφ/(24*ρ*ν3)*(5+3*tan2φ+η2-9*tan2φ*η2);
st IX = tanφ/(720*ρ*ν5)*(61+90*tan2φ+45*tan4φ);
st X = secφ/ν;
st XI = secφ/(6*ν3)*(ν/ρ+2*tan2φ);
st XII = secφ/(120*ν5)*(5+28*tan2φ+24*tan4φ);
st XIIA = secφ/(5040*ν7)*(61+662*tan2φ+1320*tan4φ+720*tan6φ);

st dE = (E-E0), dE2 = dE*dE, dE3 = dE2*dE, dE4 = dE2*dE2, dE5 = dE3*dE2, dE6 = dE4*dE2, dE7 = dE5*dE2;
```

```
 ϕ - VII*dE2 + VIII*dE4 - IX*dE6;
st λ = λ0 + X*dE - XI*dE3 + XII*dE5 - XIIA*dE7;

 point = new LatLon_OsGridRef(ϕ.toDegrees(), λ.toDegrees(), 0, LatLonEllipsoidal.datums.OSGB36);

(datum != LatLonEllipsoidal.datums.OSGB36) {
 // if point is required in datum other than OSGB36, convert it
 point = point.convertDatum(datum);
 // convertDatum() gives us a LatLon: convert to LatLon_OsGridRef which includes toOsGrid()
 point = new LatLon_OsGridRef(point.lat, point.lon, point.height, point.datum);


urn point;




es grid reference to OsGridRef object.

pts standard grid references (eg 'SU 387 148'), with or without whitespace separators, from
digit references up to 10-digit references (1m × 1m square), or fully numeric comma-separated
rences in metres (eg '438700,114800').

am   {string}    gridref - Standard format OS Grid Reference.
urns {OsGridRef} Numeric version of grid reference in metres from false origin (SW corner of
pplied grid square).
ows  {Error}     Invalid grid reference.

mple
nst grid = OsGridRef.parse('TG 51409 13177'); // grid: { easting: 651409, northing: 313177 }

parse(gridref) {
dref = String(gridref).trim();

check for fully numeric comma-separated gridref format
 match = gridref.match(/^(\d+),\s*(\d+)$/);
(match) return new OsGridRef(match[1], match[2]);

validate format
ch = gridref.match(/^[HNST][ABCDEFGHJKLMNOPQRSTUVWXYZ]\s*[0-9]+\s*[0-9]+$/i);
(!match) throw new Error(`invalid grid reference '${gridref}'`);

get numeric values of letter references, mapping A->0, B->1, C->2, etc:
 l1 = gridref.toUpperCase().charCodeAt(0) - 'A'.charCodeAt(0); // 500km square
 l2 = gridref.toUpperCase().charCodeAt(1) - 'A'.charCodeAt(0); // 100km square
shuffle down letters after 'I' since 'I' is not used in grid:
(l1 > 7) l1--;
(l2 > 7) l2--;

convert grid letters into 100km-square indexes from false origin (grid square SV):
st e100km = ((l1 - 2) % 5) * 5 + (l2 % 5);
st n100km = (19 - Math.floor(l1 / 5) * 5) - Math.floor(l2 / 5);

skip grid letters to get numeric (easting/northing) part of ref
 en = gridref.slice(2).trim().split(/\s+/);
if e/n not whitespace separated, split half way
(en.length == 1) en = [ en[0].slice(0, en[0].length / 2), en[0].slice(en[0].length / 2) ];

validation
(en[0].length != en[1].length) throw new Error(`invalid grid reference '${gridref}'`);

standardise to 10-digit refs (metres)
0] = en[0].padEnd(5, '0');
1] = en[1].padEnd(5, '0');

st e = e100km + en[0];
st n = n100km + en[1];

urn new OsGridRef(e, n);




erts 'this' numeric grid reference to standard OS Grid Reference.

am   {number} [digits=10] - Precision of returned grid reference (10 digits = metres);
gits=0 will return grid reference in numeric format.
urns {string} This grid reference in standard format.

mple
nst gridref = new OsGridRef(651409, 313177).toString(8); // 'TG 5140 1317'
nst gridref = new OsGridRef(651409, 313177).toString(0); // '651409,313177'

g(digits=10) {
(![ 0,2,4,6,8,10,12,14,16 ].includes(Number(digits))) throw new RangeError(`invalid precision '${digits}'`); // eslint-disable-line comma-spacing

 { easting: e, northing: n } = this;

use digits = 0 to return numeric format (in metres) - note northing may be >= 1e7
(digits == 0) {
 const format = { useGrouping: false,  minimumIntegerDigits: 6, maximumFractionDigits: 3 };
 const ePad = e.toLocaleString('en', format);
 const nPad = n.toLocaleString('en', format);
 return `${ePad},${nPad}`;


get the 100km-grid indices
st e100km = Math.floor(e / 100000), n100km = Math.floor(n / 100000);
```

```
:ranslate those into numeric equivalents of the grid letters
 l1 = (19 - n100km) - (19 - n100km) % 5 + Math.floor((e100km + 10) / 5);
 l2 = (19 - n100km) * 5 % 25 + e100km % 5;

:ompensate for skipped 'I' and build grid letter-pairs
(l1 > 7) l1++;
(l2 > 7) l2++;
st letterPair = String.fromCharCode(l1 + 'A'.charCodeAt(0), l2 + 'A'.charCodeAt(0));

strip 100km-grid indices from easting & northing, and reduce precision
 Math.floor((e % 100000) / Math.pow(10, 5 - digits / 2));
 Math.floor((n % 100000) / Math.pow(10, 5 - digits / 2));

pad eastings & northings with leading zeros
 e.toString().padStart(digits/2, '0');
 n.toString().padStart(digits/2, '0');

urn `${letterPair} ${e} ${n}`;





sGridRef - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */



LatLon class with method to convert LatLon point to OS Grid Reference.

 LatLonEllipsoidal

n_OsGridRef extends LatLonEllipsoidal {


erts latitude/longitude to Ordnance Survey grid reference easting/northing coordinate.

urns {OsGridRef} OS Grid Reference easting/northing.

mple
nst grid = new LatLon(52.65798, 1.71605).toOsGrid(); // TG 51409 13177
 for conversion of (historical) OSGB36 latitude/longitude point:
nst grid = new LatLon(52.65798, 1.71605).toOsGrid(LatLon.datums.OSGB36);

d() {
if necessary convert to OSGB36 first
st point = this.datum == LatLonEllipsoidal.datums.OSGB36
 ? this
 : this.convertDatum(LatLonEllipsoidal.datums.OSGB36);

st φ = point.lat.toRadians();
st λ = point.lon.toRadians();

st { a, b } = nationalGrid.ellipsoid;            // a = 6377563.396, b = 6356256.909
st φ0 = nationalGrid.trueOrigin.lat.toRadians(); // latitude of true origin, 49°N
st λ0 = nationalGrid.trueOrigin.lon.toRadians(); // longitude of true origin, 2°W
st E0 = -nationalGrid.falseOrigin.easting;       // easting of true origin, 400km
st N0 = -nationalGrid.falseOrigin.northing;      // northing of true origin, -100km
st F0 = nationalGrid.scaleFactor;                // 0.9996012717

st e2 = 1 - (b*b)/(a*a);                          // eccentricity squared
st n = (a-b)/(a+b), n2 = n*n, n3 = n*n*n;         // n, n², n³

st cosφ = Math.cos(φ), sinφ = Math.sin(φ);
st ν = a*F0/Math.sqrt(1-e2*sinφ*sinφ);            // nu = transverse radius of curvature
st ρ = a*F0*(1-e2)/Math.pow(1-e2*sinφ*sinφ, 1.5); // rho = meridional radius of curvature
st η2 = ν/ρ-1;                                    // eta = ?

st Ma = (1 + n + (5/4)*n2 + (5/4)*n3) * (φ-φ0);
st Mb = (3*n + 3*n*n + (21/8)*n3) * Math.sin(φ-φ0) * Math.cos(φ+φ0);
st Mc = ((15/8)*n2 + (15/8)*n3) * Math.sin(2*(φ-φ0)) * Math.cos(2*(φ+φ0));
st Md = (35/24)*n3 * Math.sin(3*(φ-φ0)) * Math.cos(3*(φ+φ0));
st M = b * F0 * (Ma - Mb + Mc - Md);              // meridional arc

st cos3φ = cosφ*cosφ*cosφ;
st cos5φ = cos3φ*cosφ*cosφ;
st tan2φ = Math.tan(φ)*Math.tan(φ);
st tan4φ = tan2φ*tan2φ;

st I = M + N0;
st II = (ν/2)*sinφ*cosφ;
st III = (ν/24)*sinφ*cos3φ*(5-tan2φ+9*η2);
st IIIA = (ν/720)*sinφ*cos5φ*(61-58*tan2φ+tan4φ);
st IV = ν*cosφ;
st V = (ν/6)*cos3φ*(ν/ρ-tan2φ);
st VI = (ν/120) * cos5φ * (5 - 18*tan2φ + tan4φ + 14*η2 - 58*tan2φ*η2);

st Δλ = λ-λ0;
st Δλ2 = Δλ*Δλ, Δλ3 = Δλ2*Δλ, Δλ4 = Δλ3*Δλ, Δλ5 = Δλ4*Δλ, Δλ6 = Δλ5*Δλ;

 N = I + II*Δλ2 + III*Δλ4 + IIIA*Δλ6;
 E = E0 + IV*Δλ + V*Δλ3 + VI*Δλ5;

 Number(N.toFixed(3)); // round to mm precision
 Number(E.toFixed(3));

 {
 return new OsGridRef(E, N); // note: gets truncated to SW corner of 1m grid square
atch (e) {
 throw new Error(`${e.message} from (${point.lat.toFixed(6)},${point.lon.toFixed(6)}).toOsGrid()`);
```

```
ride LatLonEllipsoidal.convertDatum() with version which returns LatLon_OsGridRef.

Datum(toDatum) {
st osgbED = super.convertDatum(toDatum); // returns LatLonEllipsoidal_Datum
st osgbOSGR = new LatLon_OsGridRef(osgbED.lat, osgbED.lon, osgbED.height, osgbED.datum);
urn osgbOSGR;




- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

GridRef as default, LatLon_OsGridRef as LatLon, Dms };
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
tools for an ellipsoidal earth model                       (c) Chris Veness 2005-2022  */
                                                                          MIT Licence  */
ss for latlon-ellipsoidal-datum & latlon-ellipsoidal-referenceframe.                   */
                                                                                       */
ble-type.co.uk/scripts/latlong-convert-coords.html                                     */
ble-type.co.uk/scripts/geodesy-library.html#latlon-ellipsoidal                         */
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

     from './dms.js';
or3d from './vector3d.js';



de/longitude point defines a geographic location on or above/below the earth's surface,
 in degrees from the equator & the International Reference Meridian and in metres above
psoid, and based on a given datum.

gh modern geodesy is based on WGS-84 (as used by GPS), this module includes WGS-84
d parameters, and it has methods for converting geodetic (latitude/longitude) points to/from
ic cartesian points; the latlon-ellipsoidal-datum and latlon-ellipsoidal-referenceframe
provide transformation parameters for converting between historical datums and between
eference frames.

ule is used for both trigonometric geodesy (eg latlon-ellipsoidal-vincenty) and n-vector
(eg latlon-nvector-ellipsoidal), and also for UTM/MGRS mapping.

latlon-ellipsoidal




d parameters; exposed through static getter below.

 ellipsoid defined is WGS84, for use in utm/mgrs, vincenty, nvector.

soids = {
{ a: 6378137, b: 6356752.314245, f: 1/298.257223563 },




exposed through static getter below.

 datum defined is WGS84, for use in utm/mgrs, vincenty, nvector.

s = {
{ ellipsoid: ellipsoids.WGS84 },



tatic properties
ze(ellipsoids.WGS84);
ze(datums.WGS84);


lipsoidal - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */



/longitude points on an ellipsoidal model earth, with ellipsoid parameters and methods
erting points to/from cartesian (ECEF) coordinates.

the core class, which will usually be used via LatLonEllipsoidal_Datum or
lipsoidal_ReferenceFrame.

nEllipsoidal {


tes a geodetic latitude/longitude point on a (WGS84) ellipsoidal model earth.

am  {number} lat - Latitude (in degrees).
am  {number} lon - Longitude (in degrees).
am  {number} [height=0] - Height above ellipsoid in metres.
ows {TypeError} Invalid lat/lon/height.

mple
```

```
oort LatLon from '/js/geodesy/latlon-ellipsoidal.js';
nst p = new LatLon(51.47788, -0.00147, 17);

:tor(lat, lon, height=0) {
(isNaN(lat) || lat == null) throw new TypeError(`invalid lat '${lat}'`);
(isNaN(lon) || lon == null) throw new TypeError(`invalid lon '${lon}'`);
(isNaN(height) || height == null) throw new TypeError(`invalid height '${height}'`);

s._lat = Dms.wrap90(Number(lat));
s._lon = Dms.wrap180(Number(lon));
s._height = Number(height);



:ude in degrees north from equator (including aliases lat, latitude): can be set as
ric or hexagesimal (deg-min-sec); returned as numeric.

()       { return this._lat; }
itude()  { return this._lat; }
(lat) {
s._lat = isNaN(lat) ? Dms.wrap90(Dms.parse(lat)) : Dms.wrap90(Number(lat));
(isNaN(this._lat)) throw new TypeError(`invalid lat '${lat}'`);

itude(lat) {
s._lat = isNaN(lat) ? Dms.wrap90(Dms.parse(lat)) : Dms.wrap90(Number(lat));
(isNaN(this._lat)) throw new TypeError(`invalid latitude '${lat}'`);


itude in degrees east from international reference meridian (including aliases lon, lng,
itude): can be set as numeric or hexagesimal (deg-min-sec); returned as numeric.

()       { return this._lon; }
()       { return this._lon; }
gitude() { return this._lon; }
(lon) {
s._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(Number(lon));
(isNaN(this._lon)) throw new TypeError(`invalid lon '${lon}'`);

(lon) {
s._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(Number(lon));
(isNaN(this._lon)) throw new TypeError(`invalid lng '${lon}'`);

gitude(lon) {
s._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(Number(lon));
(isNaN(this._lon)) throw new TypeError(`invalid longitude '${lon}'`);


nt in metres above ellipsoid.

ght()       { return this._height; }
ght(height) { this._height = Number(height); if (isNaN(this._height)) throw new TypeError(`invalid height '${height}'`); }


n.

 this is replicated within LatLonEllipsoidal in order that a LatLonEllipsoidal object can
onkey-patched to look like a LatLonEllipsoidal_Datum, for Vincenty calculations on
erent ellipsoids.

vate

um()      { return this._datum; }
um(datum) { this._datum = datum; }


osoids with their parameters; this module only defines WGS84 parameters a = 6378137, b =
752.314245, f = 1/298.257223563.

mple
nst a = LatLon.ellipsoids.WGS84.a; // 6378137

get ellipsoids() {
urn ellipsoids;


ms; this module only defines WGS84 datum, hence no datum transformations.

mple
nst a = LatLon.datums.WGS84.ellipsoid.a; // 6377563.396

get datums() {
urn datums;



es a latitude/longitude point from a variety of formats.

:ude & longitude (in degrees) can be supplied as two separate parameters, as a single
a-separated lat/lon string, or as a single object with { lat, lon } or GeoJSON properties.

latitude/longitude values may be numeric or strings; they may be signed decimal or
min-sec (hexagesimal) suffixed by compass direction (NSEW); a variety of separators are
```

```
oted. Examples -3.62, '3 37 12W', '3°37′12″W'.

sands/decimal separators must be comma/dot; use Dms.fromLocale to convert locale-specific
sands/decimal separators.

am   {number|string|Object} lat|latlon - Latitude (in degrees), or comma-separated lat/lon, or lat/lon object.
am   {number}               [lon]      - Longitude (in degrees).
am   {number}               [height=0] - Height above ellipsoid in metres.
urns {LatLon} Latitude/longitude point on WGS84 ellipsoidal model earth.
ows  {TypeError} Invalid coordinate.

mple
nst p1 = LatLon.parse(51.47788, -0.00147);              // numeric pair
nst p2 = LatLon.parse('51°28′40″N, 000°00′05″W', 17);   // dms string + height
nst p3 = LatLon.parse({ lat: 52.205, lon: 0.119 }, 17); // { lat, lon } object numeric + height

parse(...args) {
(args.length == 0) throw new TypeError('invalid (empty) point');

 lat=undefined, lon=undefined, height=undefined;

single { lat, lon } object
(typeof args[0]=='object' && (args.length==1 || !isNaN(parseFloat(args[1])))) {
 const ll = args[0];
 if (ll.type == 'Point' && Array.isArray(ll.coordinates)) { // GeoJSON
     [ lon, lat, height ] = ll.coordinates;
     height = height || 0;
 } else { // regular { lat, lon } object
     if (ll.latitude  != undefined) lat = ll.latitude;
     if (ll.lat       != undefined) lat = ll.lat;
     if (ll.longitude != undefined) lon = ll.longitude;
     if (ll.lng       != undefined) lon = ll.lng;
     if (ll.lon       != undefined) lon = ll.lon;
     if (ll.height    != undefined) height = ll.height;
     lat = Dms.wrap90(Dms.parse(lat));
     lon = Dms.wrap180(Dms.parse(lon));
 }
 if (args[1] != undefined) height = args[1];
 if (isNaN(lat) || isNaN(lon)) throw new TypeError(`invalid point '${JSON.stringify(args[0])}'`);


single comma-separated lat/lon
(typeof args[0] == 'string' && args[0].split(',').length == 2) {
 [ lat, lon ] = args[0].split(',');
 lat = Dms.wrap90(Dms.parse(lat));
 lon = Dms.wrap180(Dms.parse(lon));
 height = args[1] || 0;
 if (isNaN(lat) || isNaN(lon)) throw new TypeError(`invalid point '${args[0]}'`);


regular (lat, lon) arguments
(lat==undefined && lon==undefined) {
 [ lat, lon ] = args;
 lat = Dms.wrap90(Dms.parse(lat));
 lon = Dms.wrap180(Dms.parse(lon));
 height = args[2] || 0;
 if (isNaN(lat) || isNaN(lon)) throw new TypeError(`invalid point '${args.toString()}'`);


urn new this(lat, lon, height); // 'new this' as may return subclassed types




erts 'this' point from (geodetic) latitude/longitude coordinates to (geocentric)
esian (x/y/z) coordinates.

urns {Cartesian} Cartesian point equivalent to lat/lon point, with x, y, z in metres from
rth centre.

sian() {
x = (ν+h)⋅cosφ⋅cosλ, y = (ν+h)⋅cosφ⋅sinλ, z = (ν⋅(1-e²)+h)⋅sinφ
where ν = a/√(1−e²⋅sinφ⋅sinφ), e² = (a²-b²)/a² or (better conditioned) 2⋅f-f²
st ellipsoid = this.datum
 ? this.datum.ellipsoid
 : this.referenceFrame ? this.referenceFrame.ellipsoid : ellipsoids.WGS84;

st φ = this.lat.toRadians();
st λ = this.lon.toRadians();
st h = this.height;
st { a, f } = ellipsoid;

st sinφ = Math.sin(φ), cosφ = Math.cos(φ);
st sinλ = Math.sin(λ), cosλ = Math.cos(λ);

st eSq = 2*f - f*f;                      // 1st eccentricity squared ≡ (a²-b²)/a²
st ν = a / Math.sqrt(1 - eSq*sinφ*sinφ); // radius of curvature in prime vertical

st x = (ν+h) * cosφ * cosλ;
st y = (ν+h) * cosφ * sinλ;
st z = (ν*(1-eSq)+h) * sinφ;

urn new Cartesian(x, y, z);




ks if another point is equal to 'this' point.
```

```
am   {LatLon} point - Point to be compared against this point.
urns {bool} True if points have identical latitude, longitude, height, and datum/referenceFrame.
ows  {TypeError} Invalid point.

mple
nst p1 = new LatLon(52.205, 0.119);
nst p2 = new LatLon(52.205, 0.119);
nst equal = p1.equals(p2); // true

point) {
(!(point instanceof LatLonEllipsoidal)) throw new TypeError(`invalid point '${point}'`);

(Math.abs(this.lat - point.lat) > Number.EPSILON) return false;
(Math.abs(this.lon - point.lon) > Number.EPSILON) return false;
(Math.abs(this.height - point.height) > Number.EPSILON) return false;
(this.datum != point.datum) return false;
(this.referenceFrame != point.referenceFrame) return false;
(this.epoch != point.epoch) return false;

urn true;




rns a string representation of 'this' point, formatted as degrees, degrees+minutes, or
ees+minutes+seconds.

am   {string} [format=d] - Format point as 'd', 'dm', 'dms', or 'n' for signed numeric.
am   {number} [dp=4|2|0] - Number of decimal places to use: default 4 for d, 2 for dm, 0 for dms.
am   {number} [dpHeight=null] - Number of decimal places to use for height; default is no height display.
urns {string} Comma-separated formatted latitude/longitude.
ows  {RangeError} Invalid format.

mple
nst greenwich = new LatLon(51.47788, -0.00147, 46);
nst d = greenwich.toString();                        // 51.4779°N, 000.0015°W
nst dms = greenwich.toString('dms', 2);              // 51°28′40″N, 000°00′05″W
nst [lat, lon] = greenwich.toString('n').split(','); // 51.4779, -0.0015
nst dmsh = greenwich.toString('dms', 0, 0);          // 51°28′40″N, 000°00′06″W +46m

g(format='d', dp=undefined, dpHeight=null) {
note: explicitly set dp to undefined for passing through to toLat/toLon
(![ 'd', 'dm', 'dms', 'n' ].includes(format)) throw new RangeError(`invalid format '${format}'`);

st height = (this.height>=0 ? ' +' : ' ') + this.height.toFixed(dpHeight) + 'm';
(format == 'n') { // signed numeric degrees
 if (dp == undefined) dp = 4;
 const lat = this.lat.toFixed(dp);
 const lon = this.lon.toFixed(dp);
 return `${lat}, ${lon}${dpHeight==null ? '' : height}`;


st lat = Dms.toLat(this.lat, format, dp);
st lon = Dms.toLon(this.lon, format, dp);

urn `${lat}, ${lon}${dpHeight==null ? '' : height}`;





n  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */



rth-centered earth-fixed) geocentric cartesian coordinates.

 Vector3d

sian extends Vector3d {


tes cartesian coordinate representing ECEF (earth-centric earth-fixed) point.

am {number} x - X coordinate in metres (=> 0°N,0°E).
am {number} y - Y coordinate in metres (=> 0°N,90°E).
am {number} z - Z coordinate in metres (=> 90°N).

mple
port { Cartesian } from '/js/geodesy/latlon-ellipsoidal.js';
nst coord = new Cartesian(3980581.210, -111.159, 4966824.522);

ctor(x, y, z) {
er(x, y, z); // arguably redundant constructor, but specifies units & axes




erts 'this' (geocentric) cartesian (x/y/z) coordinate to (geodetic) latitude/longitude
t on specified ellipsoid.

 Bowring's (1985) formulation for μm precision in concise form; 'The accuracy of geodetic
tude and height equations', B R Bowring, Survey Review vol 28, 218, Oct 1985.

am   {LatLon.ellipsoids} [ellipsoid=WGS84] - Ellipsoid to use when converting point.
urns {LatLon} Latitude/longitude point defined by cartesian coordinates, on given ellipsoid.
ows  {TypeError} Invalid ellipsoid.

mple
```

```
ıst c = new Cartesian(4027893.924, 307041.993, 4919474.294);
ıst p = c.toLatLon(); // 50.7978°N, 004.3592°E

ı(ellipsoid=ellipsoids.WGS84) {
ıote ellipsoid is available as a parameter for when toLatLon gets subclassed to
Ellipsoidal_Datum / Ellipsoidal_Referenceframe.
(!ellipsoid || !ellipsoid.a) throw new TypeError(`invalid ellipsoid '${ellipsoid}'`);

st { x, y, z } = this;
st { a, b, f } = ellipsoid;

st e2 = 2*f - f*f;           // 1st eccentricity squared ≡ (a²−b²)/a²
st ε2 = e2 / (1-e2);         // 2nd eccentricity squared ≡ (a²−b²)/b²
st p = Math.sqrt(x*x + y*y); // distance from minor axis
st R = Math.sqrt(p*p + z*z); // polar radius

parametric latitude (Bowring eqn.17, replacing tanβ = z·a / p·b)
st tanβ = (b*z)/(a*p) * (1+ε2*b/R);
st sinβ = tanβ / Math.sqrt(1+tanβ*tanβ);
st cosβ = sinβ / tanβ;

geodetic latitude (Bowring eqn.18: tanφ = z+ε²·b·sin³β / p−e²·cos³β)
st φ = isNaN(cosβ) ? 0 : Math.atan2(z + ε2*b*sinβ*sinβ*sinβ, p - e2*a*cosβ*cosβ*cosβ);

longitude
st λ = Math.atan2(y, x);

height above ellipsoid (Bowring eqn.7)
st sinφ = Math.sin(φ), cosφ = Math.cos(φ);
st ν = a / Math.sqrt(1-e2*sinφ*sinφ); // length of the normal terminated by the minor axis
st h = p*cosφ + z*sinφ - (a*a/ν);

st point = new LatLonEllipsoidal(φ.toDegrees(), λ.toDegrees(), h);

urn point;




rns a string representation of 'this' cartesian point.

am   {number} [dp=0] - Number of decimal places to use.
urns {string} Comma-separated latitude/longitude.

g(dp=0) {
st x = this.x.toFixed(dp), y = this.y.toFixed(dp), z = this.z.toFixed(dp);
urn `[${x},${y},${z}]`;




- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

tLonEllipsoidal as default, Cartesian, Vector3d, Dms };
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
tools for conversions between (historical) datums          (c) Chris Veness 2005-2019  */
                                                                           MIT Licence  */
ble-type.co.uk/scripts/latlong-convert-coords.html                                      */
ble-type.co.uk/scripts/geodesy-library.html#latlon-ellipsoidal-datum                   */
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

onEllipsoidal, { Cartesian, Dms } from './latlon-ellipsoidal.js';



al geodetic datums: a latitude/longitude point defines a geographic location on or
low the  earth's surface, measured in degrees from the equator & the International
e Meridian and metres above the ellipsoid, and based on a given datum. The datum is
 a reference ellipsoid and tied to geodetic survey reference points.

eodesy is generally based on the WGS84 datum (as used for instance by GPS systems), but
ly various reference ellipsoids and datum references were used.

ule extends the core latlon-ellipsoidal module to include ellipsoid parameters and datum
mation parameters, and methods for converting between different (generally historical)


e used for UK Ordnance Survey mapping (OS National Grid References are still based on the
e historical OSGB36 datum), as well as for historical purposes.

nance Survey 'A guide to coordinate systems in Great Britain' Section 6,
ancesurvey.co.uk/docs/support/guide-coordinate-systems-great-britain.pdf, and also
ancesurvey.co.uk/blog/2014/12/2.

latlon-ellipsoidal-datum




d parameters; exposed through static getter below.

soids = {
        { a: 6378137,     b: 6356752.314245, f: 1/298.257223563 },
):      { a: 6377563.396, b: 6356256.909,    f: 1/299.3249646   },
ified:  { a: 6377340.189, b: 6356034.448,    f: 1/299.3249646   },
```

```
341:    { a: 6377397.155, b: 6356078.962818, f: 1/299.1528128   },
366:    { a: 6378206.4,   b: 6356583.8,      f: 1/294.978698214 },
380IGN: { a: 6378249.2,   b: 6356515.0,      f: 1/293.466021294 },
        { a: 6378137,     b: 6356752.314140, f: 1/298.257222101 },
4:      { a: 6378388,     b: 6356911.946,    f: 1/297           }, // aka Hayford
        { a: 6378135,     b: 6356750.5,      f: 1/298.26        },



exposed through static getter below.

s = {
sforms: t in metres, s in ppm, r in arcseconds              tx       ty        tz       s        rx        ry        rz
     { ellipsoid: ellipsoids.Intl1924,      transform: [   89.5,    93.8,    123.1,    -1.2,     0.0,      0.0,      0.156    ] }, // epsg.io/1311
     { ellipsoid: ellipsoids.GRS80,         transform: [    0,       0,        0,       0,       0,        0,        0        ] }, // epsg.io/1149; @ 1-metre level
:    { ellipsoid: ellipsoids.AiryModified,  transform: [ -482.530, 130.596, -564.557,  -8.150,   1.042,    0.214,    0.631    ] }, // epsg.io/1954
     { ellipsoid: ellipsoids.Clarke1866,    transform: [    8,    -160,     -176,       0,       0,        0,        0        ] },
     { ellipsoid: ellipsoids.GRS80,         transform: [    0.9956, -1.9103,  -0.5215, -0.00062, 0.025915, 0.009426, 0.011599 ] },
     { ellipsoid: ellipsoids.Clarke1880IGN, transform: [  168,      60,     -320,       0,       0,        0,        0        ] },
     { ellipsoid: ellipsoids.Airy1830,      transform: [ -446.448, 125.157, -542.060,  20.4894, -0.1502,  -0.2470,  -0.8421   ] }, // epsg.io/1314
:    { ellipsoid: ellipsoids.Bessel1841,    transform: [ -582,    -105,     -414,      -8.3,     1.04,     0.35,    -3.08     ] },
pan: { ellipsoid: ellipsoids.Bessel1841,    transform: [  148,    -507,     -685,       0,       0,        0,        0        ] },
     { ellipsoid: ellipsoids.WGS72,         transform: [    0,       0,       -4.5,    -0.22,    0,        0,        0.554    ] },
     { ellipsoid: ellipsoids.WGS84,         transform: [    0.0,     0.0,      0.0,     0.0,     0.0,      0.0,      0.0      ] },


      www.gov.uk/guidance/oil-and-gas-petroleum-operations-notices#pon-4
5:    www.osi.ie/wp-content/uploads/2015/05/transformations_booklet.pdf
      en.wikipedia.org/wiki/Helmert_transformation
      www.uvm.edu/giv/resources/WGS84_NAD83.pdf [strictly, WGS84(G1150) -> NAD83(CORS96) @ epoch 1997.0]
      (note NAD83(1986) ≡ WGS84(Original); confluence.qps.nl/pages/viewpage.action?pageId=29855173)
      Nouvelle Triangulation Francaise geodesie.ign.fr/contenu/fichiers/Changement_systeme_geodesique.pdf
:     www.ordnancesurvey.co.uk/docs/support/guide-coordinate-systems-great-britain.pdf
n:    kartoweb.itc.nl/geometrics/Coordinate%20transformations/coordtrans.html
apan: www.geocachingtoolbox.com?page=datumEllipsoidDetails
      www.icao.int/safety/pbn/documentation/eurocontrol/eurocontrol wgs 84 implementation manual.pdf

nsform parameters are available from earth-info.nga.mil/GandG/coordsys/datums/NATO_DT.pdf,
denmaps.info/cconv/web/cconv_params.js


 reference frames are coincident with WGS-84 at epoch 1989.0 (ie null transform) at the one metre level.



tatic properties
(ellipsoids).forEach(e => Object.freeze(ellipsoids[e]));
(datums).forEach(d => { Object.freeze(datums[d]); Object.freeze(datums[d].transform); });


 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */



/longitude points on an ellipsoidal model earth, with ellipsoid parameters and methods
erting between datums and to geocentric (ECEF) cartesian coordinates.

 LatLonEllipsoidal

nEllipsoidal_Datum extends LatLonEllipsoidal {


tes a geodetic latitude/longitude point on an ellipsoidal model earth using given datum.

am {number} lat - Latitude (in degrees).
am {number} lon - Longitude (in degrees).
am {number} [height=0] - Height above ellipsoid in metres.
am {LatLon.datums} datum - Datum this point is defined within.

mple
port LatLon from '/js/geodesy/latlon-ellipsoidal-datum.js';
nst p = new LatLon(53.3444, -6.2577, 17, LatLon.datums.Irl1975);

ctor(lat, lon, height=0, datum=datums.WGS84) {
(!datum || datum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${datum}'`);

er(lat, lon, height);

s._datum = datum;




n this point is defined within.

um() {
urn this._datum;




psoids with their parameters; semi-major axis (a), semi-minor axis (b), and flattening (f).

tening f = (a−b)/a; at least one of these parameters is derived from defining constants.

mple
nst a = LatLon.ellipsoids.Airy1830.a; // 6377563.396
```

```
jet ellipsoids() {
urn ellipsoids;



ns; with associated ellipsoid, and Helmert transform parameters to convert from WGS-84
 given datum.

 that precision of various datums will vary, and WGS-84 (original) is not defined to be
rate to better than ±1 metre. No transformation should be assumed to be accurate to
er than a metre, for many datums somewhat less.

 is a small sample of commoner datums from a large set of historical datums. I will add
datums on request.

nple
nst a = LatLon.datums.OSGB36.ellipsoid.a;                    // 6377563.396
nst tx = LatLon.datums.OSGB36.transform;                     // [ tx, ty, tz, s, rx, ry, rz ]
nst availableDatums = Object.keys(LatLon.datums).join(', '); // ED50, Irl1975, NAD27, ...

jet datums() {
urn datums;



 instance datum getter/setters are in LatLonEllipsoidal



es a latitude/longitude point from a variety of formats.

tude & longitude (in degrees) can be supplied as two separate parameters, as a single
a-separated lat/lon string, or as a single object with { lat, lon } or GeoJSON properties.

latitude/longitude values may be numeric or strings; they may be signed decimal or
min-sec (hexagesimal) suffixed by compass direction (NSEW); a variety of separators are
pted. Examples -3.62, '3 37 12W', '3°37′12″W'.

sands/decimal separators must be comma/dot; use Dms.fromLocale to convert locale-specific
sands/decimal separators.

am   {number|string|Object} lat|latlon - Geodetic Latitude (in degrees) or comma-separated lat/lon or lat/lon object.
am   {number}               [lon] - Longitude in degrees.
am   {number}               [height=0] - Height above ellipsoid in metres.
am   {LatLon.datums}        [datum=WGS84] - Datum this point is defined within.
urns {LatLon} Latitude/longitude point on ellipsoidal model earth using given datum.
ows  {TypeError} Unrecognised datum.

nple
nst p = LatLon.parse('51.47736, 0.0000', 0, LatLon.datums.OSGB36);

parse(...args) {
 datum = datums.WGS84;

if the last argument is a datum, use that, otherwise use default WGS-84
(args.length==4 || (args.length==3 && typeof args[2] == 'object')) datum = args.pop();

(!datum || datum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${datum}'`);

st point = super.parse(...args);

nt._datum = datum;

urn point;




erts 'this' lat/lon coordinate to new coordinate system.

am   {LatLon.datums} toDatum - Datum this coordinate is to be converted to.
urns {LatLon} This point converted to new datum.
ows  {TypeError} Unrecognised datum.

nple
nst pWGS84 = new LatLon(51.47788, -0.00147, 0, LatLon.datums.WGS84);
nst pOSGB = pWGS84.convertDatum(LatLon.datums.OSGB36); // 51.4773°N, 000.0001°E

Datum(toDatum) {
(!toDatum || toDatum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${toDatum}'`);

st oldCartesian = this.toCartesian();                 // convert geodetic to cartesian
st newCartesian = oldCartesian.convertDatum(toDatum); // convert datum
st newLatLon = newCartesian.toLatLon();               // convert cartesian back to geodetic

urn newLatLon;




erts 'this' point from (geodetic) latitude/longitude coordinates to (geocentric) cartesian
/z) coordinates, based on the same datum.

ow of LatLonEllipsoidal.toCartesian(), returning Cartesian augmented with
onEllipsoidal_Datum methods/properties.

urns {Cartesian} Cartesian point equivalent to lat/lon point, with x, y, z in metres from
rth centre, augmented with reference frame conversion methods and properties.
```

```
sian() {
st cartesian = super.toCartesian();
st cartesianDatum = new Cartesian_Datum(cartesian.x, cartesian.y, cartesian.z, this.datum);
urn cartesianDatum;




n  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */



 Cartesian with datum the cooordinate is based on, and methods to convert between datums
elmert 7-parameter transforms) and to convert cartesian to geodetic latitude/longitude


 Cartesian

sian_Datum extends Cartesian {


tes cartesian coordinate representing ECEF (earth-centric earth-fixed) point, on a given
n. The datum will identify the primary meridian (for the x-coordinate), and is also
ul in transforming to/from geodetic (lat/lon) coordinates.

am  {number} x - X coordinate in metres (=> 0°N,0°E).
am  {number} y - Y coordinate in metres (=> 0°N,90°E).
am  {number} z - Z coordinate in metres (=> 90°N).
am  {LatLon.datums} [datum] - Datum this coordinate is defined within.
ows {TypeError} Unrecognised datum.

mple
port { Cartesian } from '/js/geodesy/latlon-ellipsoidal-datum.js';
nst coord = new Cartesian(3980581.210, -111.159, 4966824.522);

ctor(x, y, z, datum=undefined) {
(datum && datum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${datum}'`);

er(x, y, z);

(datum) this._datum = datum;




n this point is defined within.

um() {
urn this._datum;

um(datum) {
(!datum || datum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${datum}'`);
s._datum = datum;




erts 'this' (geocentric) cartesian (x/y/z) coordinate to (geodetic) latitude/longitude
t (based on the same datum, or WGS84 if unset).

ow of Cartesian.toLatLon(), returning LatLon augmented with LatLonEllipsoidal_Datum
ods convertDatum, toCartesian, etc.

urns {LatLon} Latitude/longitude point defined by cartesian coordinates.
ows  {TypeError} Unrecognised datum

mple
nst c = new Cartesian(4027893.924, 307041.993, 4919474.294);
nst p = c.toLatLon(); // 50.7978°N, 004.3592°E

n(deprecatedDatum=undefined) {
(deprecatedDatum) {
 console.info('datum parameter to Cartesian_Datum.toLatLon is deprecated: set datum before calling toLatLon()');
 this.datum = deprecatedDatum;

st datum = this.datum || datums.WGS84;
(!datum || datum.ellipsoid==undefined) throw new TypeError(`unrecognised datum '${datum}'`);

st latLon = super.toLatLon(datum.ellipsoid); // TODO: what if datum is not geocentric?
st point = new LatLonEllipsoidal_Datum(latLon.lat, latLon.lon, latLon.height, this.datum);
urn point;




erts 'this' cartesian coordinate to new datum using Helmert 7-parameter transformation.

am   {LatLon.datums} toDatum - Datum this coordinate is to be converted to.
urns {Cartesian} This point converted to new datum.
ows  {Error} Undefined datum.

mple
nst c = new Cartesian(3980574.247, -102.127, 4966830.065, LatLon.datums.OSGB36);
convertDatum(LatLon.datums.Irl1975); // [??,??,??]

Datum(toDatum) {
TODO: what if datum is not geocentric?
```

```
(!toDatum || toDatum.ellipsoid == undefined) throw new TypeError(`unrecognised datum '${toDatum}'`);
(!this.datum) throw new TypeError('cartesian coordinate has no datum');

 oldCartesian = null;
 transform = null;

(this.datum == undefined || this.datum == datums.WGS84) {
 // converting from WGS 84
 oldCartesian = this;
 transform = toDatum.transform;

(toDatum == datums.WGS84) {
 // converting to WGS 84; use inverse transform
 oldCartesian = this;
 transform = this.datum.transform.map(p => -p);

(transform == null) {
 // neither this.datum nor toDatum are WGS84: convert this to WGS84 first
 oldCartesian = this.convertDatum(datums.WGS84);
 transform = toDatum.transform;


st newCartesian = oldCartesian.applyTransform(transform);
Cartesian.datum = toDatum;

urn newCartesian;




ies Helmert 7-parameter transformation to 'this' coordinate using transform parameters t.

 is used in converting datums (geodetic->cartesian, apply transform, cartesian->geodetic).

vate
am   {number[]} t - Transformation to apply to this coordinate.
urns {Cartesian} Transformed point.

ansform(t)   {
this point
st { x: x1, y: y1, z: z1 } = this;

transform parameters
st tx = t[0];                    // x-shift in metres
st ty = t[1];                    // y-shift in metres
st tz = t[2];                    // z-shift in metres
st s  = t[3]/1e6 + 1;            // scale: normalise parts-per-million to (s+1)
st rx = (t[4]/3600).toRadians(); // x-rotation: normalise arcseconds to radians
st ry = (t[5]/3600).toRadians(); // y-rotation: normalise arcseconds to radians
st rz = (t[6]/3600).toRadians(); // z-rotation: normalise arcseconds to radians

apply transform
st x2 = tx + x1*s  - y1*rz + z1*ry;
st y2 = ty + x1*rz + y1*s  - z1*rx;
st z2 = tz - x1*ry + y1*rx + z1*s;

urn new Cartesian_Datum(x2, y2, z2);




- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

LonEllipsoidal_Datum as default, Cartesian_Datum as Cartesian, datums, Dms };
```